LUKAS SOMMER

PROGRAMMING HETEROGENEOUS SYSTEMS WITH GENERAL AND DOMAIN-SPECIFIC FRAMEWORKS

TECHNISCHE
UNIVERSITÄT
DARMSTADT

PROGRAMMING HETEROGENEOUS SYSTEMS WITH GENERAL AND DOMAIN-SPECIFIC FRAMEWORKS

Doctoral thesis
by Lukas Sommer, M.Sc.
from Darmstadt, Germany

submitted in fulfilment of the requirements for the
degree of Doctor of Engineering (Dr.-Ing.)

Computer Science Department
Technische Universität Darmstadt

Reviewers
Prof. Dr.-Ing. Andreas Koch
Prof. Dr. Christian Plessl

Date of the oral exam
October 18, 2021

Darmstadt, 2021
D 17

Lukas Sommer: *Programming Heterogeneous Systems with General and Domain-Specific Frameworks*
Technische Universität Darmstadt
Date of the oral exam: October 18, 2021

ERKLÄRUNGEN LAUT PROMOTIONSORDNUNG

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, August 2021

Lukas Sommer

ABSTRACT

As chip manufacturing processes are getting ever closer to what is physically possible, the projections made by Moore's Law and Dennard Scaling no longer hold true, and CPU performance has been stagnating over the last decade.

At the same time, the performance requirements of many important application areas, ranging from machine learning to scientific computing, are increasing at exponential rates, creating a demand that CPUs cannot satisfy anymore.

In order to cater the performance hunger of these applications, computer architects have turned their attention towards *heterogeneous systems*. By combining CPUs with one or multiple accelerators, architects are seeking to provide the necessary performance through specialization and more efficient forms of parallelism.

And while the accelerators have successfully delivered on the promised performance in many cases, programming these heterogeneous systems is becoming increasingly difficult, as developers need to take multiple devices, execution models, and data transfers into account.

Over the course of this cumulative dissertation, we investigate two potential solutions to the enormous challenges of heterogeneous systems programming.

General programming frameworks such as OpenMP define language constructs that reflect important fundamental computing patterns and allow developers to expose an application's parallelism to the compiler for efficient mapping to the target hardware.

Domain-specific programming frameworks, on the other hand, are tailored to a single domain and provide mechanisms to capture the high-level semantics and structure of an application, which is then again mapped to the computational units of the underlying hardware in an efficient fashion.

In this thesis, we discuss the merits of both approaches in detail and show implementation examples for both.

For general programming frameworks, the selection of the most suitable framework for a class of applications and target platform is a crucial step. Using automotive software development as an example, we perform an implementation study to extensively compare three different frameworks. Based on the findings from this implementation study, we identify a number of key factors to assess the suitability of general programming frameworks for applications and target platforms.

One popular general programming framework is OpenMP, and the target offloading capabilities added in recent versions also make it an interesting candidate for targeting FPGAs. To enable the use of OpenMP for FPGA programming, we develop the first-ever prototype for OpenMP target offloading constructs on FPGAs via High-Level Synthesis. Furthermore, we design and implement an execution model and hardware extensions for multi-threaded execution in FPGA accelerators generated through High-Level Synthesis. By combining multi-threaded execution in the generated FPGA accelerators with OpenMP target offloading as programming interface, we do not only significantly reduce idle cycles and improve performance, but also provide an easy-to-use programming interface with intuitive mechanisms for data management.

In order to showcase the implementation of a domain-specific programming framework, we develop a compiler for Sum-Product Networks, a class of machine learning models. By implementing compilation flows for CPUs, GPUs and FPGAs, we are able to cover a wide range of heterogeneous system setups and achieve improvements in inference throughput of multiple orders of magnitude compared to the existing Python-based libraries. The implementation of these toolflows, which for CPU and GPU is based on the modern MLIR framework, also illustrates the role compilers play for the future of heterogeneous computing.

ZUSAMMENFASSUNG

Während sich die Herstellungsprozesse für moderne Computerchips immer stärker den Grenzen des physikalisch Machbaren annähern, treffen die vom Mooreschen Gesetz und der Dennard-Skalierbarkeit gemachten Voraussagen nicht länger zu. Infolgedessen stagniert die Leistungsfähigkeit von CPUs seit über einem Jahrzehnt.

Gleichzeitig steigt der Bedarf nach mehr Leistung in vielen wichtigen Bereichen wie dem maschinellen Lernen oder dem wissenschaftlichen Rechnen exponentiell an, sodass ein Leistungsbedarf entsteht, der von CPUs nicht länger abgedeckt werden kann.

Um diese sich auftuende Lücke zu schließen, haben Computerarchitekten ihr Augenmerk auf heterogene Systeme, die eine CPU mit einem oder mehreren Beschleunigern kombinieren, gelegt. Durch Spezialisierung und effizientere Parallelverarbeitung sollen diese Systeme den entstandenen Bedarf decken.

Die Verwendung von Beschleunigern hat zwar tatsächlich die erhofften Leistungssteigerungen gebracht, jedoch auch die Programmierung dieser Systeme deutlich verkompliziert.

Im Rahmen dieser kumulativen Dissertation diskutieren wir zwei mögliche Lösungen zur Bewältigung der gewaltigen Herausforderungen bei der Programmierung heterogener Systeme.

Allgemeine Programmiermodelle wie OpenMP stellen Sprachkonstrukte bereit, die die wichtigsten Muster in Anwendungen reflektieren und die es so Entwicklern erlauben Parallelismus herauszustellen, sodass er effizient auf die Zielplattform abgebildet werden kann.

Domänen-spezifische Programmiermodelle andererseits sind auf eine einzelne Domäne zugeschnitten und stellen abstrakte Konstrukte zur Erfassung der Struktur und Semantik einer Anwendung bereit. Diese Strukturen und Semantik werden wiederum anschließend auf die Zielhardware abgebildet.

Im Rahmen dieser Arbeit stellen wir die Stärken und Schwächen beider Lösungen detailliert dar und präsentieren Beispiele für die Implementierung beider Lösungen.

Für allgemeine Programmiermodelle ist die Auswahl des richtigen Modells für eine Klasse von Anwendungen und Zielplattformen ein erster wichtiger Schritt. Im Rahmen einer Implementierungsstudie vergleichen wir drei Programmiermodelle im Detail, wobei wir die Softwareentwicklung in der Automobilindustrie als Beispiel heranziehen. Auf Basis der aus der Implementierungsstudie gewonnenen Erkenntnisse identifizieren wir eine Reihe von Schlüsselfaktoren, die zur Beurteilung der Anwendbarkeit eines Programmiermodells für eine Anwendung und Zielplattform verwendet werden können.

Eines der beliebtesten allgemeinen Programmiermodelle ist OpenMP, und die kürzlich hinzugefügten Mechanismen zur Auslagerung von Berechnungen auf Beschleuniger machen OpenMP auch zu einem interessanten Kandidaten für die Programmierung von FPGA-basierten Beschleunigern. Um die Nutzung von OpenMP für die Programmierung von FPGAs zu ermöglichen, entwickeln wir den ersten Prototypen für die OpenMP-Auslagerungsmechanismen auf Basis von High-Level-Synthese. Außerdem entwerfen und implementieren wir ein Ausführungsmodell sowie Hardwareerweiterungen für nebenläufige Ausführung in von FPGA-High-Level-Synthese generierten Beschleunigern. Durch die Kombination der Auslagerungsmechanismen und des nebenläufigen Ausführungsmodells können wir nicht nur den Leerlauf deutlich verringern und die Ausführungsgeschwindigkeit verbessern, sondern auch eine einfach zu nutzende Programmierschnittstelle mit intuitiven Mechanismen zum Datenmanagement bereitstellen.

Um auch die Implementierung eines domänen-spezifischen Programmiermodells zu demonstrieren, entwickeln wir einen Compiler für sogenannte Sum-Product Networks, einer Klasse von Modellen aus dem Bereich des probabilistischen maschinellen Lernens. Durch die Übersetzung für CPU, GPU und FPGA schaffen wir eine breite Abdeckung möglicher heterogener Systeme und erreichen eine Verbesserung des Durchsatzes der ML-Inferenz bis hin zu mehreren Größenordnungen verglichen mit den aktuell verfügbaren, Python-basierten Softwarebibliotheken. Die Implementierung dieser Compiler illustriert so auch die Rolle von Compilern für die Zukunft des heterogenen Rechnens.

PUBLICATIONS

The following publications are part of this cumulative dissertation and can be found in Chapter 8 to Chapter 18.

[1] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. “Work-in-Progress: DAPHNE - An Automotive Benchmark Suite for Parallel Programming Models on Embedded Heterogeneous Platforms.” In: *2019 International Conference on Embedded Software (EMSOFT)*. IEEE, 2019.

[2] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. “Using Parallel Programming Models for Automotive Workloads on Heterogeneous Systems - a Case Study.” In: *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2020. DOI: 10.1109/PDP50117.2020.00010.

[3] Lukas Sommer and Andreas Koch. “OpenMP Device Offloading for Embedded Heterogeneous Platforms - Work-in-Progress.” In: *2020 International Conference on Embedded Software (EMSOFT)*. IEEE, 2020. DOI: 10.1109/EMSOFT51651.2020.9244045.

[4] Lukas Sommer, Jens Korinth, and Andreas Koch. “OpenMP device offloading to FPGA accelerators.” In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2017. DOI: 10.1109/ASAP.2017.7995280.

[5] Lukas Sommer, Julian Oppermann, Jaco Hofmann, and Andreas Koch. “Synthesis of interleaved multithreaded accelerators from OpenMP loops.” In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2017. DOI: 10.1109/RECONFIG.2017.8279823.

[6] Jens Huthmann, Lukas Sommer, Artur Podobas, Andreas Koch, and Kentaro Sano. “OpenMP Device Offloading to FPGAs Using the Nymble Infrastructure.” In: *OpenMP: Portable Multi-Level Parallelism on Modern Systems*. Ed. by Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-58144-2.

[7] Jens Huthmann, Artur Podobas, Lukas Sommer, Andreas Koch, and Kentaro Sano. “Extending High-Level Synthesis with High-Performance Computing Performance Visualization.” In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020. DOI: 10.1109/CLUSTER49012.2020.00047.

[8] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. “Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-Based Accelerators.” In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018. DOI: 10.1109/ICCD.2018.00060.

[9] Micha Ober, Jaco Hofmann, Lukas Sommer, Lukas Weber, and Andreas Koch. “High-Throughput Multi-Threaded Sum-Product Network Inference in the Reconfigurable Cloud.” In: *2019 IEEE /ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 2019. DOI: 10.1109/H2RC49586.2019.00009.

[10] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. “Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs.” In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020. DOI: 10.1109/FCCM48280.2020.00020.

[11] Lukas Sommer, Cristian Axenie, and Andreas Koch. “SPNC: Fast Sum-Product Network Inference.” In: *2021 International Workshop on IoT, Edge, and Mobile for Embedded Machine Learning (ITEM)*. Cham: Springer International Publishing, 2021.

Overall, the author of this thesis has contributed to the following peer-reviewed publications.

[1] Lukas Sommer, Julian Oppermann, and Andreas Koch. "C-based Synthesis of Area-Efficient Accelerators for OpenMP Worksharing Loops." In: *Second International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. Nov. 2016.

[2] Julian Oppermann, Lukas Sommer, and Andreas Koch. "SpExSim: assessing kernel suitability for C-based high-level hardware synthesis." In: *Journal of Supercomputing*. July 2017.

[3] Lukas Sommer, Jens Korinth, and Andreas Koch. "OpenMP device offloading to FPGA accelerators." In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2017. DOI: 10.1109/ASAP.2017.7995280.

[4] Lukas Sommer, Julian Oppermann, Jaco Hofmann, and Andreas Koch. "Synthesis of interleaved multithreaded accelerators from OpenMP loops." In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2017. DOI: 10.1109/RECONFIG.2017.8279823.

[5] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Oliver Sinnen, and Andreas Koch. "Dependence Graph Preprocessing for Faster Exact Modulo Scheduling in High-level Synthesis." In: *Intl. Conf. on Field Programmable Logic and Applications (FPL)*. Sept. 2018.

[6] Lukas Sommer, Julian Oppermann, Jens Korinth, and Andreas Koch. "Offloading OpenMP Target Regions to FPGA Accelerators Using LLVM." In: *2018 European LLVM Developers Meeting*. Apr. 2018.

[7] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-Based Accelerators." In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018. DOI: 10.1109/ICCD.2018.00060.

[8] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. "Automatic Synthesis of FPGA-based Accelerators for the Sum-Product Network Inference Problem." In: *ICML 2018 Workshop on Tractable Probabilistic Models (TPM)*. ICML. July 2018.

[9] Michael Halkenhäuser and Lukas Sommer. "An alternative OpenMP Backend for Polly." In: *2019 European LLVM Developers Meeting*. Apr. 2019.

[10] Robin Kruppe, Julian Oppermann, Lukas Sommer, and Andreas Koch. “Extending LLVM for Lightweight SPMD Vectorization: Using SIMD and Vector Instructions Easily from Any Language.” In: *2019 International Symposium on Code Generation and Optimization*. Feb. 2019.

[11] Micha Ober, Jaco Hofmann, Lukas Sommer, Lukas Weber, and Andreas Koch. “High-Throughput Multi-Threaded Sum-Product Network Inference in the Reconfigurable Cloud.” In: *2019 IEEE /ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 2019. DOI: 10.1109/H2RC49586.2019.00009.

[12] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Andreas Koch, and Oliver Sinnen. “Exact and Practical Modulo Scheduling for High-Level Synthesis.” In: *TRETS* 12.2 (May 2019), 8:1–8:26. DOI: 10.1145/3317670.

[13] Julian Oppermann, Lukas Sommer, Lukas Weber, Melanie Reuter-Oppermann, Andreas Koch, and Oliver Sinnen. “SkyCastle: A Resource-Aware Multi-Loop Scheduler for High-Level Synthesis.” In: *International Conference on Field-Programmable Technology (FPT)*. Sept. 2019.

[14] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. “EPHoS: Evaluation of Programming - Models for Heterogeneous Systems.” In: *FAT-Schriftenreihe 317* (June 2019).

[15] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. “Work-in-Progress: DAPHNE - An Automotive Benchmark Suite for Parallel Programming Models on Embedded Heterogeneous Platforms.” In: *2019 International Conference on Embedded Software (EMSOFT)*. IEEE, 2019.

[16] Lukas Weber, Lukas Sommer, Julian Oppermann, Alejandro Molina, Kristian Kersting, and Andreas Koch. “Resource-Efficient Logarithmic Number Scale Arithmetic for SPN Inference on FPGAs.” In: *International Conference on Field-Programmable Technology (FPT)*. Dec. 2019.

[17] Markus Glaser, Charles Macfarlane, Benjamin May, Sven Fleck, Lukas Sommer, Jann-Eve Stavesand, Christian Weber, Duong-Van Nguyen, Enda Ward, Ilya Rudkin, Stefan Milz, Rainer Oder, Frank Böhm, Benedikt Schonlau, Oliver Hupfeld, and Andreas Koch. “Open Standards Enable Continuous Software Development in the Automotive Industry.” In: *AutoSens Brussels (Invited Whitepaper)* (Sept. 2020).

[18] C. Heinz, J. A. Hofmann, L. Sommer, and A. Koch. “Improving Job Launch Rates in the TaPaSCo FPGA Middleware by Hardware/Software-Co-Design.” In: *2020 IEEE/ACM Interna-*

tional Workshop on Runtime and Operating Systems for Supercomputers (ROSS). Nov. 2020, pp. 22–30. DOI: 10.1109/ROSS51935.2020.00008.

[19] Jens Huthmann, Artur Podobas, Lukas Sommer, Andreas Koch, and Kentaro Sano. "Extending High-Level Synthesis with High-Performance Computing Performance Visualization." In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020. DOI: 10.1109/CLUSTER49012.2020.00047.

[20] Jens Huthmann, Lukas Sommer, Artur Podobas, Andreas Koch, and Kentaro Sano. "OpenMP Device Offloading to FPGAs Using the Nymble Infrastructure." In: *OpenMP: Portable Multi-Level Parallelism on Modern Systems*. Ed. by Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-58144-2.

[21] Lukas Sommer and Andreas Koch. "OpenMP Device Offloading for Embedded Heterogeneous Platforms - Work-in-Progress." In: *2020 International Conference on Embedded Software (EMSOFT)*. IEEE, 2020. DOI: 10.1109/EMSOFT51651.2020.9244045.

[22] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. "DAPHNE - An Automotive Benchmark Suite for Parallel Programming Models on Embedded Heterogeneous Platforms." In: *Future Automotive HW/SW Platform Design (Dagstuhl Seminar 19502)*. Ed. by Dirk Ziegenbein, Selma Saidi, Xiaobo Sharon Hu, and Sebastian Steinhorst. Vol. 9. Dagstuhl Reports. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Feb. 2020, pp. 28–66. DOI: 10.4230/DagRep.9.12.28. URL: https://drops.dagstuhl.de/opus/volltexte/2020/12010.

[23] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. "Using Parallel Programming Models for Automotive Workloads on Heterogeneous Systems - a Case Study." In: *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2020. DOI: 10.1109/PDP50117.2020.00010.

[24] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. "Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs." In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020. DOI: 10.1109/FCCM48280.2020.00020.

[25] Carsten Heinz, Jaco Hofmann, Jens Korinth, Lukas Sommer, Lukas Weber, and Andreas Koch. "The TaPaSCo Open-Source Toolflow." In: *Journal of Signal Processing Systems* (May 2021). ISSN: 1939-8115. DOI: 10.1007/s11265-021-01640-8. URL: https://doi.org/10.1007/s11265-021-01640-8.

[26] Hanna Kruppe, Lukas Sommer, Lukas Weber, Julian Oppermann, Cristian Axenie, and Andreas Koch. “Efficient Operator Sharing Modulo Scheduling for Sum-Product Network Inference on FPGAs.” In: *Intl. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*. Springer International Publishing, July 2021.

[27] Lukas Sommer, Cristian Axenie, and Andreas Koch. “SPNC: Fast Sum-Product Network Inference.” In: *2021 International Workshop on IoT, Edge, and Mobile for Embedded Machine Learning (ITEM)*. Cham: Springer International Publishing, 2021.

[28] Lukas Sommer, Michael Halkenhäuser, Cristian Axenie, and Andreas Koch. “SPNC: Accelerating Sum-Product Network Inference on CPUs and GPUs.” In: *2021 IEEE 32th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2021.

[29] Lukas Weber, Lukas Sommer, Leonardo Solis-Vasquez, Tobias Vinçon, Christian Knödler, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. “A Framework for the Automatic Generation of FPGA-based Near-Data Processing Accelerators in Smart Storage Systems.” In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. June 2021, pp. 136–143. DOI: 10.1109/IPDPSW52791.2021.00028.

ACKNOWLEDGMENTS

First of all, I am sincerely grateful to Prof. Dr.-Ing. Andreas Koch for being such a great advisor and mentor over all the years, and the many invaluable lessons I learned from him in academia and beyond.

I would also like to thank Prof. Dr. Christian Plessl for readily accepting to be the second assessor for this thesis and taking the time for the review.

I was lucky to have great teammates in all my former and current colleagues at the Embedded Systems and Applications Group at TU Darmstadt, and would like to thank them for many interesting discussions and contributions to my research. I am especially grateful to Dr.-Ing. Julian Oppermann, who has supported me through my bachelor's and master's theses, and with whom I had the pleasure to work many interesting projects, Lukas Weber for working on Sum-Product Networks with me, and Carsten Heinz for always being available on short notice for all kinds of hardware and infrastructure issues.

I am also deeply grateful to my family, friends, and girlfriend, for their continued support and for always being there when I needed them.

Last but not least, I would like to thank Dr.-Ing. Cristian Axenie for enabling me to participate in the Softwarecampus program and being my mentor.

Experiments for this research were conducted on the Lichtenberg high-performance computing cluster at TU Darmstadt. I would like to thank Xilinx, Inc. for supporting my work by hardware and software donations.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ADAS	Advanced Driver Assistance Systems
API	Application Programming Interface
ARM	ARM Limited, formerly Advanced RISC Machines Ltd.
ASIC	Application-Specific Integrated Circuit
AWS	Amazon Web Services
CAPI	Coherent Accelerator Processor Interface
CCIX	Cache Coherent Interconect for Accelerators
CGRA	Coarse-Grained Reconfigurable Array
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CXL	Compute Express Link
DAG	Directed Acyclic Graph
DNN	Deep Neural Network
DPU	Data Processing Unit
DRAM	Dynamic Random Access Memory
DSE	Design-Space Exploration
DSP	Digital Signal Processor
ECU	Electronic Control Unit
FPGA	Field Programmable Gate Array
GPGPU	General-Purpose computing on Graphics Processing Units
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
HBM	High-Bandwith Memory
HLS	High-Level Synthesis
HPC	High-Performance Computing
ILP	Instruction-Level Parallelism
ISA	Instruction Set Architecture
LLVM	LLVM Compiler Infrastructure, formerly Low-Level Virtual Machine
MLIR	Multi-Level Intermediate Representation
MPI	Message Passing Interface
NRE	Non-Recurring Expenses

PCI	Peripheral Component Interconnect
PGM	Probabilistic Graphical Model
SIMD	Single Instruction Multiple Data, Flynn's taxonomy
SIMT	Single Instruction Multiple Threads
SPN	Sum-Product Network

Part I

SYNOPSIS

1 INTRODUCTION

In the last years, chip manufacturers have driven transistor scaling ever closer to what is physically feasible. While manufacturers are confident that they will be able to sustain this trend for at least another decade in the future, using techniques such as *gate-all-around* and nano-sheets for technology nodes as small as two nanometers [30, 31], operating this close to the physical limit also means that one has to cope with effects that have been negligible in the past.

As a consequence of these effects, projections on the development of chip transistor count and energy consumption, captured in famous "laws" such as Moore's law [43] or Dennard scaling [11], do not hold true anymore to their full extent and CPU performance is stagnating.

At the same time, however, performance demand is anything but stagnating. Instead, the performance requirements in many important areas of computing, such as machine learning, are *increasing* at an exponential rate, with the consequence that CPUs are no longer able to cater the performance hunger of many important applications.

In order to address this gap between CPU performance and application performance demand, computer architects are turning their attention towards *heterogeneous systems*, which combine a CPU with one or multiple specialized accelerators. This trend can be illustrated by looking at the Top500 list, a collection of the world's 500 fastest supercomputers: While in 2011, only 19 out of these 500 systems (3.8 %) were using a Graphics Processing Unit (GPU) as accelerator [22], only ten years later in 2021, 147 out of 500 systems (29.4 %) are using some sort of accelerator [23]. In the list of the Top100 systems, the share of systems using accelerators is even higher at 45 % [23].

Accelerators promise improved performance by focusing on a specific class of problems or even a single domain rather than using generic designs, and by providing abundant parallelism for applications.

While accelerators in many cases, e.g., machine learning training, have successfully delivered on this promise, heterogeneous systems are notoriously difficult to program compared to single-CPU systems [7]. Developers need to decide where to run which code and how to move data around in the system. In addition, the different accelerators and their underlying architectures and execution models often require very different programming styles.

In order to make programming of such heterogeneous systems more accessible for developers, allowing them to leverage the *full* potential of accelerators, we are going to investigate two alternative approaches to programming such systems throughout the course of this thesis.

The first possible way to reduce the friction involved in programming heterogeneous systems is using *general* parallel and heterogeneous programming frameworks, such as OpenMP or OpenCL. Many of these programming frameworks have been established in the High-Performance Computing community for many years and have evolved over time as system design and components changed. Typically, these programming frameworks extend an existing serial programming language such as C or C++ with a number of language constructs and Application Programming Interface (API) functions that provide developers with a convenient way to target accelerators for execution, manage memory and data transfers, and allows them to expose parallelism in the application to the compiler and runtime system.

Domain-specific programming frameworks on the other hand are tailored for a single domain or class of applications. By providing domain-specific programming abstractions matching the core computational operations of that domain, these frameworks capture the high-level semantics and computational structure of an application and, when carefully designed, provide abstractions that feel *natural* to a domain expert. Domain-specific frameworks can either be implemented in the form of a standalone domain-specific language or can be embedded into an existing programming language as library functions.

We are going to discuss both alternatives and their respective merits in detail as part of this thesis and will showcase examples of how both kinds of frameworks can be implemented and used successfully to target heterogeneous systems.

In Chapter 2, we will first discuss some of the developments that have made the use of heterogeneous systems inevitable for many modern applications, and will provide insight into how accelerators can deliver better performance than CPUs and important characteristics of heterogeneous systems.

Chapter 3 will then highlight some of the biggest challenges developers are confronted with when dealing with heterogeneous systems, how general and domain-specific frameworks can help to overcome these obstacles and discuss the respective advantages and disadvantages of both solutions.

In Chapter 4, we are giving an overview of the publications which, as part of this cumulative dissertation, can be found in Part ii, and will provide insights into which aspects of general programming frameworks must be taken into consideration when selecting the right programming framework for a specific class of applications.

An overview of the publications in Part iii can be found in Chapter 5, where we will demonstrate how a general programming framework, in this case OpenMP, can be used to target a specific class of accelerators, namely Field Programmable Gate Arrays (FPGAs), in a heterogeneous system. The publications in Part iii also illustrate how the necessary compiler support and runtime extensions can be implemented to support such devices.

The second alternative to heterogeneous system programming, namely domain-specific programming frameworks, is discussed in the publications in Part iv and summarized in Chapter 6. After giving some background information on *Sum-Product Networks*, a class of machine learning models, we will demonstrate how automatic toolflows and compilation starting from a domain-specific programming interface facilitate exploiting the CPU's full features set and allow targeting accelerators such as GPUs or FPGAs in a heterogeneous system. In this case, usage of the domain-specific programming framework in the form of a Python library, and the underlying toolflows and compiler developed as part of this thesis, can lead to significant improvements in inference execution time.

Chapter 7 concludes this thesis, summarizes its contributions and gives an outlook on the future of heterogeneous systems and the corresponding programming frameworks.

2 HETEROGENEOUS SYSTEMS

As discussed in Chapter 1, modern computing systems become increasingly heterogeneous, integrating a growing number of specialized accelerators into the system. In the first part of this chapter (Section 2.1), we look at some developments in hardware capabilities and computer architecture that have led to this trend. After that, we investigate what sets heterogeneous systems apart from "traditional" computer systems, which components they contain and how they are composed (Section 2.2).

2.1 THE NEED FOR HETEROGENEOUS SYSTEMS

Given the trend towards heterogeneous systems, the question arises: *Why* are systems becoming more heterogeneous, and why is there a need for such accelerators? At first glance, using a heterogeneous system makes programming much more complicated, as we will illustrate in Chapter 3, and the execution often requires additional data movements between different components of the system, as will be discussed in Section 2.2.

The answer on why computer architects and programmers spend all this effort on these complicated systems is actually quite simple: Heterogeneous systems and, in particular, specialized accelerators are *needed* to satisfy the demand for computational power for many of the most relevant computational applications of our time, such as machine learning or climate modelling.

»The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.« *— Gordon E. Moore, 1965 in [43]*

»The new slope might approximate a doubling every 2 years, rather than every year, by the end of the decade.« *— Gordon E. Moore, 1975 in [44]*

In order to see why the computational demand of such applications often cannot be catered by "classical" CPU-only systems, we can have a look at the evolution of the computational power of CPUs over the last decades as shown in Fig. 2.1.

The plot shows a period of moderate increase in computing performance in the 1980s, followed by a period of enormous growth in the time between 1986 and roughly 2002, driven by two main factors.

The first development that enabled such growth in performance has become known as *Moore's Law*. In 1965 [43], Gordon Moore had, based on previous development up to this point, predicted that technological advancements would allow to double the number of transistors that could *economically* be manufactured onto a device roughly every year. Later on in 1975, he revised this projection, now predicting a doubling every two years [44].

The second development, called *Dennard Scaling*, is just as important for this rapid growth. Dennard et al. [11] found that, when scaling

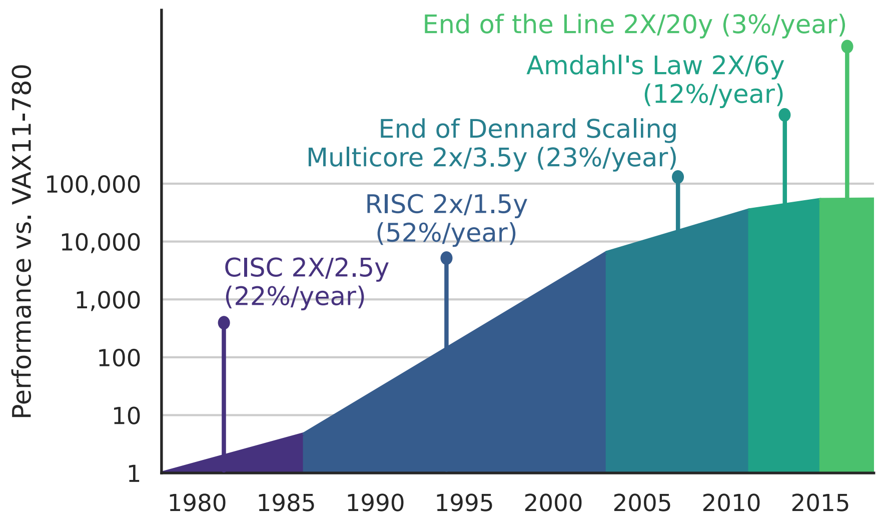


Figure 2.1: Development of computer system performance, measured using integer programs (SPECintCPU). Replotted from [21] for higher resolution.

down the transistor size and accordingly also the supply voltage, the electric properties of the individual transistor change in a way such that the power density of the device remains constant.

»[...] the power density remains constant. Thus, even if many more circuits are placed on a given integrated circuit chip, the cooling problem is essentially unchanged.« — *Robert H. Dennard et al., 1974 in [11]*

These two developments made more and more transistors available to computer architects, allowing them to double performance of CPUs and computing systems every 1.5 years. Next to scaling up the operating frequency of the circuit, most of the performance gains were achieved through so-called Instruction-Level Parallelism (ILP): Computer architects would employ the additional available transistors to simultaneously process multiple instructions in every clock cycle.

Yet, in the early 2000s, the rapid improvement of CPU performance began to slow down, as the actual characteristics of manufactured chips began to deviate from the projections made by Dennard Scaling. The main reason for this development was that supply voltage could not be scaled accordingly to transistor size anymore, due to parasitic effects and static dissipation, causing supply voltage to reach a plateau at about 1 V. Consequently, the power density did not remain constant, but rather started to increase, making energy efficiency an important concern for computer architects.

The increasing power density and, associated with it, the cooling challenges, also meant that frequency could not be scaled up as before. So, even though large manufacturers such as Intel had predicted clock speeds of up to 10 GHz before, clock speed began to saturate around 4 GHz, where it has remained to date.

It became clear that, in order to further increase the performance of computer systems, CPUs must make better use of the energy. Unfortunately, exploiting instruction level parallelism is not very energy efficient, due to the fact that, in order to find a sufficient number of parallel instructions, modern CPUs must speculate on the effects of branch instructions, which make up for roughly 25 % of most applications [21]. However, for any non-trivial input program, branch prediction will misspeculate in some cases, causing the instructions that have already entered the pipeline to be evicted. This does not only waste the energy that was already spent on now discarded instructions, but also requires additional energy to reset the pipeline. The investigation by Hennessy and Patterson in [21] shows that, over a selection of different workloads, on average 19 % of instructions are discarded after beginning execution.

In order to identify designs that would waste less energy, computer architects turned their attention to *multi-core* architectures. By replicating the same core multiple times, processing can be parallelized. Instead of relying on ILP to find parallelism, the programmer was now responsible to identify and denote the parallelism in an application, wasting less energy on misspeculated branches.

With the use of multi-core architectures, starting around the year 2004, it was possible to keep on improving CPU performance, although the improvement rates were smaller than before. However, two fundamental issues limit the improvements that can be achieved through parallelizing workloads across multiple cores.

First, multi-cores cannot fully solve the problems caused by the break-down of Dennard scaling, i.e., integrating more cores still leads to a higher power density, up to a point where the resulting heat cannot be removed with standard packaging and cooling systems.

»A fairly obvious conclusion that can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.«
— Gene M. Amdahl, 1967 in [4]

Secondly, all non-trivial programs always contain a serial portion of code, which cannot be fully parallelized (e.g., input/output). Yet, as the number of cores increases and the execution time spent on the parallel portion of an application keeps decreasing, the serial portion of the code starts to dominate the execution time and eventually limits the maximum speedup that can be achieved through parallelization across multiple cores. This effect has become known as *Amdahl's Law* [4].

Next to these fundamental challenges, recent years have also witnessed a significant slow-down in Moore's Law, a development that also Gordon E. Moore himself had predicted [45]. According to Hennessy and Patterson, the original projection of Moore's Law was already off by a factor of 15x by the year 2018 [21].

These challenges taken together, i.e., the end of Dennard Scaling, the slow-down of Moore's Law, and the effects of Amdahl's Law on parallel execution performance, have slowed down the development of CPU performance to as little as three percent per year.

This stagnating CPU performance is however met by an ever-increasing demand for more computational power by many highly relevant compute applications. An example for such a development can be found

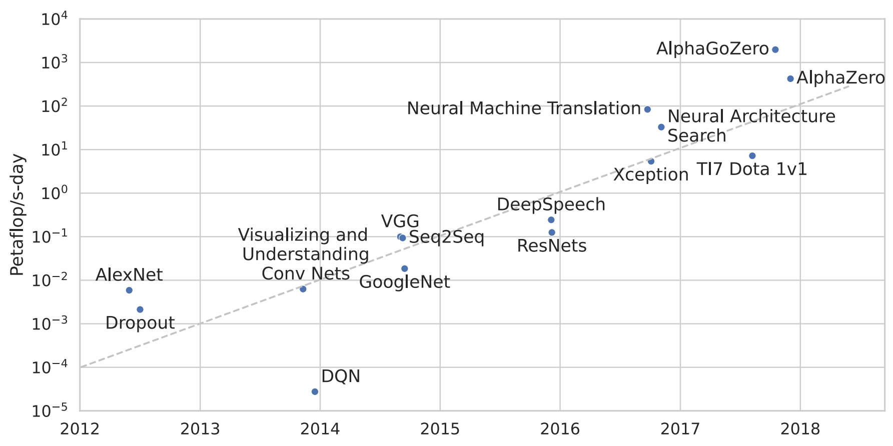


Figure 2.2: Total amount of compute performance required to train selected machine learning models. Note the log-scale on the y-axis. Replotted from [2] for higher resolution.

in Fig. 2.2, showing how the computational power required to train the biggest neural networks has developed over the last decade.

Since Krizhevsky et al. [35] used one of the now ubiquitous neural networks to win the ImageNet classification challenge in 2012, the performance requirements of the largest neural network training runs have increased by a factor of 300,000x over the years, which is equivalent to a doubling every 3.4 months.

And machine learning is far from the only application domain, which needs this kind of computational power. Other High-Performance Computing (HPC) application areas such as climate modeling or medical research are also in need for more computational performance, e.g., to further increase the resolution of climate modeling.

This leaves the world of computer architecture in the unfortunate situation that application requirements are increasing, while the development of CPUs, which have provided that computational power for many years, is stagnating.

As a consequence, computer architects are turning towards *heterogeneous* systems. In the next section, we will discuss how heterogeneous architectures allow them to make better use of the available transistors and build more energy-efficient systems.

2.2 CHARACTERISTICS OF HETEROGENEOUS SYSTEMS

The term *Heterogeneous Systems* is a rather broad term, and is, in different contexts, used to refer to a whole variety of very different systems.

In this thesis, we will focus on heterogeneous systems that combine a CPU as a General-Purpose Processor (GPP) with one or multiple accelerators in a single system. There are other sources of heterogeneity in a computer system that we will explicitly not consider, for example heterogeneous multi-cores [36]. These CPUs use the same Instruction Set Architecture (ISA) for multiple, different cores combined into one multi-core processor, such as ARM's big.LITTLE architecture [16], which is nowadays very common in the embedded and mobile sector.

Yet, even with these sources of heterogeneity left out from consideration, there is still a wide range of very different accelerators that can be combined with a CPU, and potentially other accelerators, in a heterogeneous system.

2.2.1 *Types of Accelerators*

One very common choice for accelerators in a heterogeneous system are **GPUs**. GPUs were initially designed solely for graphics processing and as such are already an example of a domain-specific accelerator in a heterogeneous system.

However, it was quickly realized that the computational pattern for which GPUs have been optimized, i.e., the massively parallel processing of many elements, can also be found in many non-graphics applications. This realization led to a trend coined as General-Purpose computing on Graphics Processing Units (GPGPU), where GPUs where employed to accelerate non-graphics workloads. Starting in the early 2000s, GPU vendors started to provide corresponding tools, e.g., Nvidia CUDA which was first released in 2007.

Nowadays, GPU vendors explicitly include functional blocks in their designs to facilitate the use of GPUs also for non-graphics tasks. Nvidia, for example, includes Tensor cores in modern GPUs to make machine learning training and inference more efficient.

And while the programming of GPUs initially required strict adherence to the SIMT (single-instruction-multiple-threads) execution model, vendors have made GPUs more flexible in this regard, too. Modern generations of Nvidia GPUs now use a dedicated program counter for each thread (not per warp as before) [47], making effects such as control-flow divergence less of an issue.

Another interesting choice for accelerators are **Application-Specific Integrated Circuits (ASICs)**, which provide huge flexibility. Therefore, ASICs can be found in heterogeneous systems in various shapes. For some compute-intensive tasks which remain largely unchanged over the lifetime of a device, task-specific ASICs are integrated into the system. One example of such ASICs, which can be found in virtually every smartphone nowadays, are specialized functional units for audio/video encoding or wireless/broadband communication. Another example for an even more specialized system is Anton 2 [56], whose architecture is tailored for molecular dynamics simulation and the fine-grained event-based processing required for this kind of application.

In other cases, ASICs are designed to cover a range of different applications from one domain. In these cases, the flexibility of ASICs allows to match the characteristics of the application domain directly in the hardware design, e.g., with regard to data formats and memory hierarchy. However, as the Non-Recurring Expenses (NRE) for ASIC design are extremely high, designers in these cases must make sure to design the ASIC in a way that allows reuse for different applications from the same domain, while still implementing a design capable of high performance and energy efficiency for applications of that domain.

Examples for this kind of accelerator can be found in machine learning training and inference: Given the growing importance of neural network approaches to machine learning, a number of companies have built specialized accelerators for this domain, e.g., Google's TPU or Graphcore's IPU.

In cases where the high cost of ASIC design and development are not justifiable or where adaption to a concrete application in the field is beneficial, **FPGAs** can be a great alternative to ASICs as accelerators in a heterogeneous system. Next to configurable logic blocks and interconnect, which allow to implement the desired functionality on a fine-grained bit-level, modern FPGAs also contain functional units such as DSP blocks for basic arithmetic operations, fast on-chip memories or fixed functionality for access to off-chip memories and fast networking.

Advancements with regard to FPGA tooling, e.g., in the area of High-Level Synthesis (HLS) for C/C++ or domain-specific tooling for machine learning, have made FPGAs much more accessible, also to non-hardware experts, and have helped the adoption of FPGAs as accelerators in datacenters as well as embedded applications.

While the FPGA's re-configurable logic allows parallelizing applications in the spatial dimension as well as, through pipelining, in the temporal dimension, the difference in clock frequency remains a challenge. While other heterogeneous system components such as CPUs, GPUs, or ASICs typically operate at multiple *Gigahertz* frequencies, recent FPGA generations are still limited to several hundred *Megahertz*.

Next to the three main kinds of accelerators explained above, there are a number of other interesting acceleration options, such as Coarse-Grained Reconfigurable Arrays (CGRAs), which provide reconfigurability at the word-level, or Data Processing Units (DPUs), which, in data-centers, are often combined with smart network interface cards (SmartNIC) and used to offload storage or computation tasks.

2.2.2 *Accelerator Architecture Motifs*

Despite their very different characteristics and the fact that they represent very diverse points on the flexibility-cost tradeoff-curve, all accelerator options discussed so far share a number of common hardware architecture motifs that allow them to make good use of the available transistors and provide high performance and energy efficiency. Dally et al. [9] identified four main motifs used for performance and energy efficiency gain and which, in different shape, can be found in all accelerators.

Parallelism is the first and most obvious way to achieve higher performance. Modern CPUs, through multi-issue pipelines, multiple cores and SIMD extensions, offer only limited parallelism. Accelerators such as GPUs, on the other hand, offer higher degrees of parallelism with hundreds of cores through a SIMD or SIMT programming model.

For domain-specific accelerators, it is also often possible to design highly parallel architectures and implementations for the most common computational kernels in a domain. One example for such specialization is the usage of an optimized systolic array structure for matrix multiplication. This operation is the most common kernel in deep neural network training and inference, therefore the specialized systolic array is at the heart of Google's TPU accelerator [32].

The second common motif, found in many accelerators, is **data specialization**. Being a general-purpose processor, CPUs typically support a variety of datatypes for computation and need to have all of the corresponding hardware units available. Accelerators, on the other hand, can focus on just the most important datatypes for a class of applications. One example for this specialization can be found in accelerators or processing elements (e.g., Nvidia's TensorCores) for machine learning: As a small number of integer bits, ranging from eight bit down to a single bit, is usually sufficient for weight representation in Deep Neural Network (DNN) inference, many DNN accelerators are specialized to use these extremely efficient formats.

Another example for data specialization can be found in floating-point computations: The IEEE-754 format specifies a number of special cases, e.g., `NaN` and `-Inf/+Inf`. Logic for detecting and handling these special cases often makes up for a significant portion of a floating-point operator's hardware footprint and latency. As the evaluation in Chapter 17 shows, removing this logic if it is not required by an application can significantly reduce the hardware area per operator instance. Furthermore, data specialization to the data formats most relevant to an application or domain can also reduce the number of clock cycles and the amount of power required to perform an arithmetic operation, resulting in higher performance and less cooling issues.

The third motif addresses one of the most urgent problems of modern CPUs: Accesses to off-chip memory (e.g., DRAM) do not only take many clock cycles to complete, but also cost a lot of energy. A single access to off-chip memory can consume multiple orders of magnitude more energy than a single arithmetic instruction executed by the CPU. Horowitz [27] determined that a single access to off-chip DRAM can consume 1,300x the energy of a 32-bit floating-point addition.

CPUs try to address this problem in a generic fashion by using an elaborate, implicitly managed cache hierarchy with multiple levels. Caches improve the energy efficiency of accesses to cached data significantly, with an access to a 1 MB cache "only" consuming about 100x of a 32-bit floating point addition [27]. However, while caching works well for some applications, it can actually harm performance for parallel execution in other cases. One example for an effect that can harm performance is *false sharing*, where one core is forced to repeatedly read a cache line from main memory, due to modifications made by another core to an unrelated data item sharing the same cache line.

Accelerators therefore often use **local and optimized memories**, such as fast on-chip memories as scratchpad memories or buffers for intermediate results. In contrast to CPUs, the memory hierarchy is often also explicitly managed, either by the programmer (GPUs, FPGAs), or the compiler and runtime system (ML accelerators). This explicit management avoids effects such as false sharing, or the eviction of frequently accessed data by less frequently accessed data, to improve data locality.

Another way for accelerators to improve the efficiency and performance over CPUs is through **reduced overhead**. In modern, superscalar, out-of-order CPUs, a significant amount of hardware is dedicated to the complex control logic required for instruction initialization and completion. By removing most of this overhead and replacing it with a pipeline specialized to the instructions actually used by the domain applications, specialized accelerators can simplify program interpretation, saving energy and hardware area. For example, as Horowitz showed in [27], only a small portion of the energy consumed by an instruction is used by the actual computation (e.g., addition). Another significant portion is required for instruction access and decoding. By using a SIMD (or SIMT) execution model instead, the energy for instruction access and decoding can be amortized across all lanes (or threads), improving energy efficiency.

Specialized instructions, for example for an entire matrix multiplication, additionally allow to reduce the runtime overhead. While a CPU needs to execute dedicated, additional instructions, e.g., to maintain loop counters, an accelerator can integrate such management directly into hardware. An example for such a technique is *zero-overhead looping* deployed in DSP devices. The number of such specialized instructions should be limited to a few instructions highly relevant to the domain/application, to avoid the problems encountered with Complex Instruction Set Computers (CISCs).

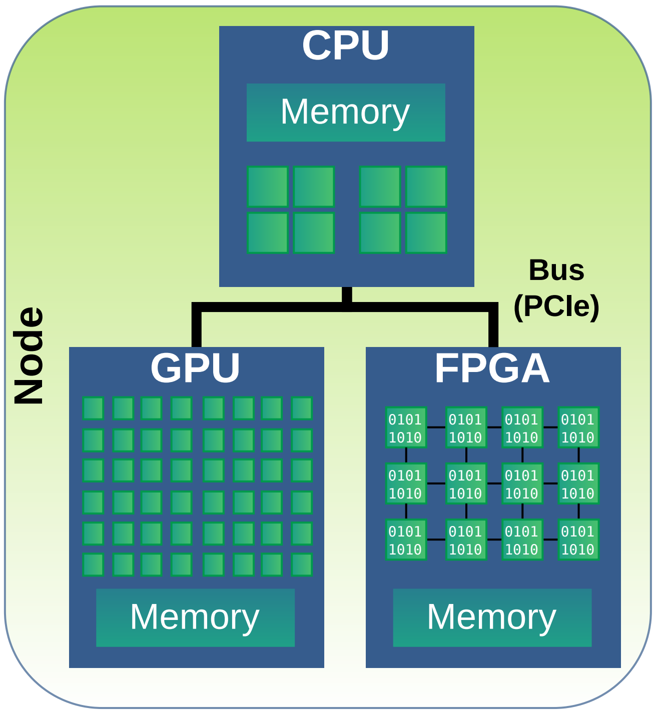


Figure 2.3: Schematic of a typical *data-center scale* heterogeneous system. Accelerators are equipped with decidated memories and do not have direct access to CPU main memory. Explicit data transfers need to be performed via the bus (e.g., PCIe).

2.2.3 *Heterogeneous System Composition*

When composing a heterogeneous system, the choice of accelerators that, next to the CPU as the general-purpose processor, go into the system, is not the only choice to be made. The coupling of the different components and the communication among them offer additional degrees of freedom that need to be considered.

On the datacenter-scale, despite efforts such as Intel's Harp or the Enzian system [3], accelerators are typically manufactured as separate components and coupled comparably loosely with the CPU. Such a system is sketched in Fig. 2.3. Accelerators such as FPGAs and GPUs are usually integrated in the form of extension cards and can communicate with the CPU through a bus system such as PCI Express. The accelerators each have their own on-chip and off-chip memory. In such a system, explicit data-transfers are required to make data available for computation for the CPU or the different accelerators. This is due to the fact that the accelerators cannot directly access the CPUs main memory (or resolve the virtual addressing), and the CPU can only access accelerator memory through comparably slow memory-mapped I/O (MMIO).

These data transfers have to be taken into consideration when assessing the acceleration potential of an application on a heterogeneous

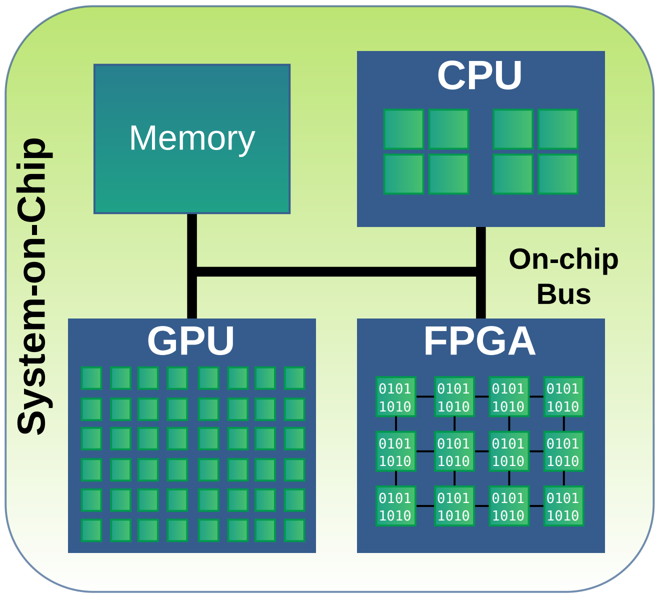


Figure 2.4: Schematic of a typical *embedded* heterogeneous system. CPU and accelerators share the same *physical* memory via the on-chip bus of the System-on-Chip.

system, as they can consume significant amounts of execution time, in particular since the bus system, e.g., PCIe Gen 3 with 12-16 GB/s, often have a significantly smaller bandwidth than the on-board, off-chip memory, e.g., HBM with up to 2 TB/s on an Nvidia A100 GPU.

Yet, the data transfers do not only affect performance, but also place an additional burden on the programmer of such a system, who needs to explicitly manage data movements when offloading computations to an accelerator. Therefore, and in order to improve efficiency of data transfers, initiatives such as Coherent Accelerator Processor Interface (CAPI), Compute Express Link (CXL) or Cache Coherent Interconect for Accelerators (CCIX) have defined protocols for cache-coherent memory access across different components. By providing a transparent view of a single memory space across CPUs and accelerators, they lift the burden of explicit data transfers from the programmer. Yet, data transfers still need to be performed, and the protocols often still rely on PCI Express as the underlying physical and electrical connection.

In embedded and mobile devices, on the other hand, CPU and accelerators are typically coupled much more tightly and manufactured onto a single SoC, as sketched in Fig. 2.4. In such a system, CPU and accelerators may *physically* share the same main memory, and no data transfers have to be performed before offloading computations to an accelerator. This changes the characteristics of the heterogeneous system significantly, as offloading can now be beneficial for much smaller tasks, as we will discuss in Chapter 4.

However, the fact that the components physically share the main memory also means that they compete for the bandwidth of the single memory interface, which can have a negative impact on performance.

3 THE PROGRAMMING CHALLENGE

With the new opportunities that open up when combining a CPU with one or multiple accelerators as discussed in the previous chapter, also comes a plethora of new challenges when considering how to program such systems.

During the era where Moore's Law and Dennard Scaling still were in full effect, the only thing that programmers had to do to increase the performance of their code was to wait a sufficient amount of time and eventually buy a new CPU. For the identification of instruction-level parallelism, which, next to advances in processing technology and higher operating frequency, was the main source of increased performance, the programmer could rely on the CPU scheduler. In addition, the compiler, through transformations such as reassociation of arithmetic computations, also tried to expose more ILP in a program.

However, starting with the introduction of multi-core CPUs in the early 2000s, the responsibility for identifying and denoting parallelism in a program started to shift towards the programmer. In order to be able to employ all the resources of modern parallel and heterogeneous systems, the input program needs to capture the parallel semantics of the application and other sources of potential performance gains.

Established, popular programming languages such as C or C++, however, provide only very limited mechanisms to denote parallelism in an application. Approaches for auto-parallelization of programs written in these languages, for example using the polyhedral model inside the compiler [17], work well for some applications. However, these techniques are not applicable in the general case, due to language-specific limitations, such as the pointer aliasing problem in C/C++. So, in order to be able to make best use of the various components present in today's heterogeneous systems, suitable programming models for the heterogeneous age of computing are needed.

3.1 REQUIREMENTS OF ADVANCED PROGRAMMING MODELS

The tasks that an advanced programming model for the heterogeneous computing age must be able to fulfill are directly related to the common accelerator architecture motifs that have been identified in Section 2.2.2.

The most important task is of course to capture the parallelism of an application. This can either happen through explicit notation of parallel constructs directly in the source-code, e.g., annotations of a parallel loop, or by capturing the semantics of a program on a very high-level of abstraction, where parallelism can then be found in the individual units of the program.

The second important task is data management. This includes the control over the local and optimized memories on the accelerator itself as well as the data *movements* necessary to make data available to processing units in a system with separate memories for each component (cf. Section 2.2.3). One option is to expose appropriate abstractions and interfaces to the programmer, as it typically is the case for GPU programming models such as CUDA and the memory hierarchy on GPUs. Another alternative is to make data movement and memory hierarchy transparent to the programmer by obtaining the information on which data is needed when on a high level of abstraction and automatically insert necessary data movement operations.

Capturing program semantics on this high level of abstraction, with comparably coarse-grained operations as basic building blocks of an application has another advantage: As discussed in Section 2.2.2, accelerators can significantly reduce the overhead of their hardware implementation and runtime execution by executing larger operations, such as a matrix multiplication, as a single instruction. Such high-level operators in the programming framework do not only have a natural mapping to the accelerator, but their use also significantly raises the abstraction level and therefore benefits programmers. Another example for a programming technique, which directly corresponds to reduced control overhead in hardware, is dataflow programming. This programming paradigm exposes parallelism by representing an application as a graph of communicating actors and is a natural fit for accelerators with multiple units communicating through streams.

Also for the last motif, namely data specialization, appropriate programming abstractions can be integrated into a heterogeneous programming framework. One example for such a mechanism is the support for arbitrary bit-width integer and fixed-point arithmetic types found in High-Level Synthesis tools for FPGAs, where reconfigurability on the bit-level allows to directly reflect such arbitrary bit-width types in hardware.

3.2 GENERAL HETEROGENEOUS PROGRAMMING FRAMEWORKS

For the realization of a programming framework which is able to fulfill the tasks mentioned above, there are various implementation alternatives. One approach are *general* heterogeneous programming frameworks, such as OpenMP, OpenCL, CUDA, or SYCL. These models are not specialized for a single domain or class of applications, but rather define abstraction mechanisms and language extensions for common computational patterns, such as parallel loops or concurrently executing tasks. Typically, they are tightly integrated with an existing, serial programming language, such as C, C++ or Fortran, and therefore allow at least partial re-use of an existing, serial code base (cf. Chapter 4).

The advantage of the generic nature of these programming models is that they can be used to accelerate a wide range of applications from very different domains, as they focus on fundamental computational patterns rather than domain-specific abstractions. Improvements of the implementation of a programming model, e.g., in the compiler or runtime library, benefit many applications, and hardware vendors only need to implement the abstractions of the programming model once to open their hardware up to many users and applications.

On the other hand, due to their general design, such programming frameworks fail to capture the domain-specific, high-level semantics of applications, and still operate on a comparably low level of abstraction, e.g., reasoning about individual loops. Data management is typically also explicit and requires careful attention by the programmer. Because the language mechanisms (e.g., pragmas) are directly integrated into program code, applications implemented in such general frameworks also mix *what* is computed with *how* it is computed, with negative effects on code readability and maintainability.

As the abstractions of general programming frameworks still operate on a comparably low level, implementations using such a framework often are specific to a single class of devices. One example for this problem can be found when comparing vendor recommendations for programming FPGAs or GPUs using OpenCL: While FPGA vendors typically recommend a *single work-item* implementation, with FPGA-specific optimizations such as pipelining applied *inside* that single work-item, GPU vendors usually recommend parallelizing an application across *many work-items* and, if applicable, also concurrently active kernels. As a consequence, OpenCL implementations optimized for FPGAs usually do not deliver good performance on GPUs and vice versa.

3.3 domain-specific programming frameworks

Domain-specific programming frameworks can be used to overcome these limitations of general programming models. They are specialized for a single domain of applications, and provide programming language abstractions for operations specific to this domain. Because these domain-specific operations typically have clearly defined semantics and the memory access pattern is known beforehand, it is possible to separate *what* is computed from *how* it is computed, by moving the mapping of domain-specific operations to accelerator facilities into the implementation of the programming framework. This also allows to vary the mapping depending on available hardware and the desired performance independently of the actual input code.

Domain-specific programming frameworks are also a means to provide programming abstractions that feel "natural" to domain-experts, i.e., the abstractions correspond directly to concepts from the domain.

Standalone domain-specific languages with a custom syntax, such as Halide [52] for image processing, are one possible incarnation of domain-specific programming frameworks, but require additional implementation effort and impose an initial learning overhead on users.

An alternative to this approach are *embedded* domain-specific programming frameworks, which are integrated into an existing, established programming language in the form of library calls or using other language mechanisms provided by the host language. Examples for this approach are popular machine learning frameworks such as Tensorflow [1] or PyTorch [48].

The downside of domain-specific frameworks is the additional implementation effort that initially has to be spent to define and implement the framework itself. For general frameworks, the implementation effort can be amortized over many different applications and domains that can use the framework. For domain-specific frameworks, on the other hand, this effort can be justified only by a large number of users or applications, or significant gains in expressive power or performance. To reduce the initial implementation effort, frameworks such as LLVM [37] or MLIR [38] can facilitate the implementation of domain-specific frameworks (cf. Chapter 6)

3.4 THE ROLE OF COMPILERS

»Designing appropriate hardware for a five-year window in a field that is changing as rapidly as ML is quite challenging. Compilation techniques, possibly themselves enhanced by ML, will be central to meeting this challenge.« — Jeff Dean, David Patterson and Cliff Young, 2018 in [10]

Compilers play a central role in the implementation of both kinds of programming frameworks, as they are responsible for mapping the application, including parallelism, to the target hardware.

Only if the compiler implementation and target mappings make the best possible use of the computational resources available in the heterogeneous system, can good performance for an application be achieved eventually.

Automating the mapping process in the compiler has two important advantages: First, it allows programming frameworks to abstract away many low-level details of the underlying hardware and, as such, makes it easier for users without intimate knowledge of hardware details to use heterogeneous systems and achieve good performance.

Secondly, it makes applications portable. Instead of having a platform-specific implementation for each potential target platform, which requires rewriting the application for each new platform, a single implementation can be used to target different systems, even such systems that are not yet available at the time an application is implemented.

To enable this portability, and even more importantly, *performance portability*, i.e., achieving good performance when porting an application across different heterogeneous systems, significant effort is spent today on designing appropriate programming abstractions that allow to provide hints on parallelism to the compiler, and on the compiler implementation itself, as we will discuss in the following chapters.

4 SELECTING THE RIGHT PROGRAMMING FRAMEWORK

In recent years, the trend towards heterogeneous systems has gained momentum, and heterogeneous systems are pervading ever more industries and areas of life. As a consequence, more and more industries will be confronted with the challenge of choosing an appropriate programming framework for their respective applications and systems.

As heterogeneous computing has become ubiquitous in some industries already, in particular the High-Performance Computing area, many well established heterogeneous, parallel programming frameworks are available and need to be taken into consideration. This raises the question of how one can select the right programming framework for their application and system, and which characteristics and aspects of programming frameworks should be considered as part of the selection process.

One example of an industry where heterogeneous systems are increasingly gaining traction is the automotive industry. This trend is mainly driven by two developments: First, while automotive vendors in the past used multiple, but each simple so-called electronic control units, they are now turning towards *centralized* domain controllers, where a single control unit handles multiple tasks, and, as a consequence, also needs to provide more computational power.

In modern upper-class saloon cars, there can be more than one hundred ECUs [13].

The second main push for the deployment of heterogeneous systems in the automotive industry is the advent of Advanced Driver Assistance Systems (ADAS) that can take over driving tasks from a human driver. The algorithms used in such systems are fairly complex and require massive amounts of computational power, e.g., for object detection or sensor fusion, which can only be provided by heterogeneous systems.

Using the automotive industry as an example, we will demonstrate which aspects can play a role when selecting an appropriate programming framework for heterogeneous systems, and how information about these aspects can be obtained to support the decision process.

Later on, this relationship might reverse: After a code base using a particular programming framework exists, vendors will only employ platforms supporting that framework.

4.1 SOFT ASPECTS OF PROGRAMMING FRAMEWORKS

Of course, there is a number of *hard* factors that can render the use of a particular programming framework for a specific application or domain completely infeasible. Naturally, an implementation of the respective programming framework must be available for the target platform, otherwise it will not be possible to deploy to that platform.

Besides that, the key abstractions of the programming framework should be a good match to the components of the target platform. For example, choosing a programming framework for distributed processing in a compute cluster (e.g., MPI) to program a system with only a single or very few compute elements, connected by a low bandwidth network, is not a good match, as the language constructs for distributed processing of the programming framework will be of no use on such a platform.

Yet, next to these factors, there is also a number of *soft* aspects that play an important role in the selection process. In the context of the automotive industry, examples for such soft aspects are **programmer productivity**, **maintainability**, or **portability**.

Programmer productivity is directly linked to the usability of a programming framework, but also the performance achieved by using such a framework. However, oftentimes the best programming framework is not necessarily the framework that eventually achieves the *highest* performance, but rather the framework that, through suitable abstractions and concise language features, allows programmers to achieve *sufficient* performance. As developer time is an expensive and, nowadays, also scarce resource, the productivity that can be achieved is an important aspect.

Maintainability is another particularly important aspect, especially in the automotive industry, where the lifetime of code, deployed in vehicles and also within the company, can extend over multiple years or even decades. The use of heterogeneous programming frameworks can have a significant impact on the maintainability of code, so this aspect needs to be taken into consideration.

As the deployed components can vary from vehicle to vehicle in a product line, portability is another important aspect to programming frameworks. Once written, the code base should be portable across different target platforms, which requires that the programming framework provides suitable mechanisms to abstract hardware details of the underlying platform.

4.2 STUDY DESIGN

Some of the aspects discussed above, in particular the soft aspects, cannot be assessed by solely investigating the end-product of the development process. Therefore, an implementation study seems to be the most appropriate approach for our effort to investigate the suitability of heterogeneous programming frameworks for the automotive domain, as such a study allows assessing aspects such as programmer productivity already *during* the development process.

The idea of the implementation study is to reflect a real-life development process, in which an existing, serial code base is ported to a modern, heterogeneous embedded system as close as possible. In contrast to other studies [25, 26], which assigned programming tasks to undergraduate students as part of class-room studies to investigate the ease-of-use of programming models, the developers in our study are experienced users of the respective programming framework. We argue that this is a closer resemblance of the real-world scenario, as in industry, such tasks would be assigned to developers which have previously used the programming framework.

Suitable serial code for typical automotive workloads from the ADAS domain was extracted from the open-source Autoware framework [33, 34] into a total of three standalone benchmark kernels, which serve as the starting point for the implementation study. More details on the extraction process and benchmark characteristics can be found in the publication in Chapter 8.

The extracted benchmark kernels are available on GitHub [57] under an open-source license.

From the wide variety of established parallel and heterogeneous programming frameworks from the HPC domain, we have selected OpenMP, OpenCL and CUDA as the most promising candidates, based on the hard factors described above.

After selection of the three different frameworks, an experienced developer for each of the frameworks was tasked with using the framework to parallelize the existing serial code base of the three benchmark kernels with the aim to offload computations to the accelerators on the heterogeneous system. During this process, some key indicators were constantly tracked, e.g., the number of working hours spent on the kernel and the speedup over the original serial code base.

To assess portability, each developer would then port each kernel to at least one other embedded, heterogeneous platform.

Even more details can be found in the extensive technical report [58]

The indicators collected during development were then complemented by an analysis of the resulting code to assess the suitability of the different programming models for the automotive domain. The publication in Chapter 9 describes the tracking process and the metrics we developed in more detail and provides detailed insights.

4.3 KEY FINDINGS

While the detailed insights regarding the individual programming frameworks can be found in Chapter 9, we can also derive a number of general properties that proved to be beneficial for programming frameworks throughout the study.

One key finding is that a programming framework that requires less restructuring of an existing serial code base clearly enhances programmer productivity and allows developers to achieve higher levels of performance faster. This is not only due to the lower effort required, but also because any restructuring of the code always bears the danger of introducing errors into the code that then require tedious debugging efforts to fix. Also, if a strategy for parallelization and offloading to accelerators can be implemented quicker, it can also be assessed very early in the development process, to see whether it is promising or not. This avoids the pitfall of wasting many working hours on a strategy that turns out to not provide any performance benefits.

A second key finding was that programming language constructs operating on a higher level of abstractions can be beneficial, and in particular can facilitate portability across platforms.

One programming framework that provides such abstractions is OpenMP and therefore allows for fast parallelization, offloading, and portability. For the implementation of these constructs, OpenMP relies on the compiler, which can also be a disadvantage if such compiler support is not available for a platform.

During the study, this was the case for the chosen target platforms. As no compiler supported the combination of ARM CPU and Nvidia GPU, it was not possible to use OpenMP device offloading features to target accelerators such as GPUs.

Later on, support for this combination was added in LLVM by other open-source contributors. In the publication in Chapter 10, we further extended this device offloading support, in particular the runtime plugin for CUDA GPUs, to *embedded* heterogeneous devices, mainly targeting the Nvidia Jetson family.

The evaluation in Chapter 10 showed that the extended implementation is able to provide performance competitive with a native CUDA implementation on embedded heterogeneous systems. Considering that our study showed that the use of OpenMP for offloading often requires *less* developer effort than a CUDA implementation, this makes OpenMP an interesting candidate for the programming of embedded heterogeneous systems.

5 OPENMP FOR FPGAS

As discussed in the previous chapter, OpenMP combines many positive characteristics of programming frameworks. While earlier versions of OpenMP have exclusively focused on multi-threaded execution on CPUs, version 4.0 of the OpenMP standard also introduced constructs for device offloading, making OpenMP an attractive candidate for heterogeneous systems programming.

5.1 DEVICE OFFLOADING FOR FPGAS

A simple example for OpenMP device offloading is shown in 5.1. The `target` directive allows to easily denote regions of code that should be executed on a *device*, i.e., an accelerator in a heterogeneous system. The first implementations of OpenMP device offloading targeted devices such as GPUs [5, 6] and DSPs [40]. Yet, the ability to specify application code for host *and* device in the same source file in an intuitive manner makes OpenMP offloading also very interesting for FPGA programming.

The currently predominant way to target FPGAs in a heterogeneous system via High-Level Synthesis is by using OpenCL. However, as discussed in Chapter 4 and the publication in Chapter 9, restructuring an application for OpenCL takes significant effort due to the verbose host API and also requires the separation of device code and host code in different compilation units.

This raises the question of whether it is possible to use OpenMP device offloading to target FPGAs. OpenMP not only allows to denote device regions within host code, but the `map` clauses, as showcased in Fig. 5.1, also provide a mechanism to specify the necessary data transfers between host and device with low overhead and in an intuitive manner.

In order to investigate this question, our publication in Chapter 11 presented the *first-ever* prototype of OpenMP offloading to FPGAs based on the LLVM [37] infrastructure. The compilation flow based on Clang allowed targeting FPGAs using OpenMP device offloading. The target regions of the code were extracted and passed on to the proprietary HLS tool Vivado HLS by Xilinx.

Through a plugin developed for this work, which integrates with LLVM's OpenMP runtime for device offloading (`libomptarget`), data transfers using the intuitive `map` clauses and execution handling were provided at runtime. This runtime plugin as well as the compilation

```
float a[N];
float b[N];
float c[N];

# pragma omp target \
  teams distribute parallel for\
  map(to:a[0:N], b[0:N]) map(from:c[0:N])
for(int i=0; i < N; ++i){
  c[i] = a[i] + b[i];
}
```

Figure 5.1: Simple example for OpenMP device offloading of a vector addition.

flow we developed relied on the open-source framework TaPaSCo [20] for automation of the HLS flow and the software/hardware interface.

At the time the work in Chapter 11 was published, TaPaSCo was still called ThreadPoolComposer (TPC).

While this first prototype already successfully demonstrated that OpenMP device offloading allows offloading regions of code to the FPGA and manage data-transfers, it still suffered from limitations, as parallel OpenMP constructs inside the target region were not yet supported and the interaction with Vivado HLS had to rely on source code.

5.2 PARALLEL CONSTRUCTS ON FPGAS

Yet, not only can the device offloading itself be interesting as input code for targeting FPGAs, in addition the concurrent constructs in OpenMP can be used for HLS. As the example in Fig. 5.1 shows, device offloading can be combined with worksharing constructs, dividing work across teams and, subsequently, threads in the team, in this example.

In prior work, either concurrent tasks [14, 15, 50] or worksharing loops [8] had been used to initiate parallel execution on the FPGA through HLS. Still, in both cases, the authors solely relied on *spatial* parallelism, i.e., they used the FPGA's hardware resources to instantiate concurrently executing accelerators for tasks or threads, but inside each accelerator, only a single thread of execution would be active.

However, inside an FPGA accelerator generated by HLS, most operators are active at all times. Usually, the execution is organized into so-called *stages* by a static schedule, and only one stage is active at a time, leaving the remaining operators idle.

Huthmann et al. [28] have demonstrated that by operating multiple threads in different stages of the *same* hardware accelerator, idle cycles and execution time can be reduced significantly. Therefore, in the publication in Chapter 12, we developed a multi-threaded execution model for High-Level Synthesis from OpenMP worksharing loops.

The execution model allows the accelerators to keep track of multiple threads, working together in an OpenMP worksharing loop. While still only a single thread is active at a time, pausing this thread, e.g., due to an access to external memory, does not cause the hardware to become idle. Instead, another thread is activated and resumes execution. In the background, the suspended thread's memory access will be completed, making the thread available for execution again.

This execution model can also be combined with the approach from prior work, so that there are multiple instances of the hardware accelerator, each executing multiple threads. As the evaluation in Chapter 12 shows, this combination can reduce idle cycles in hardware by a factor of up to 8x and improve execution time by a factor of up to 3x.

5.3 OPENMP OFFLOADING FOR MULTI-THREADED FPGA ACCELERATORS

Given the potential of multi-threaded execution on FPGAs and the benefits of the low-overhead, intuitive interface of device offloading to FPGAs, the publication in Chapter 13 combines the two approaches, yielding an easy-to-use interface for programmers for creating multi-threaded FPGA accelerators from OpenMP concurrent constructs.

The general outline of the compilation flow and integration with LLVM's OpenMP offloading runtime are similar to what was described in Section 5.1 and Chapter 11, but with one major difference: Instead of passing the content of the target region to the HLS compiler in the form of C/C++ code, which severely limits the expressiveness of the input to the HLS flow, the new compilation flow uses LLVM IR as input to the HLS compiler Nymble [29].

Using LLVM IR allows communicating information about concurrent OpenMP constructs such as `teams distribute` or `parallel for` to the HLS compiler for the generation of multi-threaded hardware accelerators. In addition, it also enables transformations on the intermediate representation to happen before the HLS compilation. These transformations make it possible to support interesting extensions, such as vectorized loads/stores, access to FPGA on-chip memory and use of OpenMP synchronization constructs (`omp critical`, `omp barrier`), inside the target region intended for offloading to the FPGA. The flow also supports the use of an optimized memory preloader through a dedicated function, similar to other OpenMP API functions.

The multi-threaded execution model used by the generated hardware accelerators is an advancement of the model presented by Huthmann et al. [28]. While the model discussed in Section 5.2 and Chapter 12 is designed to have a small hardware overhead, the execution model used in Chapter 13 trades some hardware resources for even higher performance by allowing multiple threads to be active at the same time in different stages of the generated hardware accelerator. This further reduces idle cycles and improves performance.

As detailed insights into the performance of an accelerator and potential bottlenecks are an invaluable tool for HPC developers, we have complemented the compilation flow by an approach for detailed performance tracking in HLS-generated accelerators, presented in the publication in Chapter 14.

By extending the compilation flow to insert small counters that keep track of important performance metrics such as floating-point performance, memory bandwidth and thread execution state, the approach allows to gain meaningful insight into the execution characteristics of an HLS-generated accelerator, as demonstrated in Chapter 14. To present the gathered performance information to the user, the generated profiles are compatible with the popular Paraver performance visualizer [49].

6 ACCELERATED SUM-PRODUCT NETWORK INFERENCE

While the previous two chapters focussed on general programming frameworks and OpenMP in particular, this chapter will investigate the use of *domain-specific* programming frameworks for heterogeneous systems. As previously discussed in Chapter 3, domain-specific programming frameworks provide a number of advantages that make them an attractive candidate for heterogeneous systems programming.

To demonstrate how a domain-specific programming framework can enable the usage of a variety of target platforms from a single input description and provide significant performance improvements, we are going to use the domain of Probabilistic Graphical Models (PGMs), more specifically, Sum-Product Network (SPN) [51] as an example of such a domain.

6.1 SUM-PRODUCT NETWORK BACKGROUND

Probabilistic (graphical) models such as Sum-Product Networks, are nowadays receiving increasing attention as a robust alternative and complement to other techniques from the Machine Learning domain, such as (deep) neural networks. Due to their true probabilistic interpretation, probabilistic models can much better handle the uncertainty found in many real-world applications and are also able to express uncertainty over their output, for example when confronted with out-of-domain samples, a property especially important in safety-critical applications.

While these interesting properties are common to most approaches from the domain of probabilistic models, Sum-Product Networks have an important advantage over many of these approaches: Given that the SPN was constructed according to a number of simple rules (that can be enforced in the learning process), SPNs guarantee *tractable* inference, i.e., the complexity of inference is linear with respect to the model size.

This property allows to deploy Sum-Product Networks in a wide variety of applications, from databases [24] and robotics [60] to medical imaging [53] or signal processing, e.g., robust automatic identification of speakers [46]. An overview of applications using Sum-Product Networks can be found in the survey by Sánchez-Cauce et al. [55].

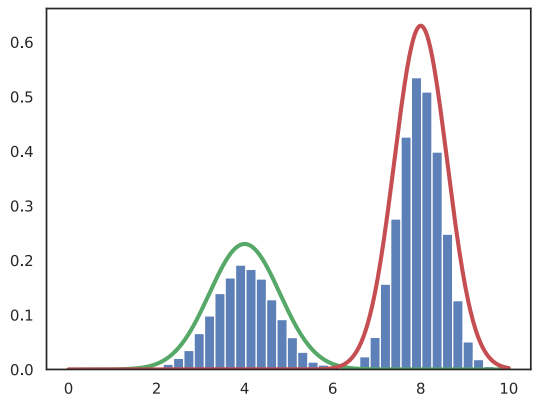


Figure 6.1: Example for the mixture of the two probability distributions, shown by the red and green curve. The mixture is used to precisely capture the underlying distribution shown by the blue bars.

6.1.1 *Model Structure*

A Sum-Product Network model captures the joint probability distribution over a set of variables in the form of a Directed Acyclic Graph (DAG). The graph consists of three different types of nodes, each with particular semantics:

- The leaf nodes of the graph capture the univariate probability distribution over a single variable, e.g., a Gaussian distribution. This distribution can directly be obtained from the observations (i.e., the training data) or a subset thereof. For example, for a discrete, categorical variable, one could simply count the number of occurrences of each category to assess the distribution. The *scope* of a leaf node is the single variable associated with it.

- Product nodes correspond to a factorization of independent variables. Simply put, two variables are independent of each other, if the realization (i.e., the value) of one does not have an impact on the probability distribution of the other. In such cases, the two variables can safely be considered separately.

  For the SPN to be valid and have guaranteed tractable inference, the scopes of the different child nodes of a product node must be disjoint. The scope of the product node itself is the union of the scopes of all child nodes.

- Weighted sum nodes correspond to a mixture of multiple distributions and assign a weight to each of the child nodes. A simple example for the mixture of distributions can be found in Fig. 6.1. The underlying distribution (blue bars) cannot be captured by a single distribution. Through mixture (i.e. by combining) two distributions (shown as red and green curve), the underlying distribution can be captured precisely. The different weights associated with the two child distributions result from the fact that the red curve covers more instances than the green.

In a valid SPN, the scope of all children of a sum node must be identical, and this scope will also be the scope of the sum node itself.

As the name indicates, leaf nodes can only occur at the leaves of the DAG and cannot have any child nodes. Every SPN also has a single root node, and multiple layers of sum- and product nodes can occur between root nodes and leaves.

6.1.2 *Learning*

While the graph structure of a Sum-Product Network, as described in the previous section, can be hand-crafted, optionally followed by weight-learning only, there are also a number of algorithms to automatically learn the DAG *structure* of the SPN model from data.

To illustrate how this learning process works, we give a brief overview of one such learning algorithm as presented by Molina et al. [41]. The base case of the algorithm is reached when only a single variable is left, in which case the algorithm will simply learn a leaf node from the univariate distribution of that variable.

If more than one variable is left, the algorithm will first try to detect independencies among these variables through a pair-wise independence test. In case the algorithm is able to find independent subsets of variables, it will create a product node and recurse on each of the subsets to obtain a child node for each of the subsets.

If the algorithm is not able to find any independent subsets, it will instead perform a partitioning of the training samples and create a sum node. Afterwards, it recurses on each of the partitions to obtain a child node for that partition. The weights associated with each child node directly correspond to the overall fraction of samples assigned to this cluster.

The simple example of the age and income distribution of the members of an university in Fig. 6.2 can be used to further illustrate the learning process.

In the first step, the algorithm would fail to detect an independence of the variables *Age* and *Income* and would therefore perform a clustering, yielding the four colored clusters in Fig. 6.2. The algorithm would then recurse on each of these clusters, where age and income are now independent, so a product node would be created for each of the clusters. In the last step, the algorithm would learn the univariate distributions for age and income, respectively, in each of the clusters.

The learning process would then yield the final Sum-Product Network model depicted in Fig. 6.3, which also illustrates the assignment

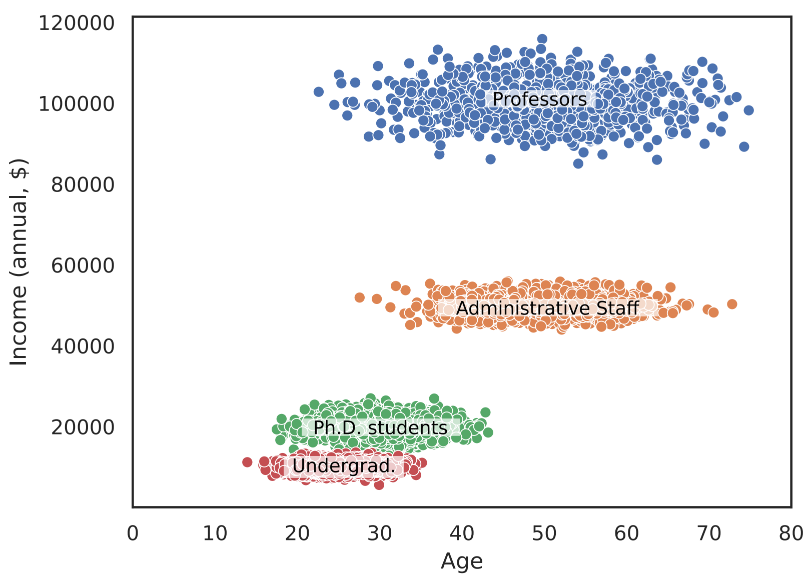


Figure 6.2: Example distribution of age and income among university staff.

of weights to child nodes of a sum node: As there are typically far fewer professors than students at a university, the blue cluster contains fewer instances, and the weight (label (a)) associated with the left-most child node of the root sum is therefore comparably small.

The algorithm for which a brief overview was provided here is only one possible way to learn the structure of a Sum-Product Network from data – an overview of other learning approaches can be found in [55].

6.1.3 *Inference*

After the SPN model structure has been obtained using one of the approaches just described, it can be used to answer probabilistic queries through inference. Examples of probabilistic queries for our example would be questions such as how likely it is that an employee of a university receives an income of I_1 at age A_1 ($\mathcal{P}(\text{Age} = A_1, \text{Income} = I_1)$, joint probability) or the probability of earning I_2 at any age ($\mathcal{P}(\text{Income} = I_2)$, marginal probability).

There are multiple kinds of inference, depending on the kind of query that should be answered, but all involve a bottom-up traversal of the SPN DAG, starting at the leaf nodes.

For example, to obtain a joint probability given some evidence, i.e., values for the variables the SPN is defined over, the inference process will first query the univariate distributions at the leaf nodes using the evidence to obtain a probability value. This probability value is then propagated towards the root node of the SPN, performing the respective operations on the way, i.e., multiplication of probabilities at each product node and weighted addition at each sum node.

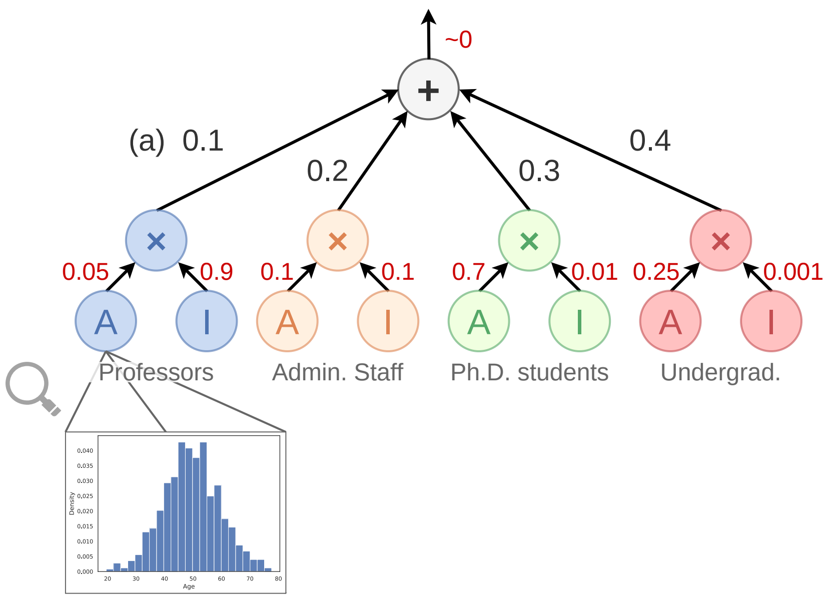


Figure 6.3: Example of an SPN model for the distribution in Fig. 6.2. Weights are annotated in black, probabilities for the example query in red. For each of the four groups of persons at the university, the algorithm has learned a sub-graph, attached as child node to the root sum node. Within each group, age and income are independent of each other, implying a product node. The distributions for age and income in each group can be obtained from the data, e.g., by counting the number of professors of each age, as shown in the magnification.

To illustrate this process, let us return to the example SPN in Fig. 6.3: Assume that the SPN should be used to answer how likely it is to earn $100,000 per year at the age of 29 when employed by an university, i.e., $\mathcal{P}(\text{Age} = 29, \text{Income} = \$100k)$.

The evidence in this case is $\text{Age} = 29$ and $\text{Income} = \$100k$, and is used to query the univariate leaf distributions. In case of the professors on the left-most side of the SPN, a low probability for the age of 29 will be obtained, as there are few professors aged 29, but a high probability of earning $100k is yielded. These two probabilities get multiplied, and the same happens for all other product nodes. The resulting probabilities are then multiplied by the associated weights and summed up to eventually obtain the final result. Unfortunately for the author of this thesis, the obtained probability is very close to zero.

Inference is the most important process when a Sum-Product Network model is actually deployed into an application, therefore the remainder of this chapter will investigate how to accelerate SPN inference on heterogeneous systems. Learning of the model structure is assumed to have taken place beforehand, e.g., offline on a large machine.

6.2 HARDWARE ACCELERATION

In previous work, various accelerator architectures have demonstrated that FPGAs are a very promising platform for the implementation of accelerators for machine learning inference, not only for (deep) neural networks [18], but also other ML approaches such as Random Forest [39]. Given that, and considering the fact that the DAG structure of Sum-Product Network models lends itself to pipelining, developing an FPGA-based accelerator for SPN inference seems promising.

Therefore, in the publication in Chapter 15, we developed an automatic toolflow as a turn-key solution to accelerated SPN inference on FPGAs. Starting from an input description of the SPN model, obtained through the open-source library SPFlow [42], the toolflow will generate a fully pipelined accelerator for SPN inference, including all the necessary interface components to embed the accelerator in a heterogeneous system and to load/store input and result data to/from device memory.

The generated accelerator uses two of the motifs discussed in Section 2.2.2: The pipelined, throughput-oriented architecture allows to overlap the execution of the inference for multiple samples inside the accelerator, a form of *parallel execution*. In addition, by using a static schedule without explicit synchronization between the arithmetic hardware operators in the datapath at runtime, and by specializing the load/store infrastructure for the access pattern of SPN inference, the accelerator also incurs a significantly *reduced overhead*. The combination of these two motifs allows the accelerator to outperform existing CPU and GPU implementations for many benchmarks.

In the publication in Chapter 16, we further extended the parallel processing to concurrent execution on multiple accelerator instances in a cloud-deployed FPGA, concretely the Amazon AWS EC2 F1 FPGA instances. The software/hardware interface was extended to not only overlap pipelined execution on multiple accelerator instances, but also to overlap data-transfers between host and FPGA with execution. This yields speedups of up to 1.6x over a 12-core Xeon CPU and up to 6.6x over a Nvidia V100 GPU.

Yet, there is a third motif left that can be employed for Sum-Product Network inference, namely *data specialization*. In the case of SPNs, that means that the hardware arithmetic operators inside the datapath can be specialized for the cases and ranges relevant for SPN inference.

So far, the accelerator was using standard IEEE-754 double arithmetic operators from the open-source FloPoCo [12] library. In the publication in Chapter 17, we therefore compared three different arithmetic formats with regard to their suitability for SPN inference. Next to a customized and optimized implementation of floating-point operators, the Posit format [19], and a format based on the logarithmic number system and specialized for SPN inference were taken into consideration.

As the FPGA provides the the necessary flexibility to realize arbitrary configurations of the arithmetic formats, a fast, software-based Design-Space Exploration (DSE) was used to determine the best configuration for each format and benchmark.

The evaluation in Chapter 17 shows that the use of specialized arithmetic formats does not only enable significant reductions in hardware resource usage, but also improves performance through the reduction of pipeline stages in the individual operators and reduces energy consumption.

6.3 ACCELERATION ON CPUS AND GPUS

Yet, not only FPGAs can provide fast Sum-Product Network inference, but there also previously unused potentials for accelerated inference on CPUs and GPUs. In particular, the Python-based inference in SPFlow [42] leaves a lot of room for improvements, as Python is designed for usability rather than for performance [21].

One approach to lift these potentials is the translation to a Tensorflow [1] graph for execution. However, as the evaluation in Chapter 15 shows, Tensorflow's abstractions are not ideally suited for Sum-Product Networks and inference on the translated SPN graph still suffers from performance issues.

Another approach to accelerate inference, which we developed and that was used a software baseline in Chapter 15, Chapter 16 and Chapter 17, is to automatically generate code from the SPN description, more specifically C++ code with OpenMP directives for CPUs and CUDA code for Nvidia GPUs. However, as already discussed in Section 5.1, the expressiveness of source code can be a limitation and not all transformations can be performed at this level.

Therefore, in the publication in Chapter 18, we developed a domain-specific compiler for Sum-Product Network inference based on the MLIR [38] and LLVM [37] frameworks. Again starting from an SPN description obtained from the SPFlow framework [42], the compiler, through a series of MLIR dialects, lowerings and transformations, automatically generates optimized executables for CPUs and CUDA GPUs, which are then loaded to execute inference on the SPN.

The MLIR based compiler for Sum-Product Networks is available on GitHub [59] under an open-source license.

Similar to the FPGA accelerator, where the parallelism in the application was leveraged for pipelining, this parallelism can also be exploited on the two target platforms. On the CPU, through multi-threading and usage of SIMD vector extensions, the compiler is able to achieve speedups of up to 814x over SPFlow. When compiling for the GPU, a speedup of up to 524x is achieved through the massively parallel SIMT execution model.

These speedups are also much higher than the speedups of 1.5x and 1.4x achieved through the translation to a Tensorflow graph.

7 CONCLUSION AND OUTLOOK

As CPU performance has been stagnating for most of the past decade due to the developments and limitations discussed in Chapter 2, the future of computing is heterogeneous. This era of heterogeneous computing does not only promise significant improvements in performance and performance-per-watt, and lots of exciting new architectures, but also comes with a number of challenges when it comes to programming these heterogeneous systems.

Throughout the course of this thesis, we have explored two potential answers to these programming challenges.

One potential solution are general programming frameworks that have been established in the HPC domain, such as OpenMP, and which have evolved over time to better adapt to for the programming challenges of the heterogeneous computing era. The selection of the right programming framework for a specific task, domain or platform is a crucial step. In Chapter 4 and the corresponding publications in Part ii, we have identified important aspects and characteristics of programming frameworks to guide such a decision.

Afterwards, in Chapter 5 and Part iii, we have demonstrated how to extend the implementation and compiler support for one such programming framework, namely OpenMP, to FPGAs as a new class of devices. Our implementation of OpenMP device offloading for FPGAs does not only provide an easy-to-use interface for targeting FPGAs in an application and manage data exchange between host and device with low overhead, but also significantly improves the utilization of the accelerator through hardware multi-threading.

Another interesting potential solution to the programming challenge are domain-specific programming frameworks. Using the example of Sum-Product Networks, a class of machine learning models, we have demonstrated in Chapter 6 and Part iv how to design and implement a domain-specific framework targeting not only GPUs and FPGAs, but also leveraging all available hardware features of the CPU. By capturing the high-level semantics and structure of the application, and using this information for the mapping to the target hardware, our framework achieves significant performance improvements for Sum-Product Network inference. At the same time, the framework's interface provides abstractions familiar to SPN domain experts.

In the foreseeable future, the trend towards heterogeneous systems will continue, with new accelerator architectures and completely new kinds of accelerators being deployed into systems. At the same time, the existing accelerators will also be used for other kinds of applications, as it is already happening with accelerators designed for machine learning training and inference, such as Cerebras' Wafer Scale Engine, being used in general scientific computing [54].

Future generations of systems will also couple the accelerators more tightly with the host CPU. Cache-coherent interfacing protocols such as CCIX or CXL are precursors of this trend, and devices integrating the CPU and accelerators in the same SoC or package, such as Intel's HARP, might also become more common in non-embedded scenarios.

In the area of general programming frameworks, the SYCL programming framework is a comparably new, yet promising candidate. It combines many of the important characteristics of programming frameworks discussed in Chapter 4, e.g., a tight coupling with the host programming language (C++), and the ability to target a wide variety of processing elements, including CPUs, GPUs, and FPGAs. Despite being around for only a few years, it has already seen significant vendor adoption, with Intel picking it up and extending it as DPC++ in their oneAPI effort.

With regard to domain-specific programming, frameworks such as MLIR [38], which provide abstractions and infrastructure for the implementation of domain-specific compilers, will be a key enabler for the development of domain-specific programming frameworks that successfully capture the high-level semantics of applications and efficiently map it to target platforms, all while providing an easy-to-use interface for domain experts.

BIBLIOGRAPHY

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensorflow: A system for large-scale machine learning." In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[2] *AI and Compute*. OpenAI. May 16, 2018. URL: `https://openai.com/blog/ai-and-compute/` (visited on 05/10/2021).

[3] Gustavo Alonso, Timothy Roscoe, David Cock, Muhsen Owaida, Kaan Kara, Dario Korolija, Zeke Wang, et al. "Tackling Hardware/Software co-design from a database perspective." In: *Proceedings of the 6th biennial Conference on Innovative Data Systems Research (CIDR), Amsterdam, Netherlands, January 2020.* 2020.

[4] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: `10.1145/1465482.1465560`. URL: `https://doi.org/10.1145/1465482.1465560`.

[5] Samuel F Antao, Alexey Bataev, Arpith C Jacob, Gheorghe-Teodor Bercea, Alexandre E Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, et al. "Offloading support for OpenMP in Clang and LLVM." In: *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE. 2016, pp. 1–11.

[6] Carlo Bertolli, Samuel F Antao, Gheorghe-Teodor Bercea, Arpith C Jacob, Alexandre E Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos, David Appelhans, et al. "Integrating GPU support for OpenMP offloading directives into Clang." In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–11.

[7] Jeronimo Castrillon et al. "A Hardware/Software Stack for Heterogeneous Systems." In: *IEEE Transactions on Multi-Scale Computing Systems* 4.3 (July 2018). Conference Name: IEEE Transactions on Multi-Scale Computing Systems, pp. 243–259. ISSN: 2332-7766. DOI: `10.1109/TMSCS.2017.2771750`.

[8] Jongsok Choi, Stephen Brown, and Jason Anderson. "From software threads to parallel hardware in high-level synthesis for FPGAs." In: *2013 International Conference on Field-Programmable Technology (FPT)*. IEEE. 2013, pp. 270–277.

[9] William J. Dally, Yatish Turakhia, and Song Han. "Domain-specific hardware accelerators." In: *Communications of the ACM* 63.7 (June 18, 2020), pp. 48–57. ISSN: 0001-0782. DOI: 10.1145/3361682. URL: https://doi.org/10.1145/3361682 (visited on 05/10/2021).

[10] Jeff Dean, David Patterson, and Cliff Young. "A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution." In: *IEEE Micro* 38.2 (Mar. 2018). Conference Name: IEEE Micro, pp. 21–29. ISSN: 1937-4143. DOI: 10.1109/MM.2018.112130030.

[11] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions." In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[12] Florent de Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo." In: *IEEE Design Test of Computers* 28.4 (2011), pp. 18–27. DOI: 10.1109/MDT.2011.44.

[13] *Elektronik und Software beherrschen Innovationen im Auto*. springer-professional.de. May 6, 2014. URL: https://www.springerprofessional.de/automobilelektronik---software/antriebsstrang/elektronik-und-software-beherrschen-innovationen-im-auto/6561802 (visited on 07/09/2021).

[14] Antonio Filgueras, Eduard Gil, Carlos Alvarez, Daniel Jimenez, Xavier Martorell, Jan Langer, and Juanjo Noguera. "Heterogeneous Tasking on SMP/FPGA SoCs: the Case of OmpSs and the Zynq." In: *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. 2013, pp. 290–291.

[15] Antonio Filgueras, Eduard Gil, Daniel Jimenez-Gonzalez, Carlos Alvarez, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees Vissers. "OmpSs@Zynq all-programmable SoC ecosystem." In: *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. 2014, pp. 137–146.

[16] P Greenhalgh. "big. LITTLE technology: The future of mobile." In: *ARM Limited, White Paper* (2013), p. 12.

[17] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "Polly — Performing Polyhreadal Optimizations on a Low-Level Intermediate Representation." In: *Parallel Processing Letters* 22.04 (2012), p. 1250010. DOI: 10.1142/S0129626412500107. eprint: https://doi.org/10.1142/S0129626412500107. URL: https://doi.org/10.1142/S0129626412500107.

[18] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. “A Survey of FPGA-Based Neural Network Inference Accelerators.” In: *ACM Trans. Reconfigurable Technol. Syst.* 12.1 (Mar. 2019). ISSN: 1936-7406. DOI: 10.1145/3289185. URL: https://doi.org/10.1145/3289185.

[19] John Gustafson and Isaac Yonemoto. “Beating Floating Point at its Own Game: Posit Arithmetic.” In: *Supercomputing Frontiers and Innovations* 4.2 (2017). ISSN: 2313-8734. URL: https://www.superfri.org/superfri/article/view/137.

[20] Carsten Heinz, Jaco Hofmann, Jens Korinth, Lukas Sommer, Lukas Weber, and Andreas Koch. “The TaPaSCo Open-Source Toolflow.” In: *Journal of Signal Processing Systems* (May 2021). ISSN: 1939-8115. DOI: 10.1007/s11265-021-01640-8. URL: https://doi.org/10.1007/s11265-021-01640-8.

[21] John L. Hennessy and David A. Patterson. “A new golden age for computer architecture.” In: *Communications of the ACM* 62.2 (Jan. 28, 2019), pp. 48–60. ISSN: 0001-0782. DOI: 10.1145/3282307. URL: https://doi.org/10.1145/3282307 (visited on 05/04/2021).

[22] *Highlights - June 2011 | TOP500*. URL: https://top500.org/lists/top500/2011/06/highlights/ (visited on 07/15/2021).

[23] *Highlights - June 2021 | TOP500*. URL: https://top500.org/lists/top500/2021/06/highs/ (visited on 07/15/2021).

[24] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. “DeepDB: Learn from Data, not from Queries!” In: *CoRR* abs/1909.00607 (2019). arXiv: 1909.00607. URL: http://arxiv.org/abs/1909.00607.

[25] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz. “Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers.” In: *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. Nov. 2005.

[26] Lorin Hochstein and Victor R. Basili. “An Empirical Study to Compare Two Parallel Programming Models.” In: *ACM SPAA*. Cambridge, Massachusetts, USA, 2006. ISBN: 1-59593-452-9.

[27] Mark Horowitz. “1.1 Computing’s energy problem (and what we can do about it).” In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). ISSN: 2376-8606. Feb. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[28] Jens Huthmann and Andreas Koch. "Optimized High-Level Synthesis of SMT Multi-Threaded Hardware Accelerators." In: *International Conference on Field-Programmable Technology (FPT)*. 2015.

[29] Jens Huthmann, Björn Liebig, Julian Oppermann, and Andreas Koch. "Hardware/software co-compilation with the Nymble system." In: *Proc. of the Intl. Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. 2013.

[30] *IBM Introduces the World's First 2-nm Node Chip - IEEE Spectrum*. en. URL: https://spectrum.ieee.org/nanoclast/semiconductors/nanotechnology/ibm-introduces-the-worlds-first-2nm-node-chip (visited on 07/15/2021).

[31] *IBM Unveils World's First 2 Nanometer Chip Technology, Opening a New Frontier for Semiconductors*. en-us. July 2021. URL: https://newsroom.ibm.com/2021-05-06-IBM-Unveils-Worlds-First-2-Nanometer-Chip-Technology,-Opening-a-New-Frontier-for-Semiconductors (visited on 07/15/2021).

[32] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. New York, NY, USA: Association for Computing Machinery, June 24, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246. URL: https://doi.org/10.1145/3079856.3080246 (visited on 07/05/2021).

[33] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. "An Open Approach to Autonomous Vehicles." In: *IEEE Micro* 35.6 (Nov. 2015), pp. 60–68. ISSN: 0272-1732.

[34] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems." In: *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. ICCPS '18. Porto, Portugal: IEEE Press, 2018, pp. 287–296. ISBN: 978-1-5386-5301-2. DOI: 10.1109/ICCPS.2018.00035. URL: https://doi.org/10.1109/ICCPS.2018.00035.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[36] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction." In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture,*

2003. MICRO-36. 2003, pp. 81–92. DOI: 10.1109/MICRO.2003.1253185.

[37] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0769521029.

[38] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. *MLIR: A Compiler Infrastructure for the End of Moore's Law*. 2020. arXiv: 2002.11054 [cs.PL].

[39] Xiang Lin, R.D. Shawn Blanton, and Donald E. Thomas. "Random Forest Architectures on FPGA for Multiple Applications." In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. GLSVLSI '17. Banff, Alberta, Canada: Association for Computing Machinery, 2017, pp. 415–418. ISBN: 9781450349727. DOI: 10.1145/3060403.3060416. URL: https://doi.org/10.1145/3060403.3060416.

[40] Gaurav Mitra, Eric Stotzer, Ajay Jayaraj, and Alistair P Rendell. "Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture." In: *International Workshop on OpenMP*. Springer. 2014, pp. 202–214.

[41] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Sriraam Natarajan, Floriana Esposito, and Kristian Kersting. "Mixed sum-product networks: A deep architecture for hybrid domains." In: *Thirty-second AAAI Conf. on artificial intelligence*. 2018.

[42] Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. *SPFlow: An Easy and Extensible Library for Deep Probabilistic Learning using Sum-Product Networks*. 2019. arXiv: 1901.03704 [cs.LG].

[43] Gordon E. Moore. *Cramming more components onto integrated circuits*. 1965.

[44] Gordon E. Moore. "Progress in digital integrated electronics." In: *Electron devices meeting*. Vol. 21. Maryland, USA. 1975, pp. 11–13.

[45] Gordon E. Moore. "No exponential is forever: but "Forever" can be delayed! [semiconductor industry]." In: *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.* 2003, 20–23 vol.1. DOI: 10.1109/ISSCC.2003.1234194.

[46] Aaron Nicolson and Kuldip K Paliwal. "Sum-product networks for robust automatic speaker identification." In: *arXiv preprint arXiv:1910.11969* (2019).

[47] Nvidia Inc. *CUDA C++ Programming Guide - Version 11.5*. Oct. 2021.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[49] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. "Paraver: A Tool to Visualize and Analyze Parallel Code." In: *Proceedings of WoTUG-18: transputer and occam developments*. Vol. 44. 1. Citeseer. 1995, pp. 17–31.

[50] Artur Podobas. "Accelerating parallel computations with openmp-driven system-on-chip generation for fpgas." In: *2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*. IEEE. 2014, pp. 149–156.

[51] Hoifung Poon and Pedro Domingos. "Sum-product networks: A new deep architecture." In: *IEEE Int. Conf. on Computer Vision Workshops)*. 2011.

[52] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530.

[53] Fabian Rathke, Mattia Desana, and Christoph Schnörr. "Locally adaptive probabilistic models for global segmentation of pathological OCT scans." In: *Int. Conf. on Medical Image Computing and Computer-Assisted Intervention*. Springer. 2017.

[54] Kamil Rocki, Dirk Van Essendelft, Ilya Sharapov, Robert Schreiber, Michael Morrison, Vladimir Kibardin, Andrey Portnoy, Jean Francois Dietiker, Madhava Syamlal, and Michael James. "Fast Stencil-Code Computation on a Wafer-Scale Processor." In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2020, pp. 1–14. DOI: `10.1109/SC41405.2020.00062`.

[55] Raquel Sánchez-Cauce, Iago París, and Francisco Javier Díez. "Sum-product networks: A survey." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[56] David E. Shaw et al. "Anton 2: Raising the Bar for Performance and Programmability in a Special-Purpose Molecular Dynamics Supercomputer." In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 41–53. DOI: 10.1109/SC.2014.9.

[57] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. *Darmstadt Automotive Parallel HeterogeNEous Benchmark-Suite (DAPHNE)*. https://github.com/esa-tu-darmstadt/daphne-benchmark. 2019.

[58] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. *FAT-Schriftenreihe 317 - EPHoS: Evaluation of Programming-Models for Heterogeneous Systems*. Tech. rep. 2019. URL: https://www.vda.de/de/services/Publikationen/fat-schriftenreihe-317.html.

[59] *SPNC – SPN Compiler*. July 2021. URL: https://github.com/esa-tu-darmstadt/spn-compiler (visited on 08/04/2021).

[60] Kaiyu Zheng, Andrzej Pronobis, and Rajesh P. N. Rao. "Learning Semantic Maps with Topological Spatial Relations Using Graph-Structured Sum-Product Networks." In: *CoRR* abs/1709.08274 (2017). arXiv: 1709.08274. URL: http://arxiv.org/abs/1709.08274.

Part II

PARALLEL PROGRAMMING MODELS FOR EMBEDDED HETEROGENEOUS PLATFORMS

8 WORK-IN-PROGRESS: DAPHNE - AN AUTOMOTIVE BENCHMARK SUITE FOR PARALLEL PROGRAMMING MODELS ON EMBEDDED HETEROGENEOUS PLATFORMS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *Work-in-Progress: DAPHNE - An Automotive Benchmark Suite for Parallel Programming Models on Embedded Heterogeneous Platforms* by *Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez and Andreas Koch* in *EMSOFT '19: Proceedings of the International Conference on Embedded Software Companion*. The contribution of the author of this thesis is summarized as follows.

» *As the corresponding and leading author, Lukas Sommer was responsible for the design of the experiments, the setup of all evaluated hardware platforms, and the OpenMP-based implementation of the benchmark kernels. Florian Stock was in charge of the extraction of the benchmark kernels from Autoware and the CUDA-based implementation. The OpenCL-based implementation was contributed by Leonardo Solis-Vasquez. Lukas Sommer was responsible for the manuscript's text, with feedback from Florian Stock and Andreas Koch.* «

ABSTRACT

Due to the ever-increasing computational demand of automotive applications, and in particular autonomous driving capabilities, the automotive industry and its suppliers are starting to adopt parallel and heterogeneous embedded computing platforms.

However, `C` and `C++`, the currently dominating programming languages in this industry, do not provide sufficient mechanisms to fully exploit such platforms. As a result, vendors have begun to employ true parallel programming models such as OpenMP, CUDA or OpenCL.

In this work, we report on a benchmark suite developed specifically to investigate the applicability of established parallel programming models to automotive workloads on heterogeneous platforms.

8.1 INTRODUCTION

The computational demands of automotive applications has increased steeply in recent years, in particular due to autonomous driving and advanced driver-assistance (ADAS) functionalities.

To meet this new computational demand, the automotive industry is starting to turn towards parallel and heterogeneous platforms. The multi-core processors and accelerators (e.g., GPUs) found on heterogeneous platforms typically require programming language support to explicitly express parallelism. However, `C` and `C++`, the currently dominating programming languages in the automotive field, lack sufficient mechanisms. As a consequence, the automotive industry is keenly interested in parallel programming models.

While a number of well-established standards for parallel and heterogeneous programming exist in the HPC community, the embedded target platforms used in automotive applications differ significantly from the HPC systems these programming models were originally tailored for.

In this work, we present the DAPHNE (Darmstadt Automotive Parallel HeterogeNEous) open-source benchmark suite [2], comprising multiple kernels from automotive applications, together with parallel implementations in different programming models and for different embedded computing platforms. This suite allows to study the applicability of parallel programming models to different automotive workloads and their performance on embedded, heterogeneous systems. The insights can also be used to establish a set of best practices, potential adaptions and possible extensions of the investigated parallel programming models.

8.2 BENCHMARK SUITE

Our goal for the development of the benchmark suite was to extract actual compute-intensive automotive workloads into easy-to-analyze standalone kernels for parallelization.

For the serial implementations of the automotive workloads that serve as the basis for our benchmarks suite, we reviewed multiple open-source frameworks for ADAS. In this process, we found Autoware [3] to be the most promising candidate as the source of the serial implementations.

Autoware contains algorithms for all steps of an AD/ADAS application, including sensing, perception, planning, decision making, and actuation. As such, the modules contained in Autoware are representative for the kind of computations found in real-world automotive applications.

Using the test dataset acquired from a real test drive and provided by Autoware, we used profiling to identify execution hotspots. As our

Table 8.1: Benchmark kernel statistics

Kernel	LoC	Input [MB]	Output [MB]	Data Sets
points2image	347	19 000.000	4 300.000	2500
euclidean_clust.	956	45.000	54.000	350
ndt_mapping	1394	4 200.000	0.008	115

aim was to accelerate code with a parallel execution model, only those code parts that seemed promising for parallelization were considered for extraction.

Currently, our benchmark suite contains three different kernels. Each kernel was provided in a skeleton for standalone execution outside of the complex ROS-based Autoware framework. References to third-party libraries were either inlined or replaced with custom implementations. To make sure that we did not introduce any artificially slow sections in this process, we compared the performance of our implementation to that of the original Autoware implementation (for an example, see Fig. 8.1).

Additionally, we extracted five data sets of increasing size with input- and reference- data for each kernel, which we provide together with the benchmark suite, see Table 8.1 for statistics.

After extraction, we parallelized each kernel using different programming models. We chose OpenMP, CUDA and OpenCL for parallelization, because these standards are supported on most embedded, heterogeneous platforms. For each kernel, we provide an implementation in each of the programming models. In the case of OpenMP we also provide implementations using the new device offloading features. For the points2image-kernel, we additionally created an implementation for Xilinx Zynq Ultrascale+ MPSoC using OpenCL with Xilinx SDAccel.

8.2.1 *points2image Kernel*

The points2image-routine gets as input a point cloud (which originates from a LIDAR) with intensity and range information. The points are then projected onto a given 2D view.

With thousands of points being projected at the same time, this seemed like a very good candidate for parallelization. However, in practice, multiple of the 3D LIDAR points may end up being projected to the same point in the 2D view. This leads to race conditions in parallel execution, and can result in incorrect results when not handled with care.

The original Autoware code prioritizes the point that is *closer* to the projected view. This behavior is implemented in our parallel versions

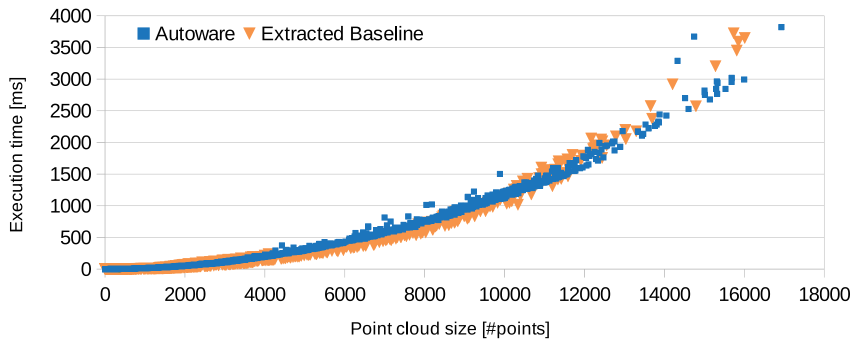


Figure 8.1: Comparison of different `radiusSearch` implementations.

by using atomic min functions in the threads accessing the 2D view. In case of accessing the same 2D element, the atomic min function guarantees that the correct thread writes its value.

Similar atomic and synchronization functions can be used to keep the array with the corresponding intensity up to date. While these atomic operations diminish the resulting speed up, the accelerators were still able to improve the run time by up to a factor of 3.5x.

8.2.2 *euclidean_clustering Kernel*

Similar to points2image, the euclidean_clustering-kernel also operates on a point cloud. The algorithm clusters points that are close together, to identify objects.

The original implementation uses a *k*d-tree to compute the distances between points. This was provided by the Point Cloud Library (PCL, [4]). To remove the dependency on the PCL and enable stand-alone, library-less compilation and execution, a custom implementation was created. As the distances from *all* points to *all* others were required, our replacement for the `radiusSearch` function from PCL employs a look-up of the distances in a precomputed table. The table does not need to hold the distances between two points, but just a boolean indicating if the distance between those points is below the given threshold. This reduces the memory required for the table significantly.

To verify that this standalone kernel has a similar performance to the original one, it was benchmarked against the original Autoware version. Fig. 8.1 shows that the performance of the two implementations is almost identical.

8.2.3 *ndt_mapping Kernel*

The ndt_mapping kernel is a SLAM (simultaneous location and mapping) algorithm that works with *Normal Distribution Transformations* (NDT, [1]). As input, two point clouds are accepted, one representing the current LIDAR scan, and a second one representing the map built during the drive so far. The newly discovered points from the LIDAR scan are then added to the map by *aligning* them with the existing map.

The computational result is a transformation consisting of a rotation and a translation. The transformation is the best transformation that fits the newly acquired LIDAR point cloud to a pre-existing map point cloud using NDT.

To make the kernel code library free, we replaced the complex SVD algorithm, with a much simpler Gaussian solver. In general the SVD solver deals better with singular matrices, but for our purpose, the precision of the replacement turned out to be sufficient.

Beyond that, we again replaced a radius search within a *k*d-tree. In this case, the search was not used to compute the distances from all points to each other within the *k*d-tree. Instead it was used to actually find the *closest* point inside the tree to a point outside the tree. This search was implemented as a linear search within the point cloud. Our evaluation showed that neither of these changes slows down the sequential baseline. On the contrary, they actually *improve* the runtime of the sequential baseline, and thus do not artificially inflate the speedups we achieve through parallelization.

8.3 conclusion and outlook

We presented the DAPHNE open-source [2] benchmark suite, comprising three different workloads typical for automotive applications. For each of the kernels we provided parallelizations with CUDA, OpenCL and OpenMP, all qualified for execution on actual embedded computing platforms.

The benchmark suite can be used to assess the applicability of established parallel programming models to automotive workloads, or to evaluate new compute platforms for the AD/ADAS domains.

In the future, we plan to extend the benchmark suite by adding more kernels and parallelization with other programming models, such as SYCL. Besides that, we will use the benchmark suite to investigate the performance of established parallel programming models on embedded heterogeneous platforms and how these models can be adapted to better meet the needs of the automotive industry.

acknowledgments

This project was funded by VDA FAT as part of the research of AK31. The authors would also like to thank Xilinx for supporting this work by donations of hard- and software, as well as Renesas and Codeplay for their help with the acquisition of the V3M and the timely support for using their programming tools.

REFERENCES

[1] P. Biber and W. Strasser. "The normal distributions transform: a new approach to laser scan matching." In: *Intl. Conf. on Intelligent Robots and Systems (IROS 2003)*. Oct. 2003.

[2] *DAPHNE Benchmark Suite*. `https://github.com/esa-tu-darmstadt/daphne-benchmark`. Accessed: 2019-08-01. 2019.

[3] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. "An Open Approach to Autonomous Vehicles." In: *IEEE Micro* 35.6 (Nov. 2015), pp. 60–68. ISSN: 0272-1732.

[4] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)." In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. Shanghai, China, May 2011.

9 USING PARALLEL PROGRAMMING MODELS FOR AUTOMOTIVE WORKLOADS ON HETEROGENEOUS SYSTEMS - A CASE STUDY

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *Using Parallel Programming Models for Automotive Workloads on Heterogeneous Systems - a Case Study* by *Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez and Andreas Koch* in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. The contribution of the author of this thesis is summarized as follows.

» *As the corresponding and leading author, Lukas Sommer was responsible for the design of the experiments and the conception of the metrics employed in the evaluation. Florian Stock was in charge of the extraction of the benchmark kernels from Autoware and the CUDA-based implementation. The OpenCL-based implementation was contributed by Leonardo Solis-Vasquez. The setup of all evaluated hardware platforms and the OpenMP-based implementation of the benchmark kernels was done by Lukas Sommer. He was also responsible for the manuscript's text, with feedback from Florian Stock, Leonardo Solis-Vasquez and Andreas Koch.* «

ABSTRACT

Due to the ever-increasing computational demand of automotive applications, and in particular autonomous driving functionalities, the automotive industry and supply vendors are starting to adopt parallel and heterogeneous embedded platforms for their products.

However, `C` and `C++`, the currently dominating programming languages in this industry, do not provide sufficient mechanisms to target such platforms. Established parallel programming models such as OpenMP and OpenCL on the other hand are tailored towards HPC systems.

In this case study, we investigate the applicability of established parallel programming models to automotive workloads on heterogeneous platforms. We pursue a practical approach by re-enacting a typical development process for typical embedded platforms and representative benchmarks.

9.1 INTRODUCTION

In recent years, the computational demands of automotive applications have been steeply increasing, in particular with the introduction of advanced driver-assistance (ADAS) and, at least partially, autonomous driving (AD) functionalities.

As these functionalities require the processing of complex, compute-intensive algorithms with high performance, the automotive industry faces challenges similar to those encountered by the high-performance computing (HPC) community about a decade ago. The computational power provided by single-core embedded processors is not sufficient anymore to meet latency and/or throughput requirements.

In reaction to these challenges, the automotive industry is starting to turn towards parallel and heterogeneous platforms [9], e.g., combining multi-core CPUs and GPUs. These multi-core processors as well as accelerators typically require programming language mechanisms to express parallelism and leverage their computational power. However, `C` and `C++`, the currently dominating programming languages in the automotive field [8], do not provide sufficient mechanisms. As a consequence, the automotive industry needs to adopt parallel and heterogeneous programming models.

While there is a number of well-established standards for parallel and heterogeneous programming in the HPC community, the embedded target platforms in the automotive industry differ significantly from the HPC systems these programming models were tailored towards. The thermal and power budget, the computational power and the coupling between host CPU and accelerator of embedded, heterogeneous platforms deployed in automotive vehicles differs significantly from HPC systems.

So although the HPC programming models can serve as a solid base, they are not usable "out-of-the-box", and will require adaption for use in automotive usage scenarios.

The aim of this case study is to investigate programming models established in the HPC field with regard to their applicability in embedded automotive applications. The intent of this case study is to provide insights into how well established programming models are suited for use in automotive applications, and how they could be improved and extended for automotive use and target platforms.

Note that we do not focus on the raw performance only, but also consider other important aspects for a *practical* usage of this parallel programming models in industry, such as programmer productivity and maintainability.

To this end, we have developed a practical approach described in Section 9.3. In our evaluation, we present a detailed analysis of important figures, e.g., programmer productivity and effort to reach certain levels of speedup over a serial implementation.

9.2 RELATED WORK

Prior studies that investigated the usability and maintainability of parallel programming models, such as [4] or [5], focused on HPC applications and algorithms, whereas our case study is focused on automotive, embedded applications. Most of these studies were also conducted as *classroom studies*, with novice programmers as developers. In this work, we explicitly do not consider the time required to learn a parallel programming model, and have experienced developers implement the kernels.

In other work, such as [18] or [12], the authors developed static and dynamic analyses to predict the performance of parallel implementations of algorithms on different embedded and also heterogeneous platforms. While they consider the underlying parallel characteristics of the algorithms and how well they map to the platforms, we investigate how well the parallelism in an algorithm can be *expressed* with the different parallel programming models, and how much programming effort is required to do so.

9.3 APPROACH/METHODOLOGY

The central aim of this case study is to investigate the use of existing (often HPC-centric) programming models for the implementation of automotive computation tasks on parallel, heterogeneous platforms, and to identify potential areas for improvement of the existing standards or tool implementations.

To this end, and in contrast to previous investigations (e.g., [8]), we take a practical and quantitive approach, based on real implementations of representative computational problems. The basic idea of our approach is to re-enact the typical development process of migrating an existing, serial implementation of an algorithm to a parallel, heterogeneous platform. With the continued integration of such platforms into automotive vehicles, many OEMs and component suppliers will be confronted with this task.

Through this re-enactment, we can investigate all relevant usability aspects of the programming models in detail, as well as the ecosystem of supporting tools.

In the following sections we will describe the individual steps of our approach in more detail.

9.3.1 *Identification of relevant programming models*

In a first step, we need to identify the candidate programming models, which we will use for implementation in our case study.

For parallel programming and the integration of dedicated acceleration, a number of programming models and standards already exist, mostly originating from the high-performance computing domain. As `C` and `C++` are the dominant programming languages in the automotive domain at this point, having almost 50% share [8], we will focus on programming models that are based upon at least one of these languages. Beyond that, we further tighten that focus to well-established programming languages with an active community, to make sure sufficient training resources and experts are available.

After reviewing the parallel programming models currently enjoying the most prominence, we selected OpenMP [11], OpenCL [10] and CUDA [2] as candidate models. The three models cover a broad spectrum, ranging from the rather high-level abstractions of OpenMP to the very explicit parallelization and offloading of OpenCL.

9.3.2 *Benchmark Selection*

The beginning of the re-enacted migration process of an existing application to a parallel, heterogeneous platform usually is the serial implementation of an algorithm. As the central aim of this project is to investigate the applicability of the programming models to *automotive* software, we chose to use algorithms from the automotive domain and their corresponding serial implementations as starting points for our implementation.

After review, we selected the open-source DAPHNE benchmark suite [15, 16] as the source for the serial implementations. The DAPHNE suite contains three different automotive kernels, called points2image, euclidean_clustering and ndt_mapping, that were extracted from the Autoware autonomous driving framework [6]. In addition, the benchmark also provides datasets with input- and reference data captured during an actual drive, that we can use to ensure the correctness of our parallel implementations.

9.3.3 *Selection and Bring-Up of Evaluation Platforms*

For testing and performance evaluation of the benchmark implementations, suitable evaluation platforms are required. In the selection process of these platforms, our central aim was to cover a broad range of current embedded, parallel and heterogeneous platforms. After a review of available technologies as step three of our approach, three different platforms were acquired:

- Nvidia Jetson TX2
- Nvidia Jetson AGX Xavier
- Renesas R-Car V3M

All three selected platforms combine a multi-core CPU with a GPU (called *image recognition engine* in case of the V3M) and are designed for automotive usage scenarios. As such, they exhibit the particular characteristics regarding computational power and energy budget typically found on automotive platforms.

9.3.4 *Benchmark implementation and porting.*

The fourth step of our approach is the actual implementation process of the benchmarks that lies at the heart of our practical, quantitative approach.

In contrast to many other surveys (e.g., [4]), we explicitly do not consider the time required to *learn* a parallel programming model here. In our implementation case study, the developers performing the implementations are already experts with multi-year experience with the respective programming models. This is similar to a real-world scenario, where companies are likely to hire developers that are familiar with programming models and have prior experience in their use.

During the implementation, the original serial code is parallelized using the means provided by the respective programming model. Additionally, the compute-intensive parts of the application are offloaded onto the parallel accelerators, i.e., the GPU, if the programming model allows to do so.

This implementation flow replicates the typical process of migrating an existing, serial code base to a new parallel, heterogeneous platform. Beyond that, many of our insights should also be applicable to the development process of new software from scratch, i.e., without a pre-existing serial implementation.

Once an application has been parallelized, it should be deployable to multiple *different* heterogeneous compute platforms, therefore *portability* plays a major role for the applicability of a programming model for the automotive domain.

To assess the portability of the selected programming models and the resulting development effort, we also re-enact the typical process of porting an application to a different platform. To this end, the resulting parallel implementation of a benchmark targeting an initial platform is also evaluated and optimized at least on a second one.

While our practical approach allows us to investigate the programming models in a real-world scenario, it does have a number of limitations that might make it less suitable for other purposes.

- The kernels were selected to study the parallelization effort for different parallel paradigms/platforms. They represent some automotive workloads, but not *all* automotive workloads.

- With Autoware's roots in fundamental academic research, their implementation is not necessarily performance optimized. Similarly, the highly modular ROS-based structure does carry a performance overhead, as it is very difficult (or not even possible at all) to optimize data transfers between host and accelerator memories across ROS node boundaries.

9.4 EVALUATION

The raw performance of the parallel implementations is *not* the key aspect of the programming models we want to investigate in this case study. But even so, achievable performance plays a crucial role when judging a parallel programming model, and can therefore not be completely neglected in this case study.

However, for the business decision on which programming model to use for the implementation on heterogeneous platforms, the following three non-functional aspects of programming models need to be considered as well:

- Programmer Productivity
- Maintainability
- Portability

All three aspects of programming models listed above are "soft" characteristics, i.e. they cannot be measured directly. Rather, one needs to quantitatively assess them indirectly through a combination of multiple metrics. To this end, we have assembled a set of measurements and metrics described in the following. After the definition of our metrics, we will investigate each of the listed aspects in Sections 9.4.1 to 9.4.3.

PROGRAMMER PRODUCTIVITY The productivity a developer achieves using a given programming model gives insights into the ease-of-use of the model.

We use a simultaneous tracking of working hours vs. achieved performance to determine which programming model yields the required performance with the least development effort. For many applications, a performance *lower* than the maximum achievable performance on a given platform is absolutely sufficient, e.g., to meet real-time requirements. In such a case, a programming model that achieves the *required* performance faster than the other models, even though this model may not be able to deliver the best peak performance, is preferable.

In our case study, the developers measure performance roughly every sixty minutes, resulting in graphs similar to the ones shown in Fig. 9.1, Fig. 9.2, and Fig. 9.3.

MAINTAINABILITY Application software in the automotive field typically has a relatively long lifetime, potentially extending to over more than a decade. Thus, good maintainability is indispensable. The effort required for the maintenance of a piece of code is dominated

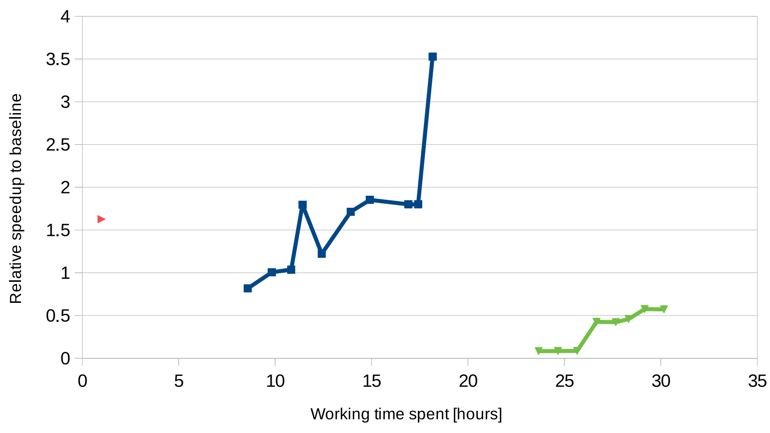


Figure 9.1: Result of simultaneous tracking of working hours vs. speedup over serial baseline to assess programmer productivity for benchmark points2image.

by the time that a developer, who is not the original author of the code, needs to become familiar with the code base in order to make the desired changes.

The maintenance effort is influenced by the code volume and the complexity of the code. To assess the impact of parallel programming models on the code volume, we measure the number of changed lines compared to the original serial version of the code. In contrast to prior work [4], we also consider *in-place* changes, because many parallel programming models also require to restructure the original code of the application.

Assessing the complexity added to the code base due to the use of a parallel programming model is more complicated. Classical software complexity metrics such as the ones proposed by McCabe [7] or Halstead [3] are tailored towards control-flow heavy business software, and are not suitable for this purpose. We therefore developed a new metric, the *Complexity Count*. The reasoning behind the complexity count, is that complexity introduced by parallel programming models stems from the inclusion of new keywords, new datatypes, runtime function calls and compiler directives defined by the programming model into the code of an application. To calculate the complexity count, we simply count the number of these additional programming model constructs in the code base, and also the number of parameters passed to these constructs, e.g., to runtime functions. For example, the use of `cudaMallocManaged(&a, vector_size * sizeof(float));` in the code would yield a complexity count of three.

PORTABILITY Once a code base has been parallelized and partially offloaded to dedicated accelerators, it should ideally be usable for multiple *different* heterogeneous platforms. The characteristics of a programming model (e.g., high-level abstractions, compiler directives,

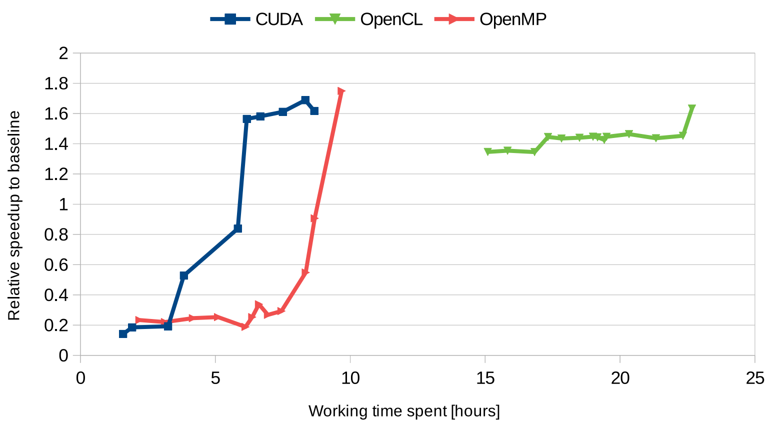


Figure 9.2: Result of simultaneous tracking of working hours vs. speedup over serial baseline to assess programmer productivity for benchmark euclidean_clustering.

etc.) can directly influence the portability. It is thus important to assess the porting effort required for each of the selected models.

Besides making the existing code compile, and compute correctly on the new platform, porting typically involves a process of incremental improvements to optimize performance on a new platform. The duration of this process indirectly provides information about the portability characteristic of a programming model. We use a simultaneous tracking of working hours spent on porting an existing implementation vs. the performance on the new platform, similar to the one we employed to measure the programmer productivity.

9.4.1 *Programmer Productivity*

While the three programming techniques employed in the study have different effort vs. performance curves in Fig. 9.1-Fig. 9.3, a trend is clear across the benchmark kernels.

OPENMP IMPLEMENTATION Across all three benchmarks investigated in our implementation case study, OpenMP typically requires the least effort to parallelize an application. For example, the parallelization of the points2image benchmark in Fig. 9.1 takes only a single hour of development effort. In general, OpenMP, mainly based on compiler directives, benefits from the fact that it typically requires less invasive restructuring for parallelization than other programming models. This also implies that the performance of the application can be assessed throughout the development cycle, which can also be a big plus for development productivity.

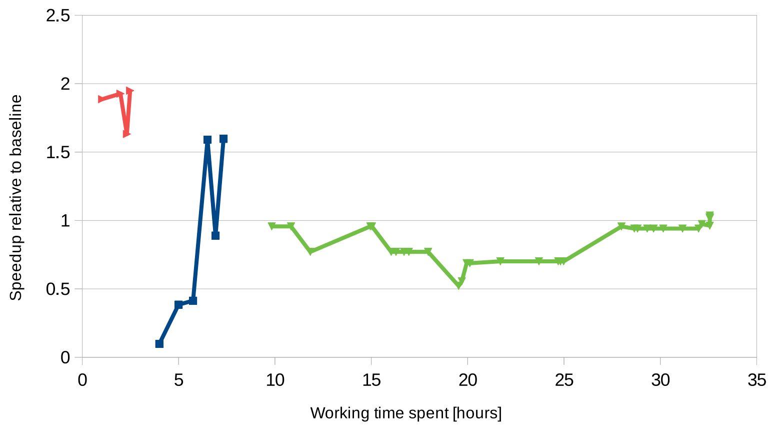


Figure 9.3: Result of simultaneous tracking of working hours vs. speedup over serial baseline to assess programmer productivity for benchmark ndt_mapping.

CUDA IMPLEMENTATION CUDA typically also allows for fast parallelization. Once a parallelization approach is determined, it can often be realized in just a few hours for kernels with the complexity of our benchmarks (e.g. Fig. 9.2 and Fig. 9.3). Performance-wise, OpenMP and CUDA are mostly comparable, the lower number of threads available on the CPU (note that we focus on the CPU-based features of OpenMP here!) and the overhead of offloading computation and data to the more powerful GPU often cancel each other out.

OPENCL IMPLEMENTATION For the majority of the benchmarks, OpenCL requires much up-front work to restructure and partition the application, resulting in a phase where performance cannot be assessed to determine the prospects of success for the chosen parallelization strategy. In Fig. 9.1 and Fig. 9.2, this is indicated by the late start of the green curve for OpenCL. The relatively complex host code for OpenCL, and the invasive changes to the serial implementation, also cause OpenCL to often require the most effort for parallel and heterogeneous implementation.

9.4.2 *Maintainability*

The ranking with regard to the required development effort also correlates with the results that we get from our metrics for *maintainability*. The evaluation of the total number of line changes (added or deleted) in relation to the LoC of the original, serial implementation is given in Fig. 9.4.

Because OpenMP allows to reuse the serial implementation almost without changes in most cases, and only requires to add the descrip-

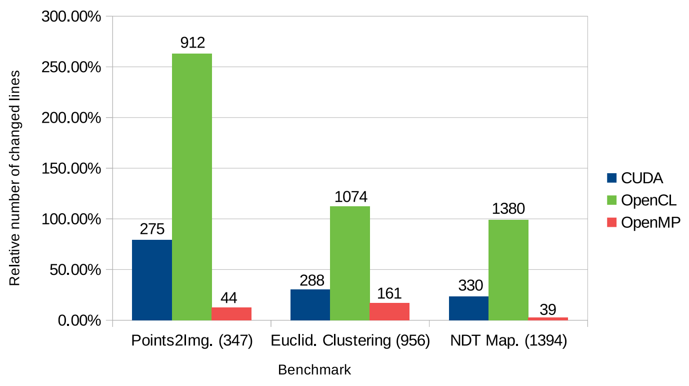


Figure 9.4: Number of changed lines relative to serial baseline. Numbers in parentheses give LoC of serial implementation.

tion of parallel semantics through compiler directives, the number of changes is relatively small (3%-17%).

In contrast, CUDA requires kernel functionality to be extracted to dedicated device functions and the inclusion of additional API calls into the host code, resulting in a significantly higher number of changes (24% to 80%).

For OpenCL, the extraction of device code to separate files and the inclusion of even more boilerplate code into the host leads to sweeping changes in the code base, ranging from 99% to 263%.

To assess the additional complexity introduced by the use of parallel programming models into an application's code, we use the *Complexity Count*. The counts for all benchmarks and models are given in Table 9.1.

Benchmark	**CUDA**	**OpenCL**	**OpenMP**
points2image	70	329	12
euclidean_clustering	17	120	15
ndt_mapping	64	113	28

Table 9.1: Complexity count.

Because the OpenMP compiler directives add parallel semantics in a descriptive/prescriptive manner and operate on a high level of abstraction, the complexity added by new keywords and directives is very low.

For CUDA, the added complexity has two main sources: New data-types and keywords are added on top of the `C++` programming language, mainly to partition the application between host and device. Additionally, a number of API functions has to be called in order to transfer data and execution to the device. Nevertheless, the complexity added is still moderate.

In OpenCL, the sources of complexity are similar to CUDA, namely new data-types and keywords for the device section and API calls in the host code. However, OpenCL requires much more host code than CUDA, resulting in significantly higher complexity counts.

For OpenCL, there is also a considerable difference between the use of the traditional C-API and the C++ wrapper API: While the points2image benchmark was implemented with the C-API, the other two benchmarks use the C++ wrapper API for the host code. Using the latter, some steps of the host-side setup process are abstracted, resulting in a notably smaller complexity count.

9.4.3 *Portability*

Similar to the discussion of development effort vs. performance, we can also see a trend of using the three different programming methods in terms of their portability.

The high-level of abstraction supported by OpenMP also benefits *portability*. Moving an existing, parallel OpenMP implementation to another platform typically boils down to a simple re-compilation on the new platform, taking less than 20 minutes to complete for each of our benchmarks.

For CUDA, the situation is similar. When moving from one platform to another, the code usually does not need to be changed, thanks to the standardization of the CUDA language by Nvidia for all its devices, and only a small number of parameters needs to be tuned.

With OpenCL, things are different. Basic features, such as support for double-precision floating-point arithmetic are only *optional* features, and different vendors typically support different versions of the OpenCL specification. To these they might add extensions that only work on platforms manufactured by this vendor.

For two of our three benchmarks, this required manual changes that often took hours. For example, adapting the points2image benchmark for the Renesas V3M platform required code changes to use only single precision floating point computations, instead of the double precision of the original code. This required almost 12 hours of development time.

9.5 CONCLUSION

In this study, we have taken a very practical approach to evaluate the applicability of today's parallel programming models in the automotive domain. We considered both the nature of typical automotive compute kernels, which are often very short compared to HPC kernels, and the constraints of actual embedded hardware platforms.

Based on our insights, we cannot declare a single "winning" programming model here. However, our experiences should serve as a first indicator for the applicability of different programming models, and show a way forward to future development and research.

The high-level abstractions defined by the **OpenMP** standard allowed for a very good programmer productivity. For the actual parallelization, OpenMP relies on the compiler, which yielded competitive performance for our benchmarks. However, we were yet not able to use the device offloading features recently added to the standard due to insufficient compiler support on the target platforms. Future research should investigate the possibility to extend the compiler support for OpenMP to such target platforms and workloads in more depth (e.g., use OpenMP to target FPGAs [13, 14]).

CUDA strikes a balance between high-level abstractions and explicit parallelization. In combination, this allows reasonable programmer productivity and good performance. However, beyond the official, proprietary compilers and runtimes from Nvidia, no competitive open implementations for CUDA exist. Thus, the use of CUDA carries the risk of vendor lock-in. Alternatives for moving CUDA outside the Nvidia ecosystem (e.g., AMD HIP/ROCm [1]) are only slowly appearing and need further evaluation in future research.

In contrast to CUDA, implementations with **OpenCL** require much more host code, and far more invasive restructuring of the application. The partitioning into multiple files for host and device code causes a large up-front effort for implementation, before parallelization and optimization can even be started.

With **SYCL** as a spiritual successor to OpenCL, the Khronos Group provides a modern, open standard designed to overcome these limitations of OpenCL. As soons as SYCL becomes available on more embedded platforms, future research should investigate the use of SYCL for the implementation of automotive workloads on the corresponding target platforms.

More details on the evaluation and an investigation of FPGAs as accelerators is available in the technical report [17].

REFERENCES

[1] Advanced Microdevices, Inc. *HIP*. https://gpuopen.com/compute-product/hip-convert-cuda-to-portable-c-code/. 2019.

[2] *CUDA C programming guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2019-02-05. 2018.

[3] Maurice Howard Halstead et al. *Elements of software science*. 1977.

[4] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz. “Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers.” In: *Supercomput-*

ing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference. Nov. 2005.

[5] Lorin Hochstein and Victor R. Basili. "An Empirical Study to Compare Two Parallel Programming Models." In: *ACM SPAA*. Cambridge, Massachusetts, USA, 2006. ISBN: 1-59593-452-9.

[6] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. "An Open Approach to Autonomous Vehicles." In: *IEEE Micro* 35.6 (Nov. 2015), pp. 60–68. ISSN: 0272-1732.

[7] Thomas J McCabe. "A complexity measure." In: *Transactions on Software Engineering* (1976).

[8] Zaur Molotnikov, Konstantin Schorp, Vincent Aravantinos, and Bernhard Schätz. *FAT-Schriftenreihe 287 - Future Programming Paradigms in the Automotive Industry*. Tech. rep. 2016.

[9] Nvidia Inc. *Nvidia Drive platform*. https://developer.nvidia.com/drive.

[10] *OpenCL Specification*. https://www.khronos.org/opencl. 2019.

[11] OpenMP Architecture Review Board. *OpenMP Application Programming Interface - OpenMP Standard 5.0*. Nov. 2018.

[12] R. Saussard, B. Bouzid, M. Vasiliu, and R. Reynaud. "Optimal Performance Prediction of ADAS Algorithms on Embedded Parallel Architectures." In: *2015 Intl. Conf. on High Performance Computing and Communications*. Aug. 2015, pp. 213–218. DOI: 10.1109/HPCC-CSS-ICESS.2015.95.

[13] Lukas Sommer, Jens Korinth, and Andreas Koch. "OpenMP Device Offloading to FPGA Accelerators." In: *2017 IEEE 28th Intl. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*. 2017.

[14] Lukas Sommer, Julian Oppermann, Jaco Hofmann, and Andreas Koch. "Synthesis of Interleaved Multithreaded Accelerators from OpenMP Loops." In: *2017 Intl. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*.

[15] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. "DAPHNE - An Automotive Benchmark Suite for Parallel Programming Models on Embedded Heterogeneous Platforms: Work-in-Progress." In: *Proceedings of the International Conference on Embedded Software Companion*. EMSOFT '19. New York, New York: Association for Computing Machinery, 2019. DOI: 10.1145/3349568.3351547. URL: https://doi.org/10.1145/3349568.3351547.

[16] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. *Darmstadt Automotive Parallel HeterogeNEous (DAPHNE) Benchmark-Suite*. https://github.com/esa-tu-darmstadt/daphne-benchmark. 2019.

[17] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. *FAT-Schriftenreihe 317 - EPHoS: Evaluation of Programming-Models for Heterogeneous Systems*. Tech. rep. 2019. URL: https://www.vda.de/de/services/Publikationen/fat-schriftenreihe-317.html.

[18] X. Wang, K. Huang, A. Knoll, and X. Qian. "A Hybrid Framework for Fast and Accurate GPU Performance Estimation through Source-Level Analysis and Trace-Based Simulation." In: *HPCA*. Feb. 2019.

10 OPENMP DEVICE OFFLOADING FOR EMBEDDED HETEROGENEOUS PLATFORMS - WORK-IN-PROGRESS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *OpenMP Device Offloading for Embedded Heterogeneous Platforms - Work-in-Progress* by *Lukas Sommer and Andreas Koch* in *2020 International Conference on Embedded Software (EMSOFT)*. The contribution of the author of this thesis is summarized as follows.

> *» As the corresponding and leading author, Lukas Sommer was responsible for the extension of the LLVM OpenMP offloading infrastructure for heterogeneous GPU platforms with physically shared memory, and the implementation of the benchmark kernels with OpenMP offloading. He also performed the experiments and was in charge of the manuscript's text, with feedback from Andreas Koch. «*

ABSTRACT

The growing computational demands of automotive applications require the use of powerful embedded, heterogeneous computing platforms in vehicles. OpenMP, and in particular its device offloading features, are a promising candidate programming model for these platforms.

In this work, we show how typical automotive workloads can be implemented and optimized with OpenMP device offloading. To this end, we also adapt the LLVM OpenMP runtime to embedded, heterogeneous platforms. Our evaluation shows that OpenMP device offloading can deliver performance similar to that of optimized CUDA implementations.

10.1 INTRODUCTION

As modern driver-assistance and autonomous driving functionalities demand large amounts of computational power on-board of vehicles, the automotive industry is starting to adopt embedded, heterogeneous platforms, such as the Nvidia Drive system, to supply the required computational power.

Programming these systems can be a challenging task and requires a programming model suitable for the platform. In their study, Sommer et al. [4] identified OpenMP as a very interesting candidate, in particular due to its ease-of-use and maintainability. With the device offloading features introduced in version 4.0, and further refined in newer versions of the OpenMP standard, OpenMP now also allows to target *heterogeneous* systems, e.g., combining a multi-core CPU and a GPU. However, in their study [4], Sommer et al. also found the compiler support for OpenMP device offloading to still be limited on embedded systems. Since the study has been conducted, the OpenMP support in compilers has evolved, e.g., with the LLVM compiler infrastructure now supporting OpenMP device offloading on ARM-based systems.

In this work, we show how OpenMP offloading can be used for automotive workloads on embedded heterogeneous platforms by accelerating three automotive workloads from the open-source DAPHNE benchmark suite [3] on an embedded platform combining a multi-core CPU and a GPU. We also modify the LLVM OpenMP runtime to facilitate management of shared memory on embedded platforms.

10.2 IMPLEMENTATION

We use the three automotive benchmark kernels points2image (P2I), euclidean_clustering (EC) and ndt_mapping (NDT) from the open-source DAPHNE suite [3] to demonstrate how OpenMP device offloading can be used to accelerate performance-critical parts of automotive applications. All three kernels represent typical automotive workloads and have been extracted from the open-source Autoware framework for autonomous driving [1].

As OpenMP constructs for device offloading differ from the parallel constructs for CPUs, a serial implementation in pure C++ is used as starting point for the study. For better comparability with the handwritten CUDA implementation provided in the DAPHNE suite, the same performance-critical sections of the program are offloaded to the GPU.

After adding the OpenMP offloading constructs to the code (e.g., `omp target teams distribute`), it would already be possible to run the kernels and offload the OpenMP target regions to a GPU. However, this implementation does not yet exploit one of the most important differences between *embedded* heterogeneous platforms and heterogeneous systems typically found in HPC domains: on embedded platforms, the different components of the system often *physically* share the same memory, as it is the case for our target system, the Nvidia Jetson platform [2].

We therefore modify the LLVM OpenMP runtime to allow for allocation of memory that can be accessed by both, the host CPU and the GPU. Using the standard OpenMP function `omp_target_alloc`, it is now possible to allocate memory accessible by *both* components, and avoid expensive data-copies. These data-copies made up for 88% of the execution time for points2image and 49% for euclidean_clustering, so allocating memory usable by both CPU and GPU significantly reduces the kernel execution time. For the benchmark ndt_mapping, execution was not even possible with application data allocated twice (once in host memory, once in GPU memory) because this wasteful allocation exceeded the system memory.

It is also possible to further optimize the ndt_mapping application performance: Profiling the kernel shows that one of the target regions is 10x slower than its CUDA counterpart. Further investigation of the region shows that the atomic update (`omp atomic update`) is compiled to a somewhat inefficient PTX code sequence, whereas the handwritten CUDA version uses the CUDA builtin function `atomicAdd`.

But using OpenMP's `declare variant` mechanism, a specialized function for the atomic update can be defined. This specialized function for the CUDA architecture will automatically be selected by the compiler when offloading to the CUDA architecture, whereas a generic version of the function will be used for other architectures, keeping the OpenMP-based implementation portable.

10.3 EVALUATION

We use an Nvidia Jetson AGX Xavier platform to compare a total of five different implementations for the three different kernels:

- Serial baseline implementation in pure C++ (Serial).
- CPU-only OpenMP-implementation (OMP CPU).
- OpenMP offloading implementation assuming separate memory (OMP Offloading).
- OpenMP offloading implementation optimized for physically shared memory (OMP Phys. Shared Mem.).
- Hand-written CUDA implementation (CUDA).

Fig. 10.1 shows the accumulated runtime for each kernel (averaged over three runs), the number of invocations corresponds to the *full* data-set of the DAPHNE benchmark and is given in parentheses. The

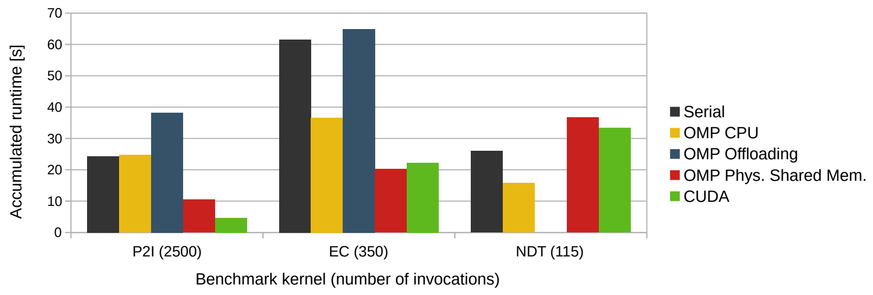


Figure 10.1: Accumulated runtime of benchmark kernel execution in seconds.

bar for *OMP Offloading* for ndt_mapping is missing due to the reasons explained in the previous section.

For the first two applications, the use of physically shared memory (OMP Phys. Shared Mem.), as enabled by our modified version of LLVM's `libomptarget`, dramatically improves the execution time compared to the version assuming separate memory. In case of the points2image kernel, there is still a significant gap between the OpenMP offloading version with physically shared memory and the handwritten CUDA implementation, but the OpenMP offloading nevertheless clearly outperforms the CPU-only OpenMP implementation. The OpenMP offloading implementation with physically shared memory of euclidean_clustering even outperforms the optimized CUDA implementation by a small margin, making the OpenMP offloading the fastest implementation of this benchmark kernel. For ndt_mapping, there is a small difference in performance between OpenMP offloading with physically shared memory and CUDA implementation, and both versions are not able to keep up with the CPU-only implementation of this kernel.

Table 10.1 further investigates the difference between OpenMP offloading with physically shared memory and hand-written CUDA implementations by looking at the execution time per invocation of the different GPU regions as given by `nvprof`. As the region execution time is almost equal for points2image, the performance difference is most likely caused by the OpenMP runtime itself. For euclidean_clustering, the execution time for the second region is almost identical, the small advantage of OpenMP offloading over CUDA stems from the difference in execution time of the first region. With the OpenMP offloading version for ndt_mapping, both regions are slightly slower than their CUDA pendants. However, an investigation of the original implementation shows that the implementation of the specialized atomic update using OpenMP `declare variant` reduces the execution time of Region 1 by a factor of more than 7x (from 87,890µs to 12,022µs).

Table 10.1: Average runtime per kernel invocation in μs.

Benchmark	Region	Calls	OMP PSM [μs]	CUDA [μs]
P2I	Region 1	2,500	118	121
EC	Region 1	1,726	541	1,728
	Region 2	191,132	8.8	7.4
NDT	Region 1	115	12,022	9,784
	Region 2	115	35,008	33,829

10.4 CONCLUSION & OUTLOOK

In this work, we have demonstrated how OpenMP device offloading can be used to accelerate automotive workloads on embedded heterogeneous platforms, and how application performance can further be improved by adapting the OpenMP runtime to the special features of embedded systems and by using advanced OpenMP mechanisms, such as platform-specialized function variants.

The optimized OpenMP offloading implementations developed in this work are available in the public DAPHNE source code repository on Github[1]. The modified version of the LLVM infrastructure is also publicly available on Github[2].

In the future, we plan to further improve the efficiency of the OpenMP runtime on embedded platforms and investigate the use of OpenMP offloading for other embedded and automotive use-cases.

REFERENCES

[1] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, et al. "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems." In: *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. ICCPS '18. 2018.

[2] Nvidia Inc. *CUDA for Tegra - Application Note*. https://docs.nvidia.com/cuda/pdf/CUDA-for-Tegra-AppNote.pdf. Accessed: 2020-06-03. 2019.

[3] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. "DAPHNE - Automotive Benchmark Suite for Parallel Programming Models on Embedded Heterogeneous Platforms - Work-in-progress." In: *Proceedings of the International Conference on Embedded Software*. EMSOFT '19. New York, NY, USA: IEEE Press, 2019.

1 https://github.com/esa-tu-darmstadt/daphne-benchmark
2 https://github.com/sommerlukas/llvm-offload-jetson

[4] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. "Using Parallel Programming Models for Automotive Workloads on Heterogeneous Systems - a Case Study." In: *28th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'20)*. 2020.

Part III

OPENMP DEVICE OFFLOADING FOR FPGAS

11 OPENMP DEVICE OFFLOADING TO FPGA ACCELERATORS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *OpenMP Device Offloading to FPGA Accelerators* by *Lukas Sommer, Jens Korinth and Andreas Koch* in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. The contribution of the author of this thesis is summarized as follows.

» As the corresponding and leading author, Lukas Sommer was responsible for the conception and implementation of the compiler frontend extension based on Clang, and the realization of the compile flow, including the interface to the Xilinx Vivado HLS compiler. With the assistance of Jens Korinth, Lukas Sommer designed and implemented the LLVM OpenMP runtime plug-in for offloading to FPGAs, based on the software API of Threadpoolcomposer. The experiments on the FPGA and the CPU were planned and conducted by Lukas Sommer. The text was joint work by Jens Korinth (section 3) and Lukas Sommer (remaining sections), with feedback from Andreas Koch. «

ABSTRACT

Future high-performance computing systems will need to include multiple specialized accelerators in a single heterogeneous system to overcome power-density limitations of CPU performance.

To program such heterogeneous systems without the need to maintain multiple code bases, OpenMP device offloading constructs can be used to execute compute-intensive regions on different kinds of accelerators.

In this work we present a proof-of-concept implementation of OpenMP offloading for FPGA-based hardware accelerators. Our implementation seamlessly integrates with the existing LLVM offloading infrastructure, and enables the user to move computations to a custom FPGA accelerator by simply adding OpenMP offloading directives to the input program.

11.1 INTRODUCTION

With new process technologies for CPUs no longer translating into performance improvements due to power density limitations, the performance increase for CPUs has declined in recent years. As a consequence, CPU-only systems are no longer able to meet the ever-increasing demands for computing power, especially in high-performance computing (HPC) scenarios.

In order to overcome these limitations and provide sufficient computational power for future computing tasks, future high-performance computing systems will need to incorporate multiple dedicated, specialized accelerators into a single heterogeneous system. Each accelerator is suitable for a limited set of operational tasks and is able to deliver a better power-efficiency for this set of tasks than the general-purpose CPU.

Beyond the GPUs that are nowadays very common in high-performance heterogeneous systems, dedicated FPGA-based hardware accelerators have received increasing attention recently. Their reconfigurability facilitates the adaptation of the accelerator to multiple tasks, delivering better performance and power-efficiency than general-purpose processors. An example for the use of FPGAs in heterogeneous systems is the deployment of FPGAs in Microsoft's Azure cloud [4].

However, programming heterogeneous systems is hard, as each system comprises multiple (potentially varying) different computational units. Adapting the software to each accelerator and system requires significant rewriting of existing code bases, resulting in high development effort and cost.

The use of a single input program for all kinds of accelerators, on the other hand, is also challenging. This is especially true as not all accelerators are suitable for all computational tasks, and typically only the most computation-intensive sections ("hot-spots") of a program should be offloaded to a dedicated device, whereas more sequential or unsuitable (e.g. control-flow intensive) regions of code should remain on the general-purpose CPU.

The OpenMP target directive, introduced and refined in recent versions of the standard [13], is a perfect fit for denoting regions of code that should be offloaded to a device in a heterogeneous system. The directive does not only allow to specify the device-suitable regions of code by easy-to-use pragmas, but the associated data mapping clauses also give the programmer full control over what and how data is mapped to the device. Therefore, the OpenMP target directive is a good choice for the programming of heterogeneous systems including an FPGA as dedicated hardware accelerator.

The LLVM compiler framework C/C++-language frontend, Clang, is currently being extended for target directives. In this work, we build on the existing Clang infrastructure to provide support for offloading to FPGA-based accelerators using OpenMP target directives. Our work is based on *ThreadPoolComposer (TPC)* [10], an automated framework for the synthesis and HW/SW interfacing of FPGA-based accelerators (available from [19]). It uses Xilinx Vivado HLS for generating hardware accelerators from C/C++. Our implementation enables the user to directly offload OpenMP target regions that are compatible with the input restrictions of Vivado HLS to FPGA hardware accelerators. The user is not required to provide specialized code in the input program to interface with the hardware accelerator and our implementation also manages data mapping to the FPGA memory.

The rest of this work is structured as follows. Section 11.2 gives an overview of related work and the existing OpenMP offloading infrastructure in Clang/LLVM. Section 11.3 provides a short introduction to the ThreadPoolComposer toolchain, on which our work is based. In Section 11.4 we describe our new compile and runtime flow, which we evaluate and compare in Section 11.5. Section 11.6 concludes our work and gives an overview on future work.

11.2 PRIOR WORK

We will begin the discussion of related work by looking at other tools which use OpenMP as input for High-Level Synthesis targeting FPGAs and compare them to our approach. Afterwards, we describe some efforts to implement OpenMP device offloading to non-FPGA targets and explain the existing LLVM offloading infrastructure in more detail.

11.2.1 *OpenMP-based FPGA-acceleration*

OpenMP-annotated source code has been used as input and starting point for a number of High-Level Synthesis approaches targeting FPGAs. These approaches focus on efficiently mapping *parallel* OpenMP-constructs to FPGA-based hardware accelerators, either using a pure hardware-flow or aiming for a mixed software/hardware-environment. For example, in [15, 16] a dedicated accelerator is synthesized for every OpenMP task in the program. Other efforts, such as the ones presented in [6–8, 18], efficiently map OpenMP worksharing loops to FPGA-accelerators capable of executing computations from multiple threads in parallel.

However, none of these approaches makes use of the OpenMP *target* directives to allow the user to specify which regions to offload to the FPGA, or how to map data from the host to the FPGA.

In contrast, the approach presented in this work uses the OpenMP *target* directive to allow programming a heterogeneous system, including FPGA-based accelerators, with a single, portable input code. The focus of our work is the offloading itself, i.e. the management of the on-device execution and the efficient mapping of data from host- to device-memory.

Using the memory mapping specified by the user in the appropriate pragma allows our tool to clearly determine which data must be transferred to/from the device memory, whereas the previously discussed approaches must employ a conservative approximation and tend to transfer more data than strictly necessary.

For the mapping of the code inside the target region to FPGA-accelerators, we currently rely on Vivado HLS. However, note that we could also integrate other HLS-approaches (e.g., those already listed above or other generic HLS systems such as Nymble [9], Bambu [14] etc.), to efficiently map the code inside the target region, potentially containing parallel OpenMP constructs, to the FPGA. Thus, our approach can be seen as complementary to the ones discussed above in that it can *additionally* provide the user with clearly defined means to specify a memory mapping and denote regions that are to be executed as FPGA-accelerators.

11.2.2 *OpenMP Device Offloading*

OpenMP device offloading has previously been implemented within the LLVM infrastructure for a number of device types. [2] presents results for OpenMP execution in a system featuring a Xeon Phi accelerator. In [11], an implementation of OpenMP target offloading for DSP accelerators is described. The target architecture comprises of a multi-core, general-purpose ARM CPU and a multi-core DSP. The volume of data transferred is optimized by allocating a contiguous block of physical memory that is shared between host and device, making the transfer to/from the device obsolete.

In [3], Bertolli et al. present an extension of the Clang language frontend and the LLVM OpenMP runtime, which facilitates the use of CUDA-enabled Nvidia GPUs for OpenMP offloading. Code regions intended for offloading are automatically translated to CUDA ptxas assembly, with OpenMP parallel constructs transformed to parallel CUDA constructs. The runtime uses the CUDA device driver to map data to/from the device and to initiate computation on the GPU.

Finally, [1] presents a concerted effort for generic and extensible support of OpenMP device offloading within the Clang/LLVM compiler infrastructure. The implementation is designed to provide easy access to common functionality, and to be extended for further device types with limited implementation effort.

During the compilation phase, a separate device-specific translation is performed. The resulting binaries for all devices and the host are then combined to a single "fat" binary.

Additionally, the LLVM OpenMP runtime has been extended by two-layered library support for device offloading, with the design of the library described in [17]. The device-agnostic libomptarget provides common functionality and offers a standardized interface for data mapping and device execution control. To this end, libomptarget interacts with the device-specific plugins on the second layer of the library, which each provide support for offloading for a certain kind of device (e.g. CUDA-enabled GPUs).

Our own work integrates seamlessly with this LLVM offloading infrastructure. In Section 11.4 we describe our custom Clang compilation flow and the implementation of our libomptarget device plugin.

11.3 THREADPOOLCOMPOSER

The ThreadPoolComposer [10] toolchain has been developed to fast-track the prototyping of FPGA-based accelerators using HLS tools, such as Vivado HLS. TPC automates the execution of the HLS tool to synthesize hardware accelerators from kernel code, and can assemble multiple instances of these accelerators (*processing elements, PEs*) in a complete top-level design, called *threadpool*, which also provides standardized connectivity to host and device memory. Regardless of its internal composition, every threadpool can be controlled by a two-layered, unified software interface, consisting of the user-facing *TPC API* and the internal *Platform API*. TPC API provides basic functions to query the device bitstream for available hardware modules, transfer data to/from the device and launch jobs on the threadpool. Platform API is only used to implement the TPC API and isolates platform-specific code (e.g., to control infrastructure hardware). This facilitates basic portability of TPC applications since the TPC API is identical for all hardware platforms (write once, run "everywhere"). Using TPC designs, the software programmer can thus take advantage of the specific capabilities of the executing platform, without having to rewrite any code. This makes TPC an ideal low-level foundation for the implementation of higher-level, heterogeneous, parallel runtimes and programming frameworks, such as OpenCL, or OpenMP.

In this work, we substantiate this claim by combining TPC and the OpenMP target offloading infrastructure to implement FPGA-based accelerators directly from OpenMP application code. Our custom Clang toolflow (described in more detail in the next section) extracts standalone C/C++ functions from the OpenMP application code (marked with the target directive), which are then fed into the TPC toolchain to generate a hardware design. Our LLVM offloading plugin automatically generates the TPC API calls to interface with the hardware, enabling transparent offloading to the FPGA.

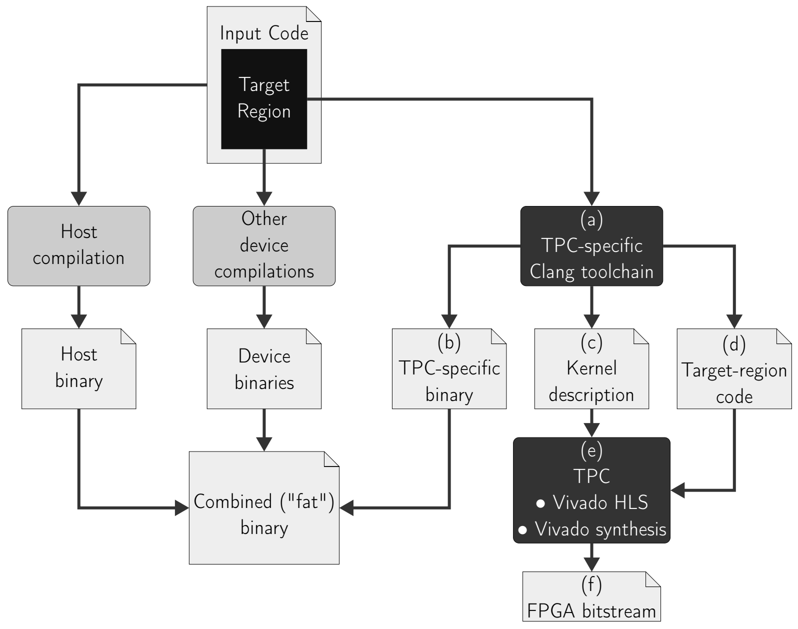


Figure 11.1: Custom device toolflow for Clang.

11.4 OFFLOADING FOR FPGAS

In this section we describe the integration of our work into the Clang/LLVM OpenMP offloading infrastructure. As stated in Section 11.2, our implementation consists of two parts, namely a custom device toolflow for Clang and a device-specific plugin implementation for libomptarget, both described in detail in the following sections.

11.4.1 *Compilation Flow*

As indicated in Section 11.2, the compilation of an input program containing one or more OpenMP target directives results in multiple calls to Clang toolflows. In addition to the regular invocation for the host compilation, a distinct toolchain is invoked for each offloading target selected in the original call to the Clang driver. In contrast to the host compilation, the scope of per-device compilation is limited to the target regions present in the input program, which have each been extracted to a separate function before.

In order to support FPGA-based OpenMP offloading, we implement a custom Clang toolflow (see (a) in Fig. 11.1), which is identified by a custom LLVM target-triple we introduced. The output of our custom toolflow consists of three parts:

- A TPC-specific device software executable (Fig. 11.1.b), which is included in the fat binary.
- An input file for Vivado HLS for each target region in the original program (Fig. 11.1.d).

- A kernel description (Fig. 11.1.c) for each target region, used to drive the TPC synthesis flow.

These output files are now described in greater detail.
TPC device software executable The device software executable is the result of our custom code generation. It directly interacts with TPC using functions from the TPC API (cf. Section 11.3). Its tasks are the transfer of parameter values (e.g. pointers to arrays used within the target region) to the FPGA, and the launch of the accelerator execution after a job ID has been acquired. The additional level of indirection introduced by this software executable (compared to direct invocation of TPC from the libomptarget-plugin) is beneficial, as it allows us to implement a more fine-grained control of the interfacing between software and hardware. For example, we could implement coarse-grain parallelism in the software executable, distributing the computations of an OpenMP worksharing loops across multiple hardware processing elements by acquiring and simultaneously launching *multiple* offloaded jobs in the software executable.
Vivado HLS input file The Vivado HLS input files resulting from our custom device compilation toolchain each contain the code for a single target region of the original program, extracted to a function. Our toolchain also preserves pragmas unknown to Clang in this regions, allowing Vivado HLS specific pragmas annotated by the user to be still present in the Vivado HLS input file. These pragmas, indicating e.g., pipelining or unrolling of a loop, could also be added automatically in the future.
Kernel description The TPC-specific kernel description identifies the Vivado HLS input file and the target function for each target region, and also lists the type (*value* or *reference*) for each parameter of the target function.

From here on, TPC automates the entire design flow: Using the kernel description and the target region code source file, TPC synthesizes a hardware module for each target region in the input program via Vivado HLS. TPC then automatically assembles a complete top-level design with host connectivity and memory access. The top-level can contain multiple hardware instances of an individual kernel (so-called *processing elements*), and also mix hardware instances of different kernels (from distinct target regions), avoiding the need to dynamically reconfigure the FPGA at runtime. Finally, the hardware design is synthesized using the Vivado Design Suite. The resulting bitstream can be directly loaded and accessed with the TPC APIs (see Fig. 11.1.f).

In summary, our custom compilation flow allows users to go from a single input code, with OpenMP target directives annotated in the program code, to a complete FPGA-design including memory- and host-connectivity, without the need for the user to provide additional low-level specifications to the flow.

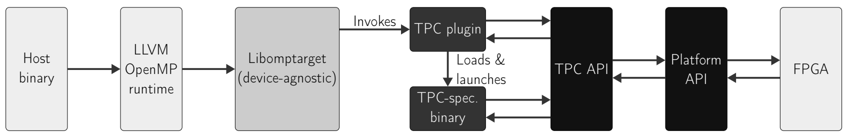


Figure 11.2: Runtime flow for TPC FPGA offloading.

11.4.2 *Runtime flow*

The OpenMP offloading model is a host-centric approach, i.e., the execution starts on the host CPU. If a target region is encountered, device execution is initiated by calling libomptarget using calls from the LLVM OpenMP runtime library interface. Should this be the first offload to a device, the device is initialized now. Before the execution on the device can start, data needs to be made available on the device. The OpenMP standard allows data to be shared between host and device, as well as for separate data spaces. In our current implementation, FPGA and host CPU do not share memory, therefore we need to allocate and transfer mapped data to the FPGA memory by invoking the TPC device plugin. During this process, libomptarget is responsible for keeping track of the mapping between host and device pointers for each variable mapped to the device, and the TPC-specific device plugin uses data transfer functions from the TPC API to initiate data transfers with the DMA engine in the FPGA bitstream.

In the next step, the TPC device-specific software executable is loaded and launched using libelf and libffi, a process similar to the one used for offloading to ELF-compatible devices in general. The loaded software executable in turn starts hardware execution (cf. Section 11.4.1). After hardware execution completes, data is copied back to the host memory, again using TPC API calls and the DMA engine on the FPGA. The entire runtime execution flow is shown in Fig. 11.2.

11.5 EVALUATION

For this initial proof-of-concept system, we use the Xilinx VC709 board as accelerator. Here, an XC7VX690T FPGA, attached to 4 GiB on-board memory, is connected to the host via PCIe Gen3 x8. This platform uses the lightweight *ffLink PCIe Gen3 interface* which achieves close-to-optimal data transfer speeds [5]. The host uses a four core Intel Core-i7-6700K at 4.0 GHz with 16 GiB of DDR4 RAM running a Fedora 22 Linux with kernel 4.0.8.

We evaluate our approach using vector- and matrix-routines from the Adept benchmark suite [12]. For each workload we create two different hardware kernels: The first without any optimisations, the second with loop pipelining for the innermost loop by manually inserting the loop pipelining pragma available in Vivado HLS. All kernels run at a frequency of 250 MHz and the pipelined kernels

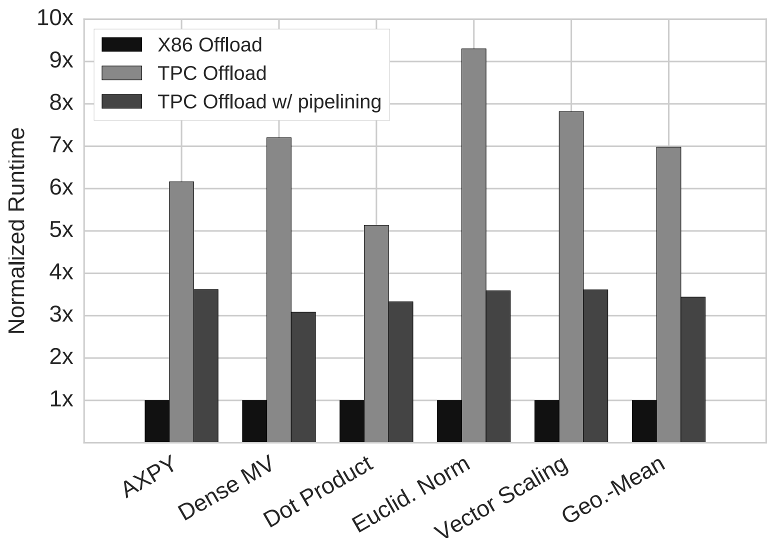


Figure 11.3: Normalized runtime.

were scheduled with an *initiation interval* of 1. For all benchmarks, we were able to offload the workload loop to the FPGA by just adding a simple OpenMP target pragma, demonstrating both the functionality and convenience of our implementation.

We compare our implementation to the x86-offloading implementation present in LLVM. The x86-executables were compiled with -O3 and use 4 cores for the OpenMP parallelised workload in each benchmark. Runtimes (normalized to the x86 offloading execution) are shown in Fig. 11.3. For the current prototype, the offloaded FPGA execution is slower (geomean ∼6.9x and ∼3.4x), with pipelining significantly speeding up the hardware execution (geomean ∼2x improvement over non-optimised kernel). However, bear in mind that the x86 offloading execution uses 4 core CPU at 4.0 GHz, whereas our current proof-of-concept implementation is still limited to a single processing element at 250 MHz.

Despite being slower than the CPUs, the prototype is actually useful to evaluate the *overhead* of our offloading approach compared to the x86 offloading implementation. We measure the runtime for different data sizes, ranging from 0.5...50 MiB of the input vector in the vector scaling benchmark. The bold black line in Fig. 11.4 depicts the unit diagonal between input data size and runtime (note the logscale!). While the overhead of offloading dominates for all three targets in the case of 0.5 MiB, all targets exhibit a gradient <1 for larger data sizes. Considering the fact that the overhead of our offloading implementation includes hardware initialization, data transfers via PCIe, and interrupt latencies, we conclude that our offloading approach via TPC is competitive.

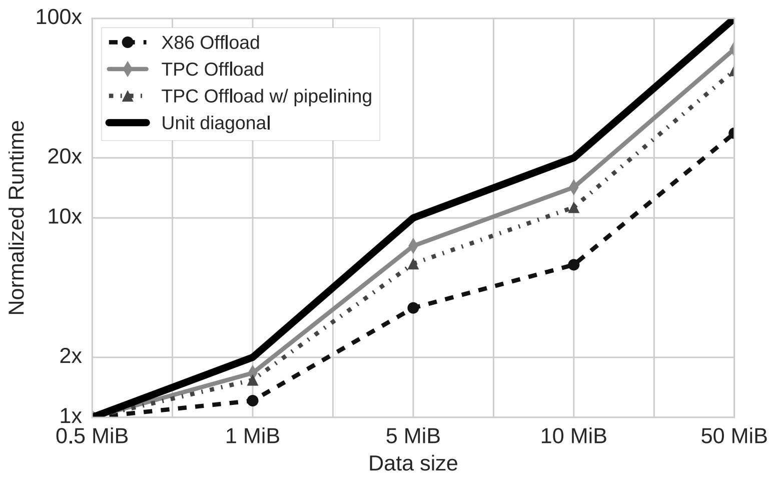


Figure 11.4: Normalized runtime vs. data size in the *vector scaling* benchmark.

11.6 CONCLUSION AND FUTURE WORK

In this work, we have presented the first fully functional implementation of the OpenMP device offloading model for FPGAs. Our implementation seamlessly integrates with the existing LLVM offloading infrastructure, and enables users to move computational workloads to a custom FPGA accelerator by simply adding OpenMP *target* directives to their code. Using our tool flow, which combines custom Clang extensions and TPC's automation of the hardware synthesis process, the user can generate a complete FPGA-design, including memory- and host-connectivity, from a single, portable input code. Furthermore, the OpenMP memory mapping clauses allow the user to precisely specify which data to transfer to/from the device memory. Our evaluation then showed that our approach does not introduce excessive overhead to the offloading process.

In our current implementation, a single PE occupies less than 1% of the FPGA's resources (not including PCIe, memory and host connectivity infrastructure). Therefore, in order to improve the performance of FPGA offloading, we intend to make use of coarse-grain parallelism in future work by automatically distributing the computation of parallel OpenMP workloads, such as `target team distribute`, across multiple identical PEs.

Beyond that, we intend to extend our implementation to additional platforms supported by TPC, e.g., the Xilinx *Zynq-series* reconfigurable SoC systems.

REFERENCES

[1] Samuel F Antao, Alexey Bataev, Arpith C Jacob, Gheorghe-Teodor Bercea, Alexandre E Eichenberger, Georgios Rokos, Matt

Martineau, Tian Jin, Guray Ozen, Zehra Sura, et al. "Offloading Support for OpenMP in Clang and LLVM." In: *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*. IEEE Press. 2016, pp. 1–11.

[2] James Barker and Josh Bowden. "Manycore parallelism through OpenMP." In: *International Workshop on OpenMP*. Springer. 2013, pp. 45–57.

[3] Carlo Bertolli, Samuel F Antao, Gheorghe-Teodor Bercea, Arpith C Jacob, Alexandre E Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos, David Appelhans, et al. "Integrating GPU support for OpenMP offloading directives into Clang." In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM. 2015, p. 5.

[4] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. "A Cloud-Scale Acceleration Architecture." In: *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Oct. 2016.

[5] David de la Chevallerie, Jens Korinth, and Andreas Koch. "ffLink: A Lightweight High-Performance Open-Source PCI Express Gen3 Interface for Reconfigurable Accelerators." In: *Highly Efficient Accelerators and Reconfigurable Technologies (HEART), International Symposium on* (2015).

[6] Jongsok Choi, S. Brown, and J. Anderson. "From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs." In: *2013 International Conference on Field-Programmable Technology (FPT)*. Dec. 2013, pp. 270–277. DOI: 10.1109/FPT.2013.6718365.

[7] Alessandro Cilardo, Luca Gallo, Antonino Mazzeo, and Nicola Mazzocca. "Efficient and Scalable OpenMP-Based System-Level Design." In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*. Mar. 2013, pp. 988–991. DOI: 10.7873/DATE.2013.206.

[8] Alessandro Cilardo, Luca Gallo, and Nicola Mazzocca. "Design Space Exploration for High-Level Synthesis of Multi-Threaded Applications." In: *Journal of Systems Architecture* 59.10, Part D (Nov. 2013), pp. 1171–1183. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2013.08.005. (Visited on 12/30/2015).

[9] Jens Huthmann, Björn Liebig, Julian Oppermann, and Andreas Koch. "Hardware/software co-compilation with the nymble system." In: *Reconfigurable and Communication-Centric Systems-on-*

Chip (ReCoSoC), 2013 8th International Workshop on. IEEE. 2013, pp. 1–8.

[10] J. Korinth, D. d. l. Chevallerie, and A. Koch. "An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures." In: *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. May 2015, pp. 195–198. DOI: 10.1109/FCCM.2015.22.

[11] Gaurav Mitra, Eric Stotzer, Ajay Jayaraj, and Alistair P Rendell. "Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture." In: *International Workshop on OpenMP*. Springer. 2014, pp. 202–214.

[12] Nick Johnson, Michèle Weiland, Trevor Carlson, and Sudarshan Balaji. *The Adept Benchmark Suite*. 2015. URL: http://www.adept-project.eu/images/whitepapers/p_590591_1446568406_Adept_Whitepaper_Benchmarks.pdf.

[13] OpenMP Architecture Review Board., ed. *OpenMP Application Programming Interface - OpenMP Standard 4.5*. Nov. 2015.

[14] Christian Pilato and Fabrizio Ferrandi. "Bambu: A Free Framework for the High-Level Synthesis of Complex Applications." In: *University Booth of DATE* (2012).

[15] A. Podobas. "Accelerating Parallel Computations with OpenMP-Driven System-on-Chip Generation for FPGAs." In: *2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*. Sept. 2014, pp. 149–156. DOI: 10.1109/MCSoC.2014.30.

[16] A. Podobas and M. Brorsson. "Empowering OpenMP with automatically generated hardware." In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. July 2016, pp. 245–252.

[17] Samuel Antao, Carlo Bertolli, Andrey Bokhanko, Alexandre Eichenberger, Hal Finkel, Sergey Ostanevich, Eric Stotzer, and Guansong Zhang. *OpenMP offload infrastructure in LLVM*. URL: https://github.com/clang-omp/OffloadingDesign.

[18] L. Sommer, J. Oppermann, and A. Koch. "C-based Synthesis of Area-Efficient Accelerators for OpenMP Worksharing Loops." In: *Second International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'16)* (2016).

[19] *ThreadPoolComposer*. https://git.esa.informatik.tu-darmstadt.de/REPARA/threadpoolcomposer.git. Accessed: 01-04-2017.

12 SYNTHESIS OF INTERLEAVED MULTITHREADED ACCELERATORS FROM OPENMP LOOPS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *Synthesis of Interleaved Multithreaded Accelerators from OpenMP Loops* by *Lukas Sommer, Julian Oppermann, Jaco Hofmann and Andreas Koch* in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. The contribution of the author of this thesis is summarized as follows.

> *» As the corresponding and leading author, Lukas Sommer was responsible for the design of the multi-threaded execution model and the implementation of the corresponding hardware components, as well as for the realization of the new hardware backend of the Nymble HLS compiler. He was also in charge of the extension of Nymble's frontend to accept OpenMP worksharing loops as input to the high-level synthesis flow and the integration with LLVM's OpenMP runtime. The wrapping of the generated processing elements for simulation and integration with Threadpoolcomposer was contributed by Jaco Hofmann. The experiments on the FPGA were planned and conducted by Lukas Sommer. He was also responsible for the manuscript's text, with feedback from Julian Oppermann and Andreas Koch. «*

ABSTRACT

Similarly to CPUs and GPUs, FPGA-based accelerators can also profit from exploiting thread-level parallelism. Thus, the synthesis tools for generating the circuits from high-level languages need to be extended appropriately.

We present an extension of the Nymble hardware/software-co-compiler for the automatic synthesis of hardware accelerators from OpenMP worksharing loops, and describe modifications to the datapath- and memory-architecture for multi-threaded execution.

The new execution model employs both spatial as well as thread-level parallelism in the microarchitecture of the generated accelerator, with the aim to efficiently hide memory access latencies.

We are able to gain raw speedups of more than a factor of 3x, and improve the utilization of the computing unit by more than factor 8x, when executing four threads instead of a single one on the computing units.

12.1 INTRODUCTION

Recently, FPGAs are increasingly employed in large datacenters as an alternative to GPUs as dedicated hardware accelerators. An example for the use of FPGAs in high-performance computing (HPC) scenarios is Microsoft's Bing and Azure Cloud [3], where all compute nodes are equipped with FPGAs for compute and high-speed network-processing tasks. FPGA-based acceleration has also been used successfully for high-speed network security monitoring [13].

High-level synthesis (HLS) of accelerator designs from high-level language input programs is a powerful method to automatically implement FPGA-based hardware accelerators. As of today, HLS-tools mainly gain their speedup over a sequential execution on a CPU by exploiting the instruction-level parallelism (ILP) present in the input programs. However, the instruction-level parallelism contained in applications is typically limited, e.g., by data-dependencies between operations or by control-flow.

For the successful employment of HLS-generated FPGA-designs in HPC-scenarios, the limited amount of speedup resulting from the exploitation of instruction-level parallelism is not sufficient. This is especially true in light of today's FPGA sizes, which provide an increasing amount of resources to the user.

To make full use of the available resources and in order to make HLS-based FPGA-designs more suitable for use in high-performance computing, HLS-tools recently started to make use of thread-level parallelism (TLP). Yet, the automatic parallelization and extraction of thread-level parallelism from input programs is difficult at best and impossible for a large range of problems.

User-annotations guiding the compiler in the parallelization and extraction of TLP from the input program are a possible and powerful alternative. A popular standard for such user-annotations is OpenMP [15]. The use of OpenMP directives for HLS allows to use existing programs as input, without the need for the insertion of special hardware annotation by the user. Furthermore, OpenMP's shared memory computation model is perfectly suited for FPGA-based hardware accelerators, because their architecture often combines the FPGA with an external memory, to which all threads running on the FPGA have access.

From the microarchitecture perspective, TLP can be exploited by carrying out each thread's computations on a dedicated hardware computing unit. However, this solution leaves plenty of room for improvement: Due to the latency caused by accesses to main memory, the computing unit will be idle a significant portion of the execution time, leaving the underlying hardware resources unused. By *interleaving* the execution of multiple threads on the computing units, these latencies can efficiently be bridged with other computations, improving the hardware resource utilization. Therefore it is the aim of this work to extend the existing Nymble hardware/software-co-compiler to synthesize hardware accelerators comprising computing units, which are able to interleave the execution of multiple threads in a time-sliced manner on the same FPGA area.

The rest of this work is structured as follows. Section 12.2 gives an overview of the existing approaches for OpenMP-based HLS. In Section 12.3 we describe the Nymble hardware-software-co-compiler, on which this work is based. Section 12.4 presents our novel hardware execution model and explains all necessary modifications to the hardware datapath- and memory-architecture. In Section 12.5 we evaluate our approach, Section 12.6 concludes and gives an outlook to future work.

12.2 RELATED WORK

There have been a number of approaches to integrate OpenMP into High-Level Synthesis, ranging from mere source-to-source transformations to complex hardware execution models featuring direct access to memory from the accelerator. Similar to our approach, some of these approaches are based on OpenMP worksharing loops, indicated by `#pragma omp parallel for` in the input program.

Leow et al. [12] as well as Dziurzanski et al. [7] pursue a pure hardware approach and need to emulate shared memory by registers (Leow) and global signals (Dziurzanski), respectively, because their hardware implementation does not include a memory system with access to shared memory.

In their work, Cilardo et al. [5, 6] use an OpenMP-annotated input program as starting point for the synthesis of specialized hardware accelerators in a System-on-chip, combined with general-purpose processors. All elements of the resulting MPSoC have full access to the memory via the communication network of the SoC.

The integration of OpenMP support into the LegUp high-level synthesis system [2] by Choi et al. [4] features multiple levels of parallelism in hardware. The resulting system is composed of a MIPS processor and multiple hardware accelerators, which have direct access to memory via the interconnect.

All of the approaches presented above exploit thread-level parallelism with the concurrent execution on multiple hardware units. The set of iterations is partitioned onto multiple identical hardware kernels, which each conduct the computations of a single thread. However, in none of the these approaches multiple threads are executed on the same hardware datapath in an interleaved or concurrent fashion. Each distinct datapath is only running a single thread at a time and this 1:1-relationship between computing units and threads is maintained throughout the execution time. Only Choi et al. are able to pipeline execution of multiple threads in a limited number of cases. Our approach, in contrast, is intended to exploit thread-level parallelism in multiple ways, by distributing the computation across multiple computing units and, at the same time, running multiple threads on each computing unit in an interleaved manner.

Concurrent execution of multiple threads has been used in existing approaches [8, 10, 16]. In contrast to our work, these approaches do not use OpenMP as a starting point for high-level synthesis. The thread-level parallelism originates from threads explicitly managed by the user or from multiple instances of the same program running in distinct processes and requires significant effort from the user. Contrary to these approaches, our work allows users to incrementally modify the program by inserting OpenMP-pragmas, which manage thread-level parallelism. Moreover, our support for OpenMP provides the user with a clearly defined shared memory model and synchronization mechanisms.

Both approaches also integrate thread pipelining, which we do not use in our current implementation.

12.3 NYMBLE HW/SW-CO-COMPILER

Our work is an extension of the Nymble hardware/software-co-compiler [9], which aims for the automatic synthesis of a large subset of `C` (including, e.g., pointer operations and irregular control flow in loops) for so-called *Adaptive Computer Systems (ACS)*, comprising an FPGA as *reconfigurable computing unit (RCU)* and a general-purpose processor.

Nymble allows the extraction of arbitrary sections of code, usually delimited by `pragmas` for synthesis, to a dedicated hardware accelerator. Furthermore, it also automatically provides all means necessary to interface between the software running on the GPP and the hardware accelerator.

For the high-level synthesis of the extracted hardware part in the Nymble compiler, the `pragma`-denoted section of the input program is represented as a hierarchical tree of so-called *control-memory-data-flow-graphs (CMDFG)*, with one CMDFG per loop in the hardware code region. A CMDFG does not only model a program's data flow between operations, but also incorporates the control flow of the input code. To this end, the control flow in the CMDFG is represented by conditional data flow and predicates, which are added to operations with side-effects (e.g., memory accesses). The tree of graphs contains one CMDFG per natural loop in the input program and reflects the nesting hierarchy of the loops in the program. During the execution of an inner loop, the graph of the surrounding loop does not continue its execution.

The work presented in [8, 10], called *Nymble-SMT*, is also based on the Nymble compilation framework. This approach uses loop-pipelining with a dynamically varying initiation interval. To ensure the minimal latency required to maintain loop-carried dependencies between iterations from the same threads, this implementation uses complex backpressure logic with tokens. In addition, dynamic operator multiplexing is required to resolve concurrent use of shared operators from different iterations of the same thread. As a consequence, the relative costs for the implementation of the Nymble-SMT multithreading model can be very high, especially for integer benchmarks which use only relatively simple operators. In this work we extend Nymble with an *alternative* multithreaded execution model (Section 12.4.1) and the corresponding controller implementation. To support thread-switching in the computing units, we augment Nymble's CMDFG-model with thread-context stores and implement an algorithm that inserts context stores only where necessary (Section 12.4.2). More details on the basic mapping from CMDFG to hardware datapath and the hardware/software-interface can be found in [9].

In contrast to Nymble-SMT, our model does not use loop pipelining to reduce the cost and complexity of the controller implementation. In order to support loop pipelining with a static II in our multithreaded model, we would need to add logic to synchronize stalls across all CDFGs of the graph-hierarchy. Besides that, the size of a thread's context, i.e. the elements that need to be stored and restored upon stall and reactivation, increases when applying loop pipelining.

Being based on OpenMP, our implementation incurs less overhead in the hardware/software-interface, as a single data-transfer and invocation is sufficient, compared to Nymble-SMT where each thread has to go through the HW/SW-interface. Contrary to Nymble-SMT, our approach supports the duplication of datapaths, which allows for the concurrent execution of multiple threads on distinct resources. Our approach employs a fair round-robin hardware thread scheduler, whereas Nymble-SMT uses a static prioritization scheme which can lead to the starvation of threads. Furthermore, we implemented a new cache- and memory-infrastructure which allows for shared memory between threads and higher operation frequencies of the kernels compared to Nymble-SMT.

12.4 NYMBLE-OMP

The aim of this work is to extend the existing implementation of Nymble to support the synthesis of *multithreaded* FPGA-accelerators based on OpenMP input programs. We therefore need to add support for processing of loop-nests marked as OpenMP worksharing loops by the `pragma omp parallel for` to Nymble. To this end, we implement the detection and extraction of OpenMP worksharing loops in

```
float a;
float x [SIZE];
float y [SIZE];

#pragma omp parallel for schedule(static)\
shared(a, x, y) private(i)\
num_threads(NUM_THREADS)
for(i=0; i<SIZE; i++){
    y[i] = a * x[i] + y[i];
}
```

Figure 12.1: Example of an OpenMP worksharing loop.

Nymble's frontend. Nymble-OMP will then construct a hierarchical tree of CMDFGs for all loops in the annotated loop-nest.

The execution on the hardware datapaths synthesized from the resulting set of CMDFGs follows a novel execution model. The new execution model facilitates the interleaved execution of multiple threads on each hardware datapath, and the distribution of the input problem among multiple identical computing units running concurrently.

The following sections describe our novel execution model and necessary modifications to Nymble's datapath architecture in detail.

12.4.1 *Execution Model*

The basic idea of OpenMP worksharing loops, such as the one shown in the code snippet in Fig. 12.1, is to distribute the work specified by the loop body across a team of threads. From the set of iterations of the annotated loop, each thread is assigned a subset, whose size is dependent on the chosen distribution scheme.

On a typical multi-core CPU, the threads in the team will be distributed across the cores present in the CPU, where the execution of multiple threads will be interleaved in a time-sliced manner in case the number of threads specified by the user exceeds the number of cores.

In contrast, the existing OpenMP-based approaches presented in Section 12.2 synthesize a dedicated hardware computing unit per thread and carry out each thread's computation on distinct hardware resources without interleaving. The advantage of this model is that a thread can always be executed if it is ready to do so.

However, not all operations' latencies can be determined statically during the HLS-scheduling process, which is especially true for cached memory accesses that can cause a significant delay in case of a cache-miss. We refer to this kind of operation as *variable-latency operations* (VLO).

If at runtime a variable-latency operation does not finish within its assumed latency, this results in a stall of the hardware, i.e., the

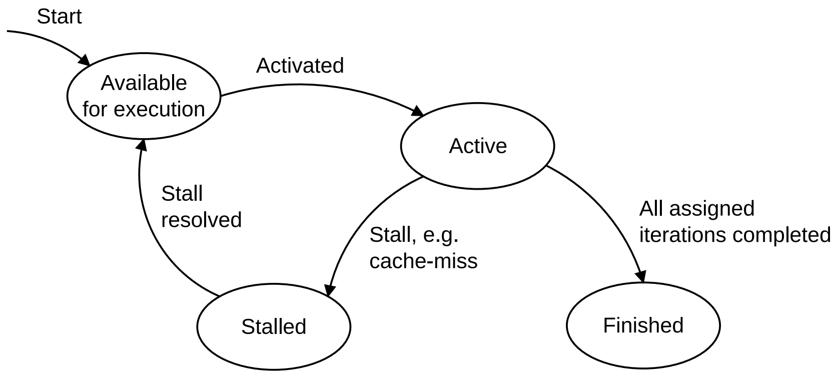


Figure 12.2: Thread state diagram.

computation on the hardware computing unit is stopped until the VLO completes.

As a consequence, the employed hardware resources are not utilized to their full extent as they remain idle for a significant fraction of the execution time, e.g., if a data item must be retrieved from main memory. With a more efficient utilization of the hardware resources one could either achieve the same runtime performance with fewer hardware resources or improve the performance of the system with the same number of hardware resources.

Thus, the central goal of our execution model is to hide statically unpredictable latencies, such as latencies caused by cached memory accesses, with the execution of another thread on the same hardware computing unit.

In case the executing thread hits a stall, i.e., if the execution of a datapath operation takes longer than statically assumed for HLS-scheduling, we perform a thread-switch, replacing the currently active thread by some other available, execution-ready thread.

In order to be able to perform a thread-switch, we need to remove the 1:1-relationship between threads and computing units, as used by existing approaches. Instead of mapping each thread in the execution context to a dedicated computing unit, we assign a *set of threads* to each computing unit. The execution of all threads assigned to each computing unit is then interleaved in a time-sliced manner. In doing so, a thread-switch is only performed in case the currently executing thread encounters a stall.

At runtime, a thread can be in one of four different states, shown in Fig. 12.2, in our execution model. The single thread in state *Active* executes until it hits a stall, causing the state transition to state *Stalled*. As a consequence, one of the available threads is chosen as next thread to be executed and activated. We currently use a round-robin scheme to select the next thread from the set of threads in state *Available for execution*, but many other selection schemes, such as techniques used for hardware-only schedulers in GPUs [11], are conceivable and could be implemented in our controller architecture.

At the same time, the execution of the variable-latency operation continues in the background, until it is completed and the previously stalled thread becomes available again.

The cycle of execution, stall, and reactivation continues for each thread until the thread eventually completes all iterations it was assigned at the beginning of the hardware execution and changes to state *Finished*.

12.4.2 *Datapath architecture and composition*

Our multithreaded execution model requires the hardware datapaths in the computing unit to be able to store the thread contexts of all non-active threads, i.e. all threads stalled or available for execution, which have been assigned to that computing unit.

On a CPU, a thread context is typically defined by the values held in registers, which are stored and restored in case of a thread context switch. In Nymble's CMDFG model (cf. Section 12.3), all data- and control-flow values are represented by data flow edges in the associated CMDFGs, so a thread's context is defined by the values that flow between the operators comprised in the hardware computing unit.

In order to carry values across clock cycle boundaries and to store thread context in the datapath, we insert two different types of intermediate storage elements into the hardware datapaths:

- Single element storage (SES) only holds a single data item, but require significantly fewer hardware resources (registers in particular) for their hardware realization.
- Thread context storage (TCS) is able to store a single data item *per thread* and can be indexed by a thread's unique ID to retrieve the value for a particular thread.

We use registers for the implementation of both kinds of storage elements, because TCS typically store 4-8 elements (max. 256 bit) and an implementation in distributed or block RAM would be a waste of memory resources.

Our datapaths are statically scheduled, i.e. each operation is assigned a time-step, called *stage* in the Nymble context. It would be a waste of resources to insert thread context storages in more locations than absolutely necessary. In general the insertion of a thread context storage to hold a value is required only if a thread switch can occur between the time the value is produced by an operation, and the time it is consumed by its latest (in the schedule) user. Expressed as a rule, an operation **OP** that produces a data value must be provided with a thread context storage if any of the stages in the interval $[\mathbf{Start}(\mathbf{OP}), \mathbf{Last\ Use}(\mathbf{OP})[$ contains a variable-latency operation.

The CMDFG-excerpt in Fig. 12.3 depicts some typical scenarios showcasing examples for applications of the rules described above.

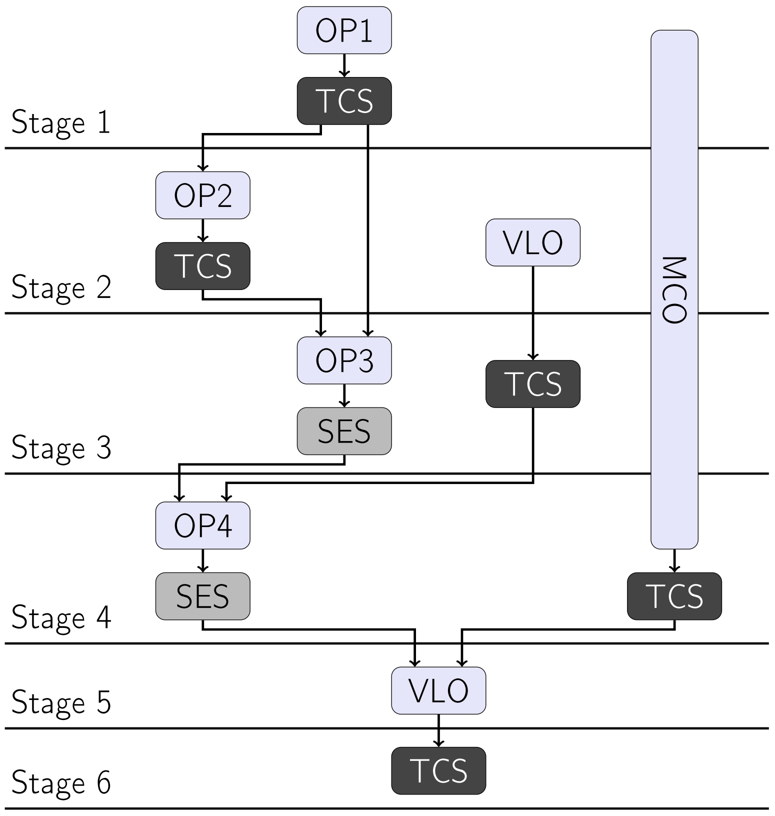


Figure 12.3: Example for the two different kinds of data storage in the datapath
(SES = single element storage, TCS = thread context storage, VLO = variable-latency operation, MCO = multi-cycle operation).

VLOs are generally provided with a thread context storage in the following stage. The multi-cycle operation (MCO, multiple clock cycles fixed latency) must be provided with a thread context storage as it spans across Stage 3, in which a thread switch can happen due to the VLO in Stage 2. Operation 1 (**OP1**) has been provided with a thread context storage because its last use (**OP3**), which is decisive for the rule, is again in Stage 3, where a thread switch can happen. In contrast to VLOs, thread-context storage linked to simple operations must be added to the same stage, as simple operators cannot hold the value. Operation 2 is also provided with a thread context storage due to the fact that the stage it has been scheduled in contains a VLO. For Operation 3, a simple single element storage is sufficient, as no thread switching can happen between its start in Stage 3 and its last use in Stage 4. The same holds true for Operation 4, because a potential thread switch caused by the variable-latency operation in Stage 5 will not occur in Stage 5, but in Stage 6.

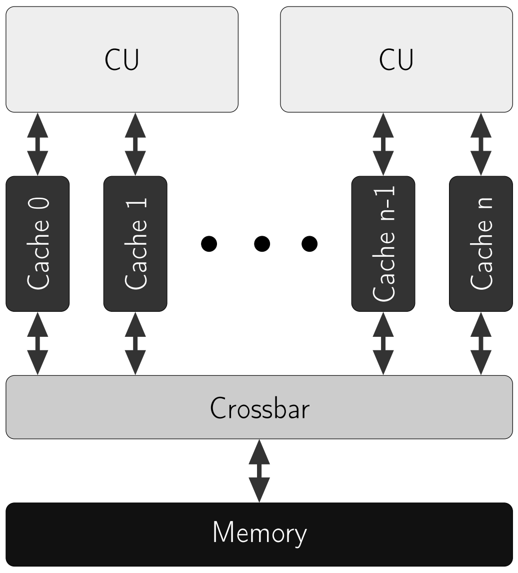


Figure 12.4: Cache- and memory-architecture, n is the total number of threads across all computing units.

With the thread-context storage included in the datapaths, the context of all currently non-active threads can be stored within the datapath, and restored when a thread's execution is resumed.

However, the interleaved execution of multiple threads on a computing unit is not the only way we can make use of the thread-level parallelism in OpenMP worksharing loops. Just as common in today's multicore CPUs, and similar to the previous approaches presented in Section 12.2, we further exploit thread-level parallelism by including *multiple* identical computing units in the hardware accelerator and distributing the computations of the worksharing loop across these computing units. The computing units work in isolation from each other. Thread context is not shared and threads do not migrate between units.

12.4.3 *Memory architecture*

In order to make use of potential spatial and/or temporal locality exhibited by memory accesses, we insert a cache-infrastructure in front of the interface providing access to the external RAM on the device. We use a direct-mapped cache with a *write-through* strategy for writes to memory. Each cache uses 512 cache-lines with 16 32-bit words in each cache-line. A dedicated AXI4 master interface is used in the kernel interface for memory accesses. The cache IP we use in our implementation provides a single AXI4 slave interface, but does not yet support reordering of accesses, i.e., a cache miss will delay all subsequent accesses, even if they want to access data present in the cache.

If such a cache would be shared by multiple threads, accesses from different threads can delay each other, with a potentially negative impact on performance. In order to achieve maximum performance, we instantiate a dedicated cache *per thread*, allowing each thread to access memory independently.

This model, depicted in Fig. 12.4, is compliant to the OpenMP shared memory model, which allows for threads to have their own view of memory between synchronization points [15]. In addition to the shared memory, the OpenMP standard allows for each thread to have *thread-private* memory, e.g., for individual loop counters. However, in our CMDFG-model such thread-local values are only present as intermediate values in the CMDFGs and are thread-private by design. Therefore we do not need to make any arrangements for explicit thread-private memory in our memory architecture.

12.5 EVALUATION

In this section we evaluate how our new execution model affects performance and the effects of the necessary changes to the datapath- and memory-architecture. We compare our new execution model to single-threaded execution and consider speedups as well as the costs of the hardware implementation of our execution model.

Similar to related work [4, 16], we use benchmarks with integer arithmetic. Our testcases are taken from the Adept benchmark suite [14]. Additionally we added an implementation of the sparse matrix-vector multiplication using the compressed row storage format. Furthermore, we also use floating-point implementations of each testcase for our evaluation. Compared to the integer versions, these testcases exhibit a different memory access behavior, i.e., memory accesses happen less frequently as the computation for each element takes longer. By including floating-point versions in our evaluation, we can study the effects of the memory access behavior in more detail. For our evaluation, vector-based benchmarks are set up to process 2000 element, matrix-based benchmarks work on matrices of 50×50 elements.

We compare our new multi-threaded execution model to an execution model where only a single thread is running on each computing unit. To this end, we examine four different configurations: A single-threaded configuration with two threads running on two computing units and three multi-threaded configurations with two computing units running two, three and four threads each (four, six, and eight threads total).

First, we evaluate the costs regarding hardware resources for the implementation of our execution model. This includes the necessary modifications to the datapath architecture (cf. Section 12.4.2) and the implementation of the controller and thread scheduler.

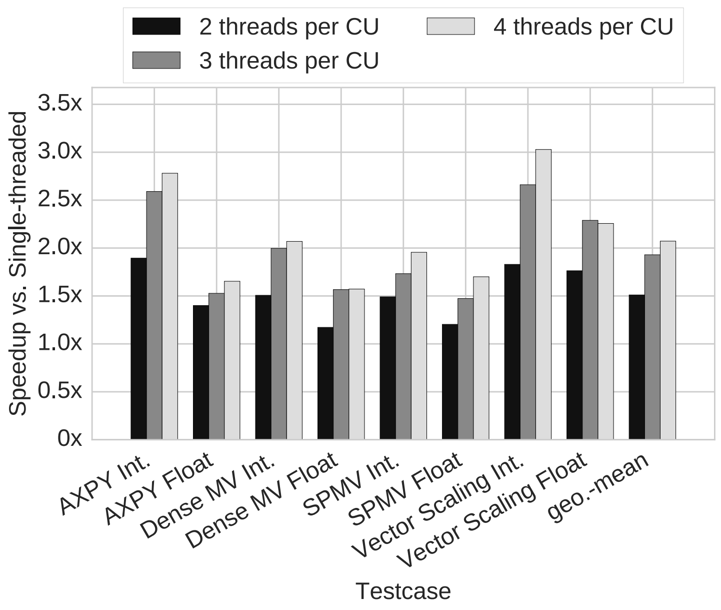


Figure 12.5: Speedup compared to single threaded execution.

We use Vivado 2016.4 for the FPGA implementation, targeting a Virtex 7 (XC7VX690T) device on a VC709 board. The memory and host-connectivity-infrastructure is configured to run at 200 MHz, independent from the actual operation frequency of the hardware kernel.

The results for all four configurations are given in Table 12.1. The number of DSP blocks used is unaffected by the choice of the execution model. As expected the number of occupied BRAM-slices increases with an increasing number of threads, as we need to spend more resources on the extra caches in the design used for the additional threads (cf. Section 12.4.3). The logic resources required for the implementation of the additional caches, our thread-switching logic and the thread-context storage also cause an increase in resource usage.

However, the biggest effect of our execution model with regard to the synthesis results is the limitation of the operation frequencies of the hardware kernels. The complexity of the combinatorial computations and indexing of thread-context storage causes the thread-switching logic to become critical for the achievable frequency and causes the frequency to decrease with an increasing number of threads interleaved on the computing unit. This effect has also been observed by Choi et al. in their work [4].

Despite the negative impact of the lower operation frequencies on performance, we are still able to gain a significant speedup over the single-threaded execution. The bar-plot in Fig. 12.5 shows the relative speedup over execution with a single thread per computing unit for each of our three multithreaded configurations. The execution times were obtained by executing the resulting FPGA accelerator designs on the VC709-board. The accelerated program is running on the host computer and data is transferred to the external RAM on the FPGA

Table 12.1: FPGA implementation results (post-place&route): Varying number of threads per computing unit

	1 thread per computing unit						**2 threads per computing unit**					
Testcase	Exec.-time (μs)	Freq. (MHz.)	LUT(%)	FF(%)	BRAM(%)	DSP(%)	Exec.-time (μs)	Freq. (MHz.)	LUT(%)	FF(%)	BRAM(%)	DSP(%)
AXPY Int.	3815	175	17.86	12.86	31.12	0.17	2013	165	20.63	15.23	34.46	0.17
AXPY Float	3997	175	18.34	13.06	31.12	0.11	2854	157	21.62	16.17	34.66	0.11
Dense MV Int.	4785	162	19.17	13.09	31.12	0.50	3175	150	23.42	17.00	34.66	0.50
Dense MV Float	5714	169	19.47	13.28	31.12	0.44	4873	135	23.68	17.18	34.66	0.44
SPMV Int.	1803	168	19.58	13.10	31.12	0.17	1209	162	23.98	17.11	34.66	0.17
SPMV Float	2086	180	19.23	12.54	30.92	0.11	1733	140	23.47	16.53	34.46	0.11
Vector Scaling Int.	3434	175	17.62	12.81	31.12	0.17	1877	155	20.80	15.74	34.66	0.17
Vector Scaling Float	3302	188	17.79	12.88	31.12	0.11	1873	157	20.93	15.81	34.66	0.11

	3 threads per computing unit						**4 threads per computing unit**					
Testcase	Exec.-time (μs)	Freq. (MHz.)	LUT(%)	FF(%)	BRAM(%)	DSP(%)	Exec.-time (μs)	Freq. (MHz.)	LUT(%)	FF(%)	BRAM(%)	DSP(%)
AXPY Int.	1473	150	23.57	17.89	37.93	0.17	1372	126	27.16	21.29	41.67	0.17
AXPY Float	2617	133	24.56	18.84	38.13	0.11	2418	130	27.55	21.48	41.67	0.11
Dense MV Int.	2396	150	26.71	20.09	38.13	0.50	2312	132	30.60	23.17	41.67	0.50
Dense MV Float	3648	146	27.07	20.27	38.13	0.44	3635	134	30.76	23.35	41.67	0.44
SPMV Int.	1040	141	27.26	20.24	38.13	0.17	922	133	30.91	23.34	41.67	0.17
SPMV Float	1416	139	27.55	20.42	38.13	0.11	1227	139	31.33	23.51	41.67	0.11
Vector Scaling Int.	1291	150	22.86	17.54	37.93	0.17	1134	122	25.56	20.09	41.46	0.17
Vector Scaling Float	1443	150	23.66	18.38	38.13	0.11	1463	127	26.47	20.94	41.67	0.11

Number of LUTs, FFs, BRAM slices and DSP blocks are given as percentage for brevity. The total number of available resources are 433200, 866400, 1470 and 3600 respectively.

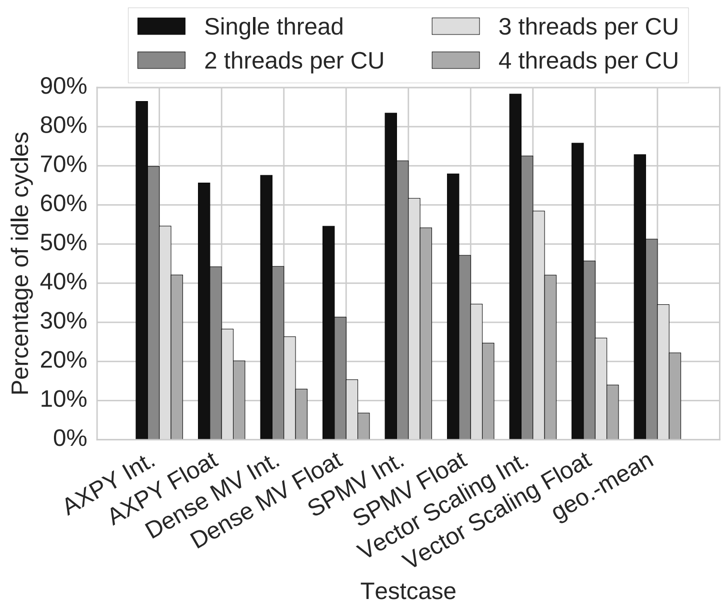


Figure 12.6: Percentage of idle cycles encountered during execution.

board using PCI Express. We use performance counters inside the kernels and combine them with the clock period achieved during FPGA implementation to calculate the runtime. The data transfer time from host to FPGA and back are not included in the runtime, as they are independent of the execution model used.

We are able to achieve a significant speedup in all testcases, in some cases of more than a factor of 3x by interleaving the computation of multiple threads. The geo.-mean speedups are 1.51x, 1.93x and 2.07x, respectively.

The impact of our execution model is also visible in Fig. 12.6, which relates the number of idle cycles to the number of overall cycles. While the single-threaded computing unit lies idle for more than half of the execution time in almost all cases, our execution model improves the utilization of the computing units significantly. With an increasing number of threads assigned to each computing unit, stalls can be hidden more effectively with computations from other threads, resulting in a reduction of the idle-cycles by more than factor 8x (testcase *Dense MV Float*).

While there is a significant increase in speedup when going from two threads per CU to three threads per CU for most of the cases, this trend cannot always be sustained when increasing the number of threads per CU to four. In those cases the interleaving of three threads on each CU already leads to a high utilization (cf. Fig. 12.6), leaving less headroom for an extra thread. In combination with the reduced clock frequency, the addition of another thread is not always beneficial, as can be seen in testcase *Vector Scaling Float*.

As described earlier, the performance is also affected by the frequency of memory accesses. As visible in Fig. 12.5, we generally achieve a higher speedup for integer benchmarks, where memory access happen more frequently, and the time spent during stalls caused by memory accesses makes up for a greater portion of the execution time. Especially in these cases the strength of our execution model, the effective hiding of memory access latencies, comes into play.

In summary, the synthesis of hardware accelerators from OpenMP programs clearly benefits from the interleaving of threads on the computing units in our execution model. The number of idle cycles decreases significantly (up to factor 8x) and speedups of more than a factor of 3x can be achieved. The implementation of our execution model requires some additional resources, especially for the extra caches. The optimal number of threads is dependent on the input program and the memory access behavior exhibited by the program.

12.6 CONCLUSION AND OUTLOOK

In this work, we presented an extension of the Nymble hardware/software-co-compiler to automatically generate multithreaded hardware accelerators from OpenMP worksharing loops. The novel execution model presented here as well as the modifications made to datapath- and memory-architecture allow to interleave the execution of multiple threads on one or more hardware computing units in a time-sliced manner.

We investigated the impact of our new execution model on hardware resource consumption and performance and compared our model to single-threaded execution. Our results show that the HLS of FPGA-based accelerators clearly benefits from our execution model. The interleaving of multiple threads on a computing unit allows to efficiently hide latencies caused by memory accesses. We were able to gain raw speedups of more than a factor of 3x with four-way multithreaded execution and improve the utilization of the computing units by more than a factor of 8x.

Our evaluation also showed that the optimal number of threads per computing unit is dependent on the input program and its memory access behavior. In the future, we want to automatically determine the best number of threads using compiler analyses. Besides that, we plan to add support for the OpenMP offloading constructs [1] to our compiler, allowing the user to clearly denote regions of code that should be extracted to a hardware accelerator and which and how data is mapped to the device memory. We also want to further investigate the impact of the memory- and cache-architecture on performance and how this infrastructure can be adapted to the input problem.

REFERENCES

[1] Samuel F. Antao, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. "Offloading Support for OpenMP in Clang and LLVM." In: *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*. LLVM-HPC '16. Salt Lake City, Utah: IEEE Press, 2016, pp. 1–11. ISBN: 978-1-5090-3878-7.

[2] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. "LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems." In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950423. (Visited on 05/08/2016).

[3] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. "A Cloud-Scale Acceleration Architecture." In: *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Oct. 2016.

[4] Jongsok Choi, S. Brown, and J. Anderson. "From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs." In: *2013 International Conference on Field-Programmable Technology (FPT)*. Dec. 2013, pp. 270–277. DOI: 10.1109/FPT.2013.6718365.

[5] Alessandro Cilardo, Luca Gallo, Antonino Mazzeo, and Nicola Mazzocca. "Efficient and Scalable OpenMP-Based System-Level Design." In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*. Mar. 2013, pp. 988–991. DOI: 10.7873/DATE.2013.206.

[6] Alessandro Cilardo, Luca Gallo, and Nicola Mazzocca. "Design Space Exploration for High-Level Synthesis of Multi-Threaded Applications." In: *Journal of Systems Architecture* 59.10, Part D (Nov. 2013), pp. 1171–1183. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2013.08.005. (Visited on 12/30/2015).

[7] Piotr Dziurzanski, Wlodzimierz Bielecki, Konrad Trifunovic, and M. Kleszczonek. "A System for Transforming an ANSI C Code with OpenMP Directives into a SystemC Description." In: *DDECS*. Vol. 6. 2006, pp. 151–152.

[8] J. Huthmann and A. Koch. “Optimized high-level synthesis of SMT multi-threaded hardware accelerators.” In: *2015 International Conference on Field Programmable Technology (FPT)*. 2015, pp. 176–183. DOI: 10.1109/FPT.2015.7393145.

[9] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch. “Hardware/Software Co-Compilation with the Nymble System.” In: *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. July 2013, pp. 1–8. DOI: 10.1109/ReCoSoC.2013.6581538.

[10] J. Huthmann, J. Oppermann, and A. Koch. “Automatic high-level synthesis of multi-threaded hardware accelerators.” In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2014, pp. 1–4. DOI: 10.1109/FPL.2014.6927432.

[11] Shin-Ying Lee and Carole-Jean Wu. “CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads.” In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT ’14. New York, NY, USA: ACM, 2014, pp. 175–186. ISBN: 978-1-4503-2809-8. (Visited on 03/11/2017).

[12] Y.Y. Leow, C.Y. Ng, and W.F. Wong. “Generating Hardware from OpenMP Programs.” In: *IEEE International Conference on Field Programmable Technology, 2006. FPT 2006*. Dec. 2006, pp. 73–80. DOI: 10.1109/FPT.2006.270297.

[13] Sascha Muhlbach, Martin Brunner, Christopher Roblee, and Andreas Koch. “Malcobox: Designing a 10 gb/s malware collection honeypot using reconfigurable technology.” In: *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE. 2010, pp. 592–595.

[14] Nick Johnson, Michèle Weiland, Trevor Carlson, and Sudarshan Balaji. *The Adept Benchmark Suite*. 2015. URL: http://www.adept-project.eu/images/whitepapers/p_590591_1446568406_Adept_Whitepaper_Benchmarks.pdf.

[15] OpenMP Architecture Review Board., ed. *OpenMP Application Programming Interface - OpenMP Standard 4.5*. Nov. 2015.

[16] Mingxing Tan, Bin Liu, Steve Dai, and Zhiru Zhang. “Multithreaded Pipeline Synthesis for Data-parallel Kernels.” In: *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD ’14. San Jose, California, 2014. URL: http://dl.acm.org/citation.cfm?id=2691365.2691510.

13 OPENMP DEVICE OFFLOADING TO FPGAS USING THE NYMBLE INFRASTRUCTURE

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *OpenMP Device Offloading to FPGAs Using the Nymble Infrastructure* by *Jens Huthmann, Lukas Sommer, Artur Podobas, Andreas Koch and Kentaro Sano* in *OpenMP: Portable Multi-Level Parallelism on Modern Systems (16th International Workshop on OpenMP, IWOMP 2020)*. The contribution of the author of this thesis is summarized as follows.

» *As the corresponding author, Lukas Sommer was responsible for the design and implementation of the Clang-based frontend for the processing of OpenMP device offloading code for the Nymble HLS compiler, including the transformations and optimizations performed on LLVM IR, preparing the IR for processing by the Nymble hardware backend. He was also in charge of the LLVM OpenMP offloading runtime plugin used to communicate with the FPGA accelerator. The architecture, hardware multithreading model and implementation in the Nymble HLS hardware backend was done by Jens Huthmann, as well as the integration with Intel OPAE shell. Experiments on CPU and FPGA were also performed by Jens Huthmann. The benchmark code was prepared by Jens Huthmann and Artur Podobas. The manuscript was joint work by all authors.* «

ABSTRACT

Next to GPUs, FPGAs are an attractive target for OpenMP device offloading, as they allow to implement highly efficient, applications-specific accelerators. However, prior approaches to support OpenMP device offloading for FPGAs have been limited by the interfaces provided by the FPGA vendors' HLS tool interfaces or their integration with the OpenMP runtime, e.g., for data mapping.

This work presents an approach to OpenMP device offloading for FPGAs based on the LLVM compiler infrastructure and the Nymble HLS compiler. The automatic compilation flow uses LLVM IR for HLS-specific optimizations and transformation and for the interaction with the Nymble HLS compiler. Parallel OpenMP constructs are automatically mapped to hardware threads executing simultaneously in the generated FPGA accelerator and the accelerator is integrated into `libomptarget` to support data-mapping.

In a case study, we demonstrate the use of the compilation flow and evaluate its performance.

13.1 INTRODUCTION

As the end of transistor scaling [30] draws near, researchers are actively pursuing and evaluating alternative emerging architectures and computing paradigms, with which they hope to continue performance scaling we have grown used to rely on. Among the more salient of these emerging architectures are reconfigurable systems, whose silicon plasticity/reconfigurability provides a partial remedy for the end of Moore's law [22]– we do not need more transistors, we just need to repurpose the existing transistor to better fit the requirements of our applications.

Today, Field-Programmable Gate Arrays (FPGAs) are among the more popular and mature reconfigurable systems available. While early FPGAs had limited computing capabilities, and were primarily used for circuit simulation and digital signal processing, modern FPGAs – on the other hand – feature tens of TeraFLOP/s of raw single-precision performance, and are capable of rivaling both general-purpose and graphics processing units (GPUs) in power efficiency and/or raw execution performance. Furthermore, with the increased maturity of High-Level Synthesis [12] tools, using FPGAs is no longer monopolized by hardware architectures, and instead, anyone with knowledge of C/C++/Java programming can map applications onto these exciting new architectures. Today, several research groups have already mapped important High-Performance Computing (HPC) applications onto FPGAs, with benefits illustrated over existing approaches [16, 26, 33–35]. These efforts have led to several research laboratories setting up large FPGA-based testbeds to investigate the role of these reconfigurable devices in a post Exa-scale era, such as the Noctua cluster at Paderborn University or the Cygnus cluster at University of Tsukuba.

In this paper, we present the Nymble OpenMP HLS infrastructure, which is a self-contained compilation tool-kit for running (a subset of) OpenMP constructs on FPGAs, and also visualize them using the Paraver [23] visualization tool. Unlike existing OpenMP HLS approaches, which use source-to-source compilation and rely on commercial black box compilers for hardware generation, Nymble is transparent and fully transforms OpenMP code down to Register Transfer Level (RTL) Verilog code without external dependencies. This, in turn, enables users to get a better insight into what hardware is actually generated, while at the same time providing an open platform for FPGA-based OpenMP research.

Our contributions in this paper are:

- A description over the Nymble infrastructure, including details on the front-end compilation and the hardware generation & architecture, including which OpenMP constructs Nymble supports and how they are implemented,

- A use-case showing how Nymble transforms well-known OpenMP code into hardware, including empirical performance evaluation, and
- A discussion on the future of OpenMP for FPGAs, including challenges and directions

13.2 MOTIVATION

Today, FPGAs are being considered to complement (and compete with) the general-purpose processor and GPUs that currently reside in modern HPC infrastructure. Several research laboratories are already setting up large FPGA-based testbeds to investigate the role of these reconfigurable devices in a post-Exa-scale era, such as for example the Noctua cluster at Paderborn University or the Cygnus cluster at the University of Tsukuba.

Meanwhile, using these accelerators in a user-friendly way (that is, without resorting to writing RTL code), is often limited to using vendor-specific toolchains, such as for example Intel's OpenCL SDK for FPGA [8] or Xilinx SDSoC/SDAccel [32]. While these toolchains are often high-performing, they are also very tied to a specific execution model. Furthermore, adding or researching into alternative programming models using these vendor solutions (such as for example OpenMP) is challenging, because tools are closed source, and even if some aspects can be changed (such as the Board Support Package, BSP), these changes become non-trivial.

There are methods to extend functionality, such as using source-to-source methods to transcompile OpenMP [10], but these methods have no way of even remotely controlling or dictating how the underlying hardware is generated. Finally, vendor tools and road-maps are not always necessarily aligned with what we as users or researchers need, meaning that it is imperative to look at alternative approaches, in particular for guiding and doing research on OpenMP execution on future FPGAs. The Nymble OpenMP infrastructure aspires to be one such alternative for OpenMP researchers and users.

13.3 THE NYMBLE OPENMP INFRASTRUCTURE

The goal of this work is to develop a compilation flow that maps OpenMP target regions to FPGA-based accelerators without requiring manual intervention by the user. The compilation flow is based on the LLVM compiler infrastructure [18] and its implementation of OpenMP. In contrast to many prior approaches that use source-to-source transformations on AST-level to extract target regions for HLS (see Section 13.6 for detailed discussion), the compilation flow in this work uses LLVM IR to interact with the HLS tool. This approach

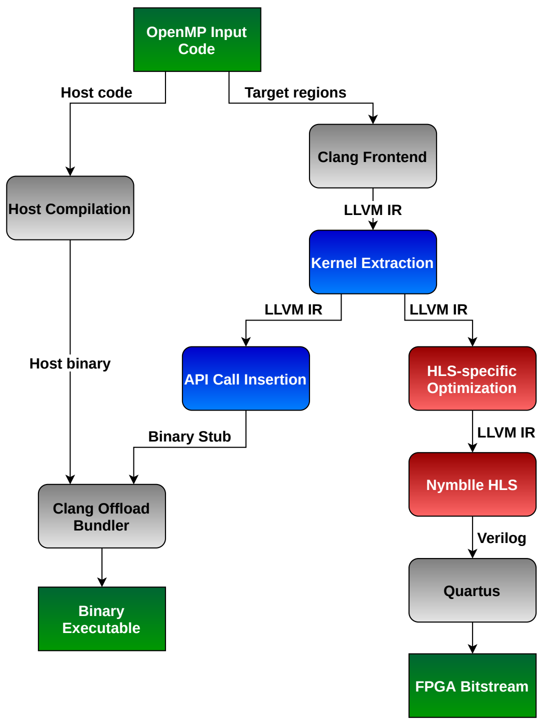


Figure 13.1: Overview of the compilation flow.

facilitates code transformations that can be used to transform and optimize target regions, described in more detail in Section 13.3.1.

As the commercially available HLS-tools only provide source-level interfaces and no official interface on IR-level, the state-of-the-art academic HLS compiler *Nymble* [15] is used for the actual High-Level Synthesis of the target regions. Besides providing an IR-level interface, Nymblle also supports true multi-threading in the generated accelerators [14], described in more detail in Section 13.3.2.

13.3.1 *Compilation Flow*

Fig. 13.1 presents an overview of our compilation flow. For OpenMP device offloading, LLVM's Clang frontend uses separate compilation passes for host- and device code. For this work, the host compilation remains completely unchanged and therefore supports any host code and OpenMP host constructs that Clang supports.

The device compilation flow (shown on the right-hand side of Fig. 13.1) does not only support the basic `target` directive to denote target regions and the *full* range of data-mapping constructs (`map`-clause, `target data`-directive, array-sections, etc.), but also provides two kinds of parallelism: The `teams` or `parallel` construct can be used inside a target region to express parallelism, Section 13.3.2 explains how this parallelism is realized in hardware. Note that in our current prototype, only one of these constructs can be used at a time and nested parallelism is not supported. For the `teams` construct, the `distribute` construct is also supported to specify worksharing for a loop nest.

Similar to many approaches investigated in the survey by Mayer et al. [21] (see Section 13.6 for detailed discussion), a binary stub for execution on the host machine is generated as one of the products of the device compilation flow. In this work, the binary stub is not only used to initiate the FPGA execution, but also to handle parallelism. Parallel constructs will spawn multiple software threads in the binary stub, these threads then interact with one hardware thread each in the FPGA-accelerator in an 1:1-relationship. This approach allows to re-use the standard mechanisms from LLVM's OpenMP runtime `libomp` to manage thread spawning and worksharing. Therefore, after generating LLVM IR in the Clang frontend, the *Kernel Extraction* splits the outlined target function into the stub to remain on the host and the actual target region kernel function for High-Level Synthesis.

The *API Call Insertion* then inserts calls to a thin wrapper library around Intel's *Open Programmable Acceleration Engine*[1] into the stub function to transfer function arguments and initiate hardware execution. Note that, in contrast to approaches such as [17], data-management is not handled via generated API calls, but rather through a plugin for LLVM's `libomptarget`, enabling the whole range of data-mapping clauses/constructs, including array sections and uni-directional transfers (`to`/`from` clause). The stub is then compiled for the host machine (`x86-64` in our case) and included in the binary executable using the Clang Offload-Bundler [1]. At runtime, the stub is loaded by `libomptarget` and initiates the execution on the FPGA accelerator.

The extracted HLS kernel undergoes a number of transformations and optimizations before actual High-Level Synthesis (*HLS-specific Optimizations* in Fig. 13.1). The transformations are mainly concerned with transforming OpenMP language constructs into constructs suitable for High-Level Synthesis. Currently, the prototype supports the OpenMP API runtime functions `omp_get_thread_num`, `omp_get_num_threads`, `omp_get_team_num` and `omp_get_num_teams`, which, in addition to `teams distribute`, can be used to assign individual workloads to the different threads. Besides that, the synchronization constructs `omp critical` and `omp barrier` are also supported inside target regions and mapped to efficient implementations using hardware semaphores.

Static allocation of thread-private memory inside the target region (`alloca` in LLVM IR) is also supported by the compilation flow and HLS backend and automatically mapped to low-latency accessible local memory (SRAM) on the FPGA device. Vector datatypes are also allowed in the target regions, but arithmetic operations on vectors are realized as individual operations on each vector element, as vector operations do not provide significant benefits in FPGA hardware.

1 https://opae.github.io/

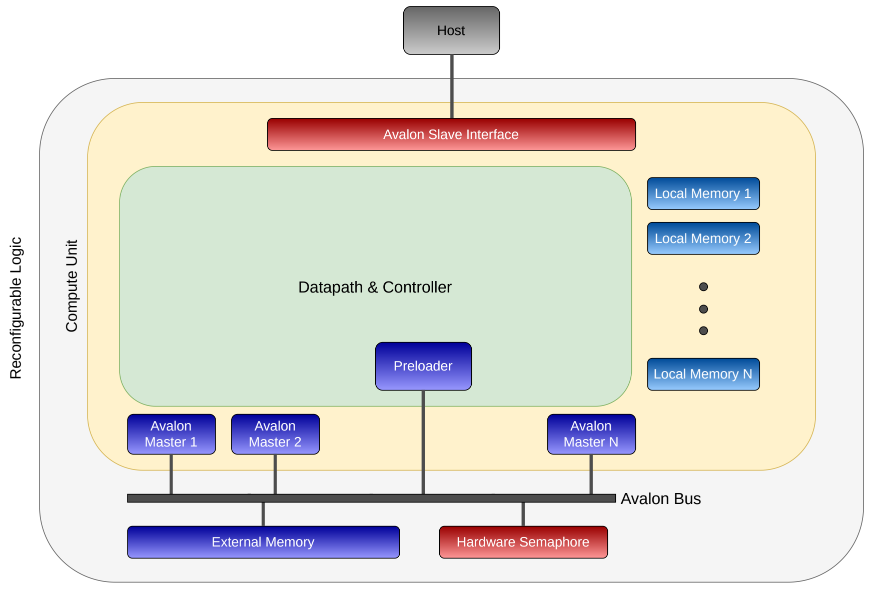


Figure 13.2: Hardware architecture of the reconfigurable accelerator.

Therefore, to allow for more fine-grained scheduling during HLS, we automatically partition vector-wide thread-private memories into individual local memories for each element while preserving array semantics as one of the optimization steps.

The transformed LLVM IR is then passed to the Nymble HLS backend, which performs the typical HLS steps of allocation, binding and scheduling. For this purpose, the LLVM IR is transformed into a *control dataflow graph* (CDFG) representation, as described in [15]. More details on the mapping of different constructs to hardware will be presented in the next section.

The final product of the Nymble HLS backend is an HDL (Verilog) description of the accelerator, which is passed to Intel's Quartus software for synthesis and place-and-route, eventually yielding an FPGA bitstream.

13.3.2 *Hardware Architecture*

The overall hardware architecture of the generated FPGA accelerator is depicted in Fig. 13.2. The *Avalon slave interface* of the compute unit (CU) that is connected to the host is used as entry point for the hardware execution. The memory mapped register file can be used to pass kernel arguments and other information (e.g., thread ID) from the software thread to the corresponding hardware thread.

For larger data, the accelerator supports two different kinds of memory:

- Small, on-chip (SRAM) local memories (*LMEM*) are directly connected to the compute unit. These memories can be used as thread-private memory.

- *External memory* (DRAM) located on the FPGA-board can be used to hold large amounts of data and also for data-exchange with the host RAM using the OpenMP data-mapping constructs via the `libomptarget`-plugin. This memory is connected to the CU via an Avalon bus, with a dedicated Avalon master port per hardware thread.

As the data-width of the external memory interface is usually higher than the size of single data-item of primitive type (e.g., `float`), vector data-types can be used in the OpenMP input code to improve the memory access efficiency. Where possible, vector-wide memory accesses are automatically mapped to Avalon burst accesses.

Another mechanism to further improve the memory access efficiency is the use of the *Preloader*. By using calls to the custom function `omp_target_preload` in the OpenMP input code to transfer data between global memory and thread-private local memory, the required data can be transferred efficiently in a single burst transfer. A more detailed discussion of the Preloader can be found in Section 13.4.1.

The Avalon bus system is also used to integrate the memory-mapped *Hardware Semaphore* that is used to realize the `omp critical` and `omp barrier` synchronization constructs.

The execution inside the *Datapath* is based on the Nymble-MT execution model presented in prior work by Huthmann et al. [14]. The unique feature of this execution model is the fact that it supports the simultaneous execution of multiple hardware threads in a *single* compute-unit, whereas most other FPGA-based approaches achieve thread-level parallelism through spatial replication of the compute-unit (e.g., [6], cf. Section 13.6 for discussion).

To allow for simultaneous execution of multiple hardware threads, the operations found in the data-flow graph of the kernel are organized into so-called *stages* according to their static HLS schedule. The different stages can operated independently by the controller, allowing multiple threads to be active in different stages simultaneously. The stage-based execution model in addition also support loop pipelining.

The threads can operate completely independently of each other in this model, also allowing threads to start and finish at different points in time. Hardware threads are launched by their software counterpart (as stated in the previous section, we use a 1:1-relationship between software- and hardware threads) through the entry point in the Avalon slave interface. Parallelism in the OpenMP execution model (threads/teams) is automatically mapped to these simultaneously operating threads by the compilation flow presented here.

A major challenge in the stage-based execution model is the integration of operations for which the latency (in clock cycles) cannot be determined statically, e.g., accesses to external memory, which we call *variable-latency operations* (VLO). These operations are scheduled assuming their minimum latency. In case a VLO exceeds the assumed latency at execution time, the execution of the encountering thread is suspended until the VLO completes. To make sure that a single thread encountering a longer-than-expected latency does not block other threads, stages containing a VLO allow for thread re-ordering, i.e., threads can overtake each other in these stages.

13.3.3 *Performance Visualization*

Just as with any other device or target platform, the optimization of application code is an important step to achieve performance on FPGAs and is often an iterative process. To assist developers in this process, the compilation flow developed in this work provides mechanisms to automatically include various performance counters directly in the generated hardware. While the full details of the hardware implementation are out of scope for this work, the performance counters were designed to be as non-invasive as possible, i.e., to not have an impact on the performance of the investigated accelerator design, e.g. by increasing the initiation interval of pipelined loops.

The performance counters allow to capture important metrics such as memory bandwidth, arithmetic operations per time-interval (e.g. GFLOPs) or hardware thread idle times and facilitate the analysis and optimization of the target regions offloaded to the FPGA. After the execution on the FPGA completes, the collected performance data is exported in the Paraver trace format for use with the popular HPC performance visualization tool Paraver [23]. The integration with a state-of-the-art HPC visualization tool makes the performance analysis of the FPGA target regions more accessible for HPC domain experts.

13.4 EVALUATION

To demonstrate the compilation flow from OpenMP with target offloading to FPGA-based accelerators, we use a well-understood benchmark as case study. The selected application allows to test the different features of the compilation flow and architecture template by covering the supported OpenMP constructs as mentioned in the previous section, including synchronization.

For the application, a single compute unit is implemented inside the FPGA, supporting the simultaneous execution of up to four threads. The implementation of the compilation flow is based on LLVM release 9.0 and Quartus Prime version 18.1.2 is used for synthesizing the generated Verilog code to an FPGA bitstream.

The targeted FPGA is an Intel FPGA PAC D5005 card. The card is coupled via PCIe to the host processor, a quad-core Xeon Gold 5122 CPU which executes the host-portion of the applications and is also used for CPU benchmarking. Note that the performance figures always include data-transfers between host- and FPGA external memory via

PCIe, initiated through libomptarget, i.e., the numbers reported here are end-to-end performance of the FPGA offloading.

13.4.1 *Case Study: GEMM*

As an example application, we use the general matrix multiplication (GEMM). The FPGA accelerator is compiled from a blocked version of GEMM and the different hardware threads compute distinct submatrices of the overall result matrix. Inside the computation of each thread, the computation is partially unrolled to exploit the potential of *spatial* parallelism provided by FPGAs. To reduce the number of expensive accesses to global, external memory, local memory is used to buffer inputs and intermediate results. To further improve the efficiency of memory access to the input matrices A and B, the threads preload blocks of the input matrices into the local memory using the preloader that is part of the compute unit. For users of the compilation flow, the preloading capability is available through a simple C++ template function called omp_target_preload (cf. Listing 13.1), which simply gets passed the relevant pointers to external and local memory and the number and type of the elements to load.

```
template <typename T, int ELEMENTS>
void omp_target_preload(size_t offset,
                        size_t stride,
                        size_t num_transfers,
                        void* globalSrc,
                        void* localDst){...}
```

Listing 13.1: Definition of the omp_target_preload-function

The preloader will then collect the access to multiple elements in a single Avalon (burst) request, significantly improving the memory access efficiency. To further leverage the spatial parallelism, double buffering is implemented for the local memory and the preloading for the next block happens in parallel to the computation of the current block. All these optimizations have been implemented using standard OpenMP or, in case of unrolling (pragma unroll), compiler annotations and C++ constructs. The omp_target_preload-function was designed to be very generic and corresponds to a pattern often found in accelerator programming (e.g., GPU programming), the preloading of relevant input data from global memory to local memory. An usage example of the preload-function can be found in Listing 13.2.

```
void gemm(float* A,...){
  [...]
  VECTOR A_local[BUFFERING][BLOCK_SIZE];
  omp_target_preload<float, BLOCK_SIZE>(
      (i*DIM)+k,
      DIM,
      BLOCK_SIZE,
      (void*) A,
      (void*) &A_local[buffer%BUFFERING *
          ↪ BLOCK_SIZE]);
  [...]
}
```

Listing 13.2: Usage example of the `omp_target_preload`-function (excerpt).

Fig. 13.3 shows the performance of the FPGA accelerator with different numbers of hardware threads executing simultaneously in the single compute unit for matrices of dimensions 8192×8192. While the performance of the accelerator almost doubles when going from a single to two threads, the increase slows down for three and four threads, respectively. In these cases, the threads do not only compete for compute resources in the multithreaded accelerator, but also for memory bandwidth to the external memory. The comparison with the BLAS implementation from the ATLAS library [31] on the Xeon CPU shows that the accelerator with a single thread outperforms a single thread on the CPU, but is not able to keep up with an execution with four threads on the CPU, partially also due to the data-transfers between host and FPGA.

In terms of hardware resource usage, the accelerator takes up 14% of logic resources, 16% of BRAM and 18% DSPs at a frequency of 183 MHz. Despite the relative low resource usage, it does not make sense to further increase the number of threads due to the negative impact on operating frequency. Instead, the remaining resources could be utitlized to duplicate the accelerator and compute on multiple compute units in parallel in future versions of the proposed architecture.

In order to validate the support for OpenMP synchronization constructs via a lock implemented in the bus-attached hardware semaphore, an alternative version of GEMM, where each thread computes parts of the result for each element of the result matrix. The computed partial result is then added to the overall result inside a `critical` region. Even though the hardware semaphore allows for efficient locking, this version of GEMM, due to the very frequent access to global memory, delivers less performance than the optimized version using local memories described above.

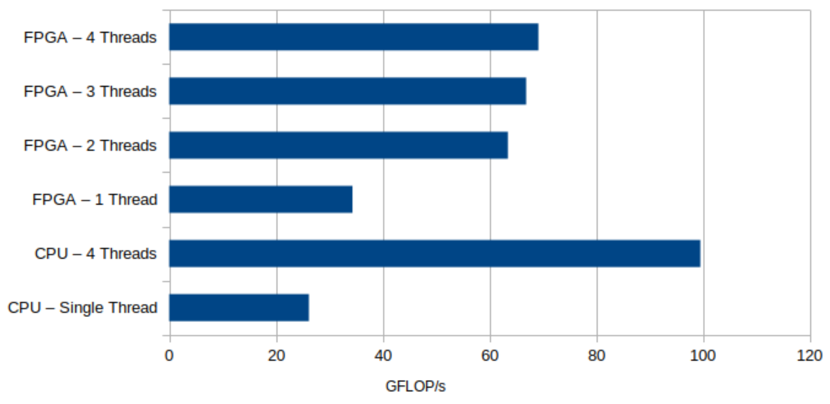


Figure 13.3: Arithmetic performance of the blocked GEMM computation in GFLOP/s with different numbers of hardware threads simultaneously active in the compute unit.

13.5 DISCUSSION

In this paper, we have demonstrated the Nymble infrastructure and shown that we can support a significant subset of OpenMP target offloading on FPGAs without much loss of generality, and that many of the properties (load-imbalance, scalability, etc.) materialize even in hardware. However, there are ample opportunities and future work for OpenMP on FPGAs, some of which we discuss herein.

OpenMP tasking, introduced in v3.0 (and dependent tasks in v4.0), is a construct that we would like to support in the Nymble subsystem. In theory, all necessary ingredients to support tasking is already provided by Nymble, and scheduling could be in a very software manner. However, such a solution would likely bloat the generated hardware, and a more customized approach is preferable (such as Nexus [9]), but a trade-off between consumed FPGA resources and the added performance must be performed. Alternatively, we could outsource task-management to a soft-core (e.g., a RISC-V [2]) that only orchestrates and resolves dependencies. More importantly, the FPGA allows for customizing communication between threads (and thus tasks), leading to interesting opportunities, particularly for dependent tasks.

One exciting future direction is concerning the synchronization and atomicity of operations. Today, Nymble uses a customized mutex hardware core (that is memory mapped) to support atomicity and synchronization. While this is a correct and functional way of supporting them, there are likely better ways that leverage the customization that FPGAs give us. For example, since we are working with an FPGA, we could, in theory, place the functionality of atom updates inside the external memory controller (DDR4 in our case). Similarly, rather than going through shared memory for synchronization, we could have a system-wide token bus that synchronizes all the threads (by sending and forwarding a synchronization token).

Another opportunity, unique for the FPGA, is concerning the recent memory allocations added in OpenMP. Because the memory hierarchy can be fully customized, we foresee that there are many future opportunities for tuning these for a particular performance criteria (e.g., execution time or power-consumption). For example, we could mark part of the FPGA that would be dedicated to the memory hierarchy as a partially reconfigurable region, and then *dynamically* adapt and optimize the actual hardware in real-time, such as for example changing cache sizes or replacement policies, scratchpad memories, coherency (or coherency-less) islands of memory, and so on and forth, in order to facilitate high-performance, low-latency producer/consumer patterns in (for example) the OpenMP 4.0 dependent tasks.

The representation of floating-point numbers has recently become a hot topic, with multiple authors proposing (and evaluating) new representations such as Posit [13] and Elias encoding [20]. Today, OpenMP does not contain support for setting a particular region to use a specific representation, but in the future, it might. FPGAs can execute arithmetic operations on these exciting new representations at high speed [27]. If selecting number representation will be part of future OpenMP standard, then FPGAs will be the platform that can exploit it to the fullest.

Finally, scaling OpenMP onto multiple FPGAs is an open question. On hand, we could rely on OpenMP's accelerator directives, and treat each device a discrete system with little to no access to other systems. However, on FPGAs, we can do more, and create/include special hardware to (for example) support a shared-memory view across multiple FPGAs, or use tasks as containers that encapsulate produced/consumed data, that are exchanged among FPGAs.

In short, our understanding of OpenMP on FPGAs is just starting, and there are ample opportunities and future directions where this work affect OpenMP in the future.

13.6 RELATED WORK

As OpenMP-based programming is very attractive for integrating FPGAs into HPC systems and toolflows, a number of previous works has presented approaches for mapping OpenMP to FPGAs. A good overview of these approaches can be found in the survey by Mayer et al. [21].

Early approaches tried to map OpenMP tasks [4, 11, 24, 25] or worksharing constructs [6, 7, 19], such as `parallel for` to FPGA accelerators. As these approaches date back to the time before the OpenMP target constructs were standardized, no OpenMP constructs for specifying data mapping and device-specific execution were available for these approaches.

More recent approaches combine the OpenMP device constructs with commercially available HLS tools. Many of these works take an approach where target regions are extracted from the input program on AST-level [3, 28], making OpenMP-specific optimizations before HLS difficult. The approach presented by Ceissler et al. [5] even requires the accelerator cores to be implemented in a hardware-description language and uses OpenMP only for the integration into the overall application. Only the work by Knaust et al. [17] uses IR (namely LLVM-IR) to interact with the HLS tool through an undocumented interface. However, as the data-transfers via the OpenCL API are statically generated during compile-time, their approach does not support array sections or mapping of data in only one direction (`to` or `from`), a limitation not found on our approach.

All of the tools mentioned above try to achieve a speedup over sequential execution through spatial parallelism (e.g., a dedicated accelerator core per thread) and classical HLS optimization techniques such as loop pipelining, but none of them supports actual hardware multi-threading inside the accelerator core. In contrast, in [29], OpenMP worksharing loops were mapped to multi-threaded accelerator cores. However, their threading model is much more limited than the one used in this work, as in their model, only a single thread can be active at a time and threads would only be switched when the active thread was suspended due to memory access latency.

As one of the key challenges for an effective mapping of OpenMP constructs to FPGA hardware, Mayer et al. [21] identified the code analysis and optimization across the border between compiler frontend and low-level HLS tool. With our fully integrated compilation flow from input program to Verilog, we are able to propagate information across this border and exploit knowledge of the underlying FPGA execution model for high-level, FPGA-specific transformations on IR-level in the compiler frontend.

13.7 conclusion

This work presented a compilation flow for targeting FPGAs with OpenMP device offloading, in combination with a complete integration in `libomptarget` for complete data management support. The presented compile flow supports a significant subset of OpenMP for device offloading, including parallel constructs (e.g., `parallel`, `teams`) that are mapped to actual hardware threads executing simultaneously in the generated, multi-threaded accelerator, a unique feature of the presented approach.

By optimizing across the border between compiler front-end and the HLS-tool based on LLVM and the academic HLS-compiler Nymble, FPGA-specific optimizations were integrated in the compile flow. This insight could also be interesting for FPGA's vendor and a motivation to further open up their HLS-compiler IR interfaces for OpenMP-based compilation flows.

The case study showed that it is possible to target FPGAs from OpenMP programs, using only standard programming language constructs and annotations, without any HLS-specific extensions, and also showcased an integration of a data preloading functionality that could also be of interest on other accelerator architectures (e.g. GPUs). As described in Section 13.5, OpenMP is an interesting option for integrating FPGAs into parallel and heterogeneous applications, with a number of interesting research avenues.

REFERENCES

[1] Samuel F. Antão, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. "Offloading Support for OpenMP in Clang and LLVM." In: *Third Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2016, Salt Lake City, UT, USA, November 14, 2016*. IEEE Computer Society, 2016, pp. 1–11. DOI: 10.1109/LLVM-HPC.2016.006. URL: https://doi.org/10.1109/LLVM-HPC.2016.006.

[2] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tech. rep. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, Aug. 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[3] Jaume Bosch, Xubin Tan, Antonio Filgueras, Miquel Vidal, Marc Mateu, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. "Application Acceleration on FPGAs with OmpSs@FPGA." In: *International Conference on Field-Programmable Technology, FPT 2018, Naha, Okinawa, Japan, December 10-14, 2018*. IEEE, 2018, pp. 70–77. DOI: 10.1109/FPT.2018.00021. URL: https://doi.org/10.1109/FPT.2018.00021.

[4] Daniel Cabrera, Xavier Martorell, Georgi Gaydadjiev, Eduard Ayguadé, and Daniel Jiménez-González. "OpenMP extensions for FPGA accelerators." In: *Proceedings of the 2009 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2009), Samos, Greece, July 20-23, 2009*. Ed. by Walid A. Najjar and Michael J. Schulte. IEEE, 2009, pp. 17–24. DOI: 10.1109/ICSAMOS.2009.5289237. URL: https://doi.org/10.1109/ICSAMOS.2009.5289237.

[5] Ciro Ceissler, Ramon Nepomuceno, Marcio Machado Pereira, and Guido Araujo. "Automatic Offloading of Cluster Accelerators." In: *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*. IEEE Computer Society, 2018, p. 224. DOI: 10.1109/FCCM.2018.00058. URL: https://doi.org/10.1109/FCCM.2018.00058.

[6] Jongsok Choi, Stephen Dean Brown, and Jason Helge Anderson. "From software threads to parallel hardware in high-level synthesis for FPGAs." In: *2013 International Conference on Field-Programmable Technology, FPT 2013, Kyoto, Japan, December 9-11, 2013*. IEEE, 2013, pp. 270–277. DOI: 10.1109/FPT.2013.6718365. URL: https://doi.org/10.1109/FPT.2013.6718365.

[7] Alessandro Cilardo, Luca Gallo, Antonino Mazzeo, and Nicola Mazzocca. "Efficient and scalable OpenMP-based system-level design." In: *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*. Ed. by Enrico Macii. EDA Consortium San Jose, CA, USA / ACM DL, 2013, pp. 988–991. DOI: 10.7873/DATE.2013.206. URL: https://doi.org/10.7873/DATE.2013.206.

[8] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. "From OpenCL to high-performance hardware on FPGAs." In: *22nd international conference on field programmable logic and applications (FPL)*. IEEE. 2012, pp. 531–534.

[9] Tamer Dallou, Nina Engelhardt, Ahmed Elhossini, and Ben Juurlink. "Nexus#: A distributed hardware task manager for task-based programming models." In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015, pp. 1129–1138.

[10] Antonio Filgueras, Eduard Gil, Daniel Jimenez-Gonzalez, Carlos Alvarez, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees Vissers. "OmpSs@ Zynq all-programmable SoC ecosystem." In: *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. 2014, pp. 137–146.

[11] Antonio Filgueras, Eduard Gil, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees A. Vissers. "OmpSs@Zynq all-programmable SoC ecosystem." In: *The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014*. Ed. by Vaughn Betz and George A. Constantinides. ACM, 2014, pp. 137–146. DOI: 10.1145/2554688.2554777. URL: https://doi.org/10.1145/2554688.2554777.

[12] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[13] John L Gustafson and Isaac T Yonemoto. "Beating floating point at its own game: Posit arithmetic." In: *Supercomputing Frontiers and Innovations* 4.2 (2017), pp. 71–86.

[14] Jens Huthmann and Andreas Koch. "Optimized high-level synthesis of SMT multi-threaded hardware accelerators." In: *2015 International Conference on Field Programmable Technology, FPT 2015, Queenstown, New Zealand, December 7-9, 2015*. IEEE, 2015, pp. 176–183. DOI: 10.1109/FPT.2015.7393145. URL: https://doi.org/10.1109/FPT.2015.7393145.

[15] Jens Huthmann, Björn Liebig, Julian Oppermann, and Andreas Koch. "Hardware/software co-compilation with the Nymble system." In: *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), Darmstadt, Germany, July 10-12, 2013*. IEEE, 2013, pp. 1–8. DOI: 10.1109/ReCoSoC.2013.6581538. URL: https://doi.org/10.1109/ReCoSoC.2013.6581538.

[16] Jens Huthmann, Abiko Shin, Artur Podobas, Kentaro Sano, and Hiroyuki Takizawa. "Scaling Performance for N-Body Stream Computation with a Ring of FPGAs." In: *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. 2019, pp. 1–6.

[17] Marius Knaust, Florian Mayer, and Thomas Steinke. "OpenMP to FPGA Offloading Prototype Using OpenCL SDK." In: *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. IEEE, 2019, pp. 387–390. DOI: 10.1109/IPDPSW.2019.00072. URL: https://doi.org/10.1109/IPDPSW.2019.00072.

[18] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665. URL: https://doi.org/10.1109/CGO.2004.1281665.

[19] Y. Y. Leow, C. Y. Ng, and Weng-Fai Wong. "Generating hardware from OpenMP programs." In: *2006 IEEE International Conference on Field Programmable Technology, FPT 2006, Bangkok, Thailand, December 13-15, 2006*. Ed. by George A. Constantinides, Wai-Kei Mak, Phaophak Sirisuk, and Theerayod Wiangtong. IEEE, 2006, pp. 73–80. DOI: 10.1109/FPT.2006.270297. URL: https://doi.org/10.1109/FPT.2006.270297.

[20] Peter Lindstrom. "Universal Coding of the Reals using Bisection." In: *Proceedings of the Conference for Next Generation Arithmetic 2019*. 2019, pp. 1–10.

[21] Florian Mayer, Marius Knaust, and Michael Philippsen. "OpenMP on FPGAs - A Survey." In: *OpenMP: Conquering the Full Hardware Spectrum - 15th International Workshop on OpenMP, IWOMP 2019, Auckland, New Zealand, September 11-13, 2019, Proceedings*. Ed. by Xing Fan, Bronis R. de Supinski, Oliver Sinnen, and Nasser Giacaman. Vol. 11718. Lecture Notes in Computer Science. Springer, 2019, pp. 94–108. DOI: 10.1007/978-3-030-28596-8_7. URL: https://doi.org/10.1007/978-3-030-28596-8%5C_7.

[22] Gordon E Moore. "Cramming more components onto integrated circuits." In: *Electronics Magazine* 38.8 (Apr. 1965).

[23] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. "Paraver: A tool to visualize and analyze parallel code." In: *Proceedings of WoTUG-18: transputer and occam developments*. Vol. 44. 1. Citeseer. 1995, pp. 17–31.

[24] Artur Podobas. "Accelerating Parallel Computations with OpenMP-Driven System-on-Chip Generation for FPGAs." In: *IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, MCSoC 2014, Aizu-Wakamatsu, Japan, September 23-25, 2014*. IEEE Computer Society, 2014, pp. 149–156. DOI: 10.1109/MCSoC.2014.30. URL: https://doi.org/10.1109/MCSoC.2014.30.

[25] Artur Podobas and Mats Brorsson. "Empowering OpenMP with automatically generated hardware." In: *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2016, Agios Konstantinos, Samos Island, Greece, July 17-21, 2016*. Ed. by Walid A. Najjar and Andreas Gerstlauer. IEEE, 2016, pp. 245–252. DOI: 10.1109/SAMOS.2016.7818354. URL: https://doi.org/10.1109/SAMOS.2016.7818354.

[26] Artur Podobas and Satoshi Matsuoka. "Designing and Accelerating Spiking Neural Networks using OpenCL for FPGAs." In: *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE. 2017, pp. 255–258.

[27] Artur Podobas and Satoshi Matsuoka. "Hardware implementation of POSITs and their application in FPGAs." In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2018, pp. 138–145.

[28] Lukas Sommer, Jens Korinth, and Andreas Koch. "OpenMP device offloading to FPGA accelerators." In: *28th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2017, Seattle, WA, USA, July 10-12, 2017*. IEEE Computer Society, 2017, pp. 201–205. DOI: 10.1109/ASAP.2017.7995280. URL: https://doi.org/10.1109/ASAP.2017.7995280.

[29] Lukas Sommer, Julian Oppermann, Jaco Hofmann, and Andreas Koch. "Synthesis of interleaved multithreaded accelerators from OpenMP loops." In: *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017, Cancun, Mexico, December 4-6, 2017*. IEEE, 2017, pp. 1–7. DOI: 10.1109/RECONFIG.2017.8279823. URL: https://doi.org/10.1109/RECONFIG.2017.8279823.

[30] M Mitchell Waldrop. "The chips are down for Moore's law." In: *Nature News* 530.7589 (2016), p. 144.

[31] R. Clint Whaley and Antoine Petitet. "Minimizing development and maintenance costs in supporting persistently optimized BLAS." In: *Software: Practice and Experience* 35.2 (Feb. 2005), pp. 101–121.

[32] Loring Wirbel. "Xilinx SDAccel: a unified development environment for tomorrow's data center." In: *The Linley Group Inc* (2014).

[33] Chen Yang, Tong Geng, Tianqi Wang, Charles Lin, Jiayi Sheng, Vipin Sachdeva, Woody Sherman, and Martin Herbordt. "Molecular Dynamics Range-Limited Force Evaluation Optimized for FPGAs." In: *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Vol. 2160. IEEE. 2019, pp. 263–271.

[34] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. "Evaluating and optimizing OpenCL kernels for High Performance Computing with FPGAs." In: *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2016, pp. 409–420.

[35] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. "High-performance high-order stencil computation on FPGAs using opencl." In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2018, pp. 123–130.

Copyright Notice: Material from: Jens Huthmann et al. "OpenMP Device Offloading to FPGAs Using the Nymble Infrastructure." In: *OpenMP: Portable Multi-Level Parallelism on Modern Systems*. Ed. by Kent Milfeld et al. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-58144-2.

14 EXTENDING HIGH-LEVEL SYNTHESIS WITH HIGH-PERFORMANCE COMPUTING PERFORMANCE VISUALIZATION

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *Extending High-Level Synthesis with High-Performance Computing Performance Visualization* by *Jens Huthmann, Artur Podobas, Lukas Sommer, Andreas Koch and Kentaro Sano* in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. The contribution of the author of this thesis is summarized as follows.

» Lukas Sommer was responsible for the conception and implementation of the OpenMP offloading frontend for the Nymble HLS compiler. He was also in charge of the integration of tracing into the OpenMP offloading runtime plugin and the post-processing of profiling data and their conversion into the Paraver format. The hardware modules and the architecture for the collection and efficient storage of profiling information in the HLS-generated FPGA accelerator were designed and implemented by Jens Huthmann, who was also responsible for the experiments and performance optimizations on the FPGA. The benchmark code was prepared by Jens Huthmann and Artur Podobas. The manuscript was joint work by all authors. «

ABSTRACT

The recent maturity in High-Level Synthesis (HLS) has renewed the interest of using Field-Programmable Gate-Arrays (FPGAs) to accelerate High-Performance Computing (HPC) applications. Today, several studies have shown performance- and power-benefits of using FPGAs compared to existing approaches for a number of application kernels with ample room for improvements. Unfortunately, modern HLS tools offer little support to gain clarity and insight regarding why a certain application behaves as it does on the FPGA, and most experts rely on intuition or abstract performance models.

In this work, we hypothesize that existing profiling and visualization tools used in the HPC domain are also usable for understanding performance on FPGAs. We extend an existing HLS tool-chain to support Paraver – a state-of-the-art visualization and profiling tool well-known in HPC. We describe how each of the events and states are collected, and empirically quantify its hardware overhead. Finally, we practically apply our contribution to two different applications, demonstrating

how the tool can be used to provide unique insights into application execution and how it can be used to guide optimizations.

14.1 INTRODUCTION

The past decades' pursuit for better and more productive High-Level Synthesis (HLS) tools has recently sparked a flurry of innovative research in using Field-Programmable Gate-Arrays (FPGAs) in High-Performance Computing (HPC). Here, FPGAs are being examined as alternative to traditional accelerators, and also to possibly mitigate the effects of Moore's law by providing a silicon substrate whose functionality changes through time. Today, several authors [22, 26, 28, 29] have already demonstrated performance- and/or power-consumption benefits of using FPGAs over server-class general-purpose processors, many-core accelerators (e.g. Xeon PHI), and graphics processing units (GPUs). However, while most works show empirical benefits of using HLS and FPGAs over alternative forms of computing, little clarity is provided into why a certain application performs as it does, and where opportunities to improve are.

High-Performance Computing has a long tradition of a diverse arsenal of profiling and visualization tools, particularly those aimed at understanding bottlenecks and limitations of high-performance applications. Profilers and visualizers have been developed for most parallel programming models, including both thread-based [17, 24] and task-based [21] models.

In this paper, we hypothesize that existing HPC visualization tools are sufficiently general to also be applied for understanding FPGA applications compiled through HLS. Confirming our hypothesis is important for several reasons: **(i)** it would provide the means to better understand the performance HLS tools yield, including identifying bottlenecks (e.g. memory-, compute- or latency-boundness), and **(ii)** FPGAs could be more seamlessly integrated into HPC tool infrastructures.

To test our hypothesis, we extend a state-of-the-art HLS tool to include modules that continuously monitor states and generate events based on the execution. Our HLS tool supports a sub-set of the OpenMP [7] 4.0 accelerator directives, allowing for multiple forms of parallelism (SISD, SIMD, MIMD) and also supports shared-memory synchronization (thread-level barriers and critical sections). Our methods are general and can be adopted for different HLS or visualization tools. We demonstrate our efforts targeting the Paraver tool-chain [24], which represents state-of-the-art in HPC profiling and visualization, and is actively used in a number of HPC centers to understand performance.

In this study we claim the following contributions:

- We describe in detail how to integrate support for an HPC profiling infrastructure into FPGA High-Level Synthesis flows, quantifying area and utilization as well as measuring the impact on performance
- Using two different applications, we demonstrate how our profiling infrastructure can be used to the understand performance of HLS-generated accelerators, showing step-by-step how to reason about and overcome the bottlenecks.

14.2 background and motivation

Understanding the performance of applications through visualization has long been an active research field in HPC. Some noteworthy visualization methodologies include Score-P [18], Vampir [17], and Paraver [24] (see the review by Isaacs et al. [16] for a complete list); these tools are heavily used to port and obtain performance in state-of-the-art HPC systems. Many of these tools work both on general-purpose processors (CPUs) and Graphics Processing Units (GPUs), and are used to provide intuitive understanding behind both application execution and performance.

The most common way of extending a particular programming model or framework (e.g. OpenMP [7]) to support trace-generation is to intercept API function calls and time-stamp the related activities. For example, when a thread is created (e.g. using `pthread_create`), a profiling library intercepts the API call and time-stamps the thread creation and saves it to a log. The log is later formatted and can be viewed using one of said visualization tools, which also provide a rich set of analyses. Due to the generality of modern processors, trace generation is often trivial to implement with low overhead, and much of the research focuses on other challenges, e.g. compression, and how to manage the often tens of GB's of trace-data.

The situation is different for HLS on FPGAs. Here, there are no easily available interception mechanism. While it is possible to add tracing mechanics into the code itself – as is done in CPU tracing – this can severely impact the hardware generation, for example increase initiation intervals (II) of loops through false dependencies, or impact memory behavior or overheads. Furthermore, monitoring stalls and memory-bandwidth is very hard since the application is not exposed to those signals. Ideally, one would change the HLS compiler itself to incorporate tracing and profiling. This option requires access to the source-code of HLS compilers, most of which are closed-source. Furthermore, the impact of fully supporting all features of an HLS generated pipeline from the perspective of hardware constructs known in HPC remains unknown and unmeasured. While trace-generation and profiling indeed are much more challenging on FPGAs than on general-purpose systems, there can also be more rewards. For

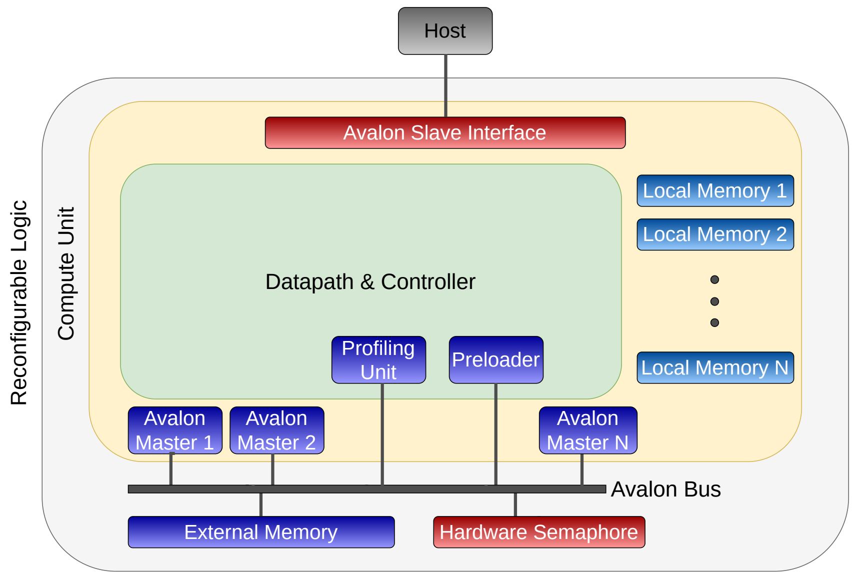


Figure 14.1: Overview of the architecture template, including the Datapath and Controller, generated by Nymble, and the integrated profiling unit. All components are connected to four DDR4-banks through the Avalon bus.

example, applications running on general-purpose systems are often oblivious of low-level architectural details happening on CPUs (and performance counters are often limited); on the other hand, FPGAs are fully aware over its hardware, and much more (and interesting) information can be obtained from its execution.

Our work aspires to unify performance visualization of FPGAs to that currently existing in HPC, in order to leverage the mature methodology available in HPC and also to help bridge (and hence popularize) the use of FPGAs in HPC. Our work, to the best of our knowledge, is the first effort to fully integrate an HPC visualization framework into HLS compilers in order to reason around performance and efficiency of the HLS-generated code.

14.3 NYMBLE HLS COMPILER

In this work, the ability to collect information for HPC performance visualization tools in HLS-generated FPGA-accelerators is integrated directly in an automated compile flow. The established academic HLS compiler *Nymble* [14] serves as basis for the compilation flow.

Nymble originally targeted Xilinx devices and was adapted to also target Intel FPGA boards for this work.

14.3.1 *Compilation Flow*

In earlier versions of Nymble, users had to use specialized, custom annotations (C/C++ pragmas) to mark regions of the application to be executed on the FPGA. The necessary data-transfers were automatically inferred by the compiler, pessimistically assuming that all data had to be transferred to the FPGA and back to the host after execution.

In order to make the Nymble HLS compiler more accessible for users from the HPC-domain, a new frontend was added which uses the OpenMP target offloading constructs instead of the custom, Nymble-specific pragmas. These constructs, standardized in version 4.0 and following the OpenMP standard, do not only allow to denote target regions with standardized annotations, but also allow users to clearly specify which and how data has to be transferred, avoiding unnecessary costly data transfers between CPU and FPGA memories.

With the new frontend, it is possible to use any C/C++-program with OpenMP annotations as input to the Nymble HLS compiler. The automated HLS flow will create an accelerator design for the target regions in the application (currently limited to one target region per application) as Verilog HDL code. Together with the architectural template shown in Fig. 14.1, this accelerator forms a complete FPGA-design that can be synthesized using the vendor's standard tools (Quartus in Intel's case).

As shown in Fig. 14.1, the generated accelerator has access to two different kinds of memory: Small, but fast local memories and the large external DRAM memory on the FPGA-board, which is also used to exchange data between host and FPGA-accelerator. Data-transfers are automatically handled as specified by the corresponding OpenMP clauses (`map`). The preloader can be used to efficiently pre-load data from the external memory to the local memory for faster access, and the hardware semaphore connected to the Avalon bus is used to handle OpenMP synchronization constructs (`critical` and `barrier`).

14.3.2 *Execution Model*

The execution inside the generated accelerator is organized according to a static schedule computed at synthesis time to determine the start times of individual operations.

However, for some operations, it is not possible to statically determine their operation delay. One example of such *variable-latency operations* (VLO) are (cached) memory accesses, which allow Nymble to access local (BRAM) and external (DRAM) memory. A second example of variable-latency operations are inner (nested) loops with statically unknown trip count: These nodes are embedded into the dataflow graph of the surrounding loop as a *single* operation node with statically unknown delay. At runtime, the execution of the outer loop's graph is paused during execution of the inner loop.

At synthesis time, the scheduler assumes the expected *minimum* delay for VLOs. To account for longer delays of VLOs during execution, the surrounding hardware execution needs to be suspended (stalled). The number of stalls, e.g., due to external memory accesses, is an important performance figure.

However, fitting each operation with dynamic control signals (e.g., a handshake) would be too expensive in terms of hardware resource consumption. The Nymble controller therefore orchestrates the execution at the granularity of *stages*, which contain all nodes active in a single pipeline stage of the datapath. Controlling the execution at the granularity of stages requires a smaller number of control signals and less controller logic, while still allowing to suspend the execution when a variable-latency operation, e.g., access to external DDR memory, exceeds the delay assumed during scheduling.

The unique feature of the extended Nymble-MT [15] execution model employed in this work is that the model allows for the *simultaneous* execution of multiple hardware threads. Different hardware threads can be active in different stages at the same time, significantly increasing the overall throughput and resource efficiency of the accelerator. With the new OpenMP-based frontend for the Nymble HLS compiler, parallel OpenMP constructs (e.g., `teams`, `teams distribute` or `parallel`) are directly mapped to parallel hardware threads executing simultaneously in the generated accelerator.

In contrast to the basic *C-slow execution model* presented by Leiserson et al. [19], Nymble-MT also uses *thread reordering* to allow faster threads to overtake slower threads during execution. To enable a stage for thread reordering, the stage must be able to hold the context (e.g., intermediate results) of all hardware threads for all operations contained in the stage. As this requires significant amounts of hardware resources, it is not reasonable to generally enable thread reordering for *all* stages. Instead, Nymble-MT selectively enables thread reordering only for those stages that actually contain variable-latency operations, whereas the other stages in between form a *static region*. In the reordering stages, a hardware thread scheduler (HTS) selects one available thread for execution as soon as the following stage becomes available. Huthmann et al. also presented more elaborate techniques to optimize the placement of reordering stages in [13], but this is out of scope for this work.

14.4 EXTENDING HIGH-LEVEL SYNTHESIS WITH HPC PROFILING SUPPORT IN PARAVER

In this work, we extend the Nymble HLS compiler to include the capability of sampling and monitoring states and events, captured in the format required by modern HPC visualization tools, by embedding hardware counters for profiling directly into the generated accelerator.

Although the concrete implementation of this work is specific to the Nymble compiler, the general methodology developed in this work is generic enough to be used with and integrated into the compilation flow of other academic (e.g., LegUp [3]) or industrial (e.g., Intel OpenCL [6], Xilinx Vivado HLS [27]) HLS tools. Our additions have negligible impact on the overall compile time.

In this section, we give a brief introduction to Paraver and why we chose it, before going deeper into how different metrics are implemented and collected inside our HLS tool-flow.

14.4.1 *Introduction to Paraver*

Paraver is a state-of-the-art visualization tool that brings clarity into how applications execute in HPC environments. The premise behind Paraver is that many bottlenecks in applications can easily be identified visually, such as how memory-bound the application is, or the degree of load-(im)balance across threads. Paraver visualizes the execution of actors (e.g., CPUs, Threads, MPI-ranks etc.) in a time-line view, where different colors or values represent the behavior of the thread.

Paraver supports three types of records: states, events, and communication. In this work, we have focused on supporting states and events, and excluded communication since they are primarily used for MPI communication, which for us is subject for future work in visualizing *multi-FPGA* execution.

14.4.2 *Hardware Collection of Paraver States and Events*

As shown in the architectural template diagram in Fig. 14.1, the profiling unit is integrated into the generated datapath and directly hooks-into and snoops all compute pipelines that compose the accelerator. The profiling unit is backed by the external memory, with the collected performance counters being periodically stored to external memory to avoid overflow of the counters. There they can later be accessed from the host for analysis.

As previously stated, Paraver supports three types of records, out of which we support two: states and events. A state describes the situation a thread is currently in, whereas events are, on the other hand, near-instantaneous measurements of a certain metric.

14.4.2.1 *State Recording*

States are useful in analyzing high-level details regarding the execution, such as whether the application is well-balanced (all threads contribute equally), or how many serialization points exist in the ap-

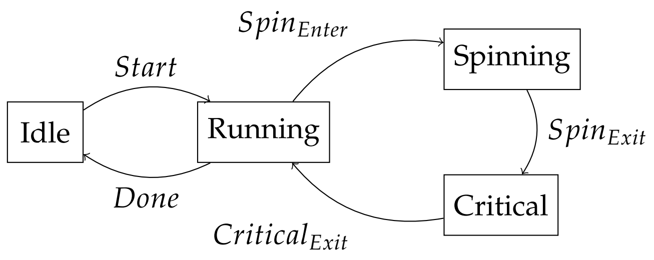

$$\begin{tikzcd}
\text{Idle} \arrow[r, bend left, "Start"] & \text{Running} \arrow[l, bend left, "Done"] \arrow[r, bend left, "Spin_{Enter}"] & \text{Spinning} \arrow[d, bend left, "Spin_{Exit}"] \\
& & \text{Critical} \arrow[ul, bend left, "Critical_{Exit}"]
\end{tikzcd}$$

Figure 14.2: The state transition diagram of our implementation for recording OpenMP critical sections. It includes profiling both the time spent in acquiring (spinning) on the lock as well as the time spent inside the protected section.

plication (the time spent in critical sections). In our implementation, threads can be in four different states, as shown in Fig. 14.2, and the state information is available per hardware thread. The two basic states are *Running* and *Idle*. The *running* state indicates that a user (or run-time system) has loaded a context and explicitly started the accelerator. The *idle* state indicates that there is no currently loaded context, and/or the previous context finished executing. Our representation of running/idle is identical to how it is used for CPUs and GPUs.

Next to that, Nymble supports OpenMP `#pragma omp critical` sections. In multi-threaded applications, critical sections are used to provide thread-safe access to data. The absence of critical sections can lead to race conditions, breaking the application. Since the timing of entering, exiting, and waiting for the lock associated with the critical section is important, these are recorded as a state of the pipeline. The state *Spinning* is used to express that a thread is currently waiting to enter the critical section, which is currently occupied by another thread. The state *Critical* is used to track the time a thread spends inside the critical section. As a thread cannot resume computation while waiting to enter a critical section, the time spent in state *Spinning* can be an important performance figure.

The current state for each thread is stored in a register. Because the state can change for multiple threads at once, each time at least one thread changes it state, we record the current state for all threads together with the current clock count. Each state is represented as a 2-bit value: 00 for idle, 01 for running, 10 for critical, and 11 for spinning. The size of each state record is $2 * N_{threads} + 32$ bits wide. Each record is saved into a buffer; when the buffer is nearly full, the buffer is flushed to the external memory, and resumes operations. Currently, the width of the buffer is equal to the data-width of the external memory controller (512-bit), but can be tuned with other sizes.

14.4.2.2 *Event recording*

In general-purpose processors, events are often recorded by performance counters, and accessed through tools such as PAPI [20]. They include metrics such as memory-bandwidth (GB/s), compute performance (e.g. FLOP/s), or how often resources stall (% stall cycles). Implementation-wise, events are often collected for a certain time (the sampling period), and are then time-stamped and saved at periodic intervals.

In our methodology, we extended the HLS compiler to automatically insert support for collecting metrics as events. Recording an event is different from recording a state, as a threads' state can change at any time during execution, while an event is only acquired at periodic intervals.

For each of the supported events, we added a performance counter module to the accelerator. As we need to aggregate values from multiple sources (stages, operations, and others), this module has two inputs for each source. The event to be recorded from that source, and a condition if the value is valid. In each clock cycle, all valid values are added to the running aggregate. All aggregated events are periodically flushed to external memory. This period is user-adjustable, and is a proxy over fine-grained information that is required, but is also subject to a larger tracer – the higher the period, the more data is produced. Next, we will introduce the different metrics we support and how they are collected.

STALLS As described in Section 14.3.2, the Nymble-MT execution model supports variable-latency operations. If the execution of VLOs exceeds the minimum latency assumed during static scheduling, the pipeline is stalled.

Stalls in HLS environments are different from stalls in a CPU or GPU. While a CPU only has a single pipeline that can stall (a binary metric, the program counter stalls), in a Nymble accelerator every stage that contains a variable-latency operation can stall (a compile-time known discrete maximal value). In our implementation, a stall can happen due to one of two things: **(i)** a memory access takes longer than expected, or **(ii)** a resource (e.g., memory port) is shared across many threads and arbitration leads to a stall.

In this work, stalls are collected as events and the collection of stalls is implemented by snooping the control signals. For visualization purposes, it would be impractical to show per-pipeline-stage stalls, as these may occur in large numbers. Instead, we argue that an aggregated per-thread stall event is more useful from a visual perspective.

A high number of stalls during execution can be an important hint that the applications performance on the FPGA is limited by the execution of variable-latency operations, e.g., due to latency when accessing external memory, or caused by contention when entering a critical section, and can guide optimization of the application.

COMPUTE PERFORMANCE Compute performance in Nymble can be classified as two types: floating-point and integer performance. Each compute-stage in the pipeline has a number of both integer- and floating-point operations scheduled onto it, determined at compile-time. By snooping the control-bus associated with each compute-stage, we can watch the value of the per-stage activation signal to determine whether the arithmetic units in the stage are active. We can then track the number of active units over time to measure the complete performance. We collect the compute-performance as an event, where each threads' compute performance is aggregated and sampled during an execution time window.

If the profiled compute performance falls short of the expected performance, this can be an indicator that the accelerator is not able to supply the arithmetic operators with data due to excessive memory access latency. Preloading data from external DRAM memory to local BRAM memory can help to improve the overall compute performance.

MEMORY PERFORMANCE For inserting the performance counters for memory accesses, there are multiple options: The counters could be placed directly at each memory operation in the pipeline, or they could be placed in the Avalon memory interface of the CU. All memory operations in the pipeline are multiplexed to one Avalon read- and one Avalon write port per thread. Therefore, we decided to place the memory performance counters in the central Avalon interface and evaluate the memory requests coming from the operators, as this reduces the footprint of the memory performance counters.

Tracking the memory bandwidth by collecting the memory requests coming from the operators to the Avalon interface incurs a small time skew. However, to get rid of this skew, we would additionally need to track memory responses, which would significantly increase the footprint of the profiling infrastructure.

Information about the memory throughput of the application over time can provide insights about the application's memory access pattern. Often, replacing many accesses to single data-items with a single read of multiple data-elements (e.g., a submatrix) into fast local memory can significantly improve the memory bandwidth and, in turn, the overall application performance.

14.5 RESULTS

14.5.1 *Experimental Methodology*

All experiments were carried out using a Stratix 10 SX-280HN2F43E2VG FPGA on an Intel D5005 PAC card using the Quartus 19.2 software. The OpenMP frontend for Nymble is based on LLVM version 9.

All performance measurements were captured using the performance counters described in the previous section. For visualization we used Paraver 4.8.2. Today, Paraver does not support the notion of *cycles*. For all cases in the graphs where microseconds are used, these are in fact cycles.

We used eight *simultaneous* threads in a single accelerator. More than eight threads in a single accelerator did not increase the performance further, because at this point all computing resources are filled. Adding more threads only increases congestion. Instantiating another accelerator would be possible, but is out of scope for this work.

14.5.2 *Profiling Overhead and Hardware Footprint*

The inclusion of the necessary infrastructure for the state- and event tracing of course incurs a small hardware overhead. For our first case-study, investigation of the post-P&R results shows that the inclusion of our tracing infrastructure increases the number of registers by at most 5.4% (geo.-mean 2.41%) and the number of ALMs by at most 4% (geo.-mean 3.42%), so the footprint of the tracing infrastructure is comparably small. The impact on the operating frequency of the accelerator is negligible (maximum degradation 8 MHz at 140 MHz).

A direct comparison of the different performance counters shows that each of the counters contributes similarly to the hardware overhead, none of the counters could be identified as being remarkably expensive.

For our second study, the number of registers is increased by 1.3% and the number of ALMs by 1.5%. The impact on the operating frequency of the accelerator is even smaller with a degradation of only 1 MHz at 148 MHz.

14.5.3 *Case-study: Matrix Multiplication*

To illustrate how our methodology can be used to guide optimizations and reason about performance of applications compiled with HLS, we turn our attention to one of the perhaps most important computational kernels: the general matrix multiplication (GEMM). In order to demonstrate our methodology, we consider matrices with size 512x512.

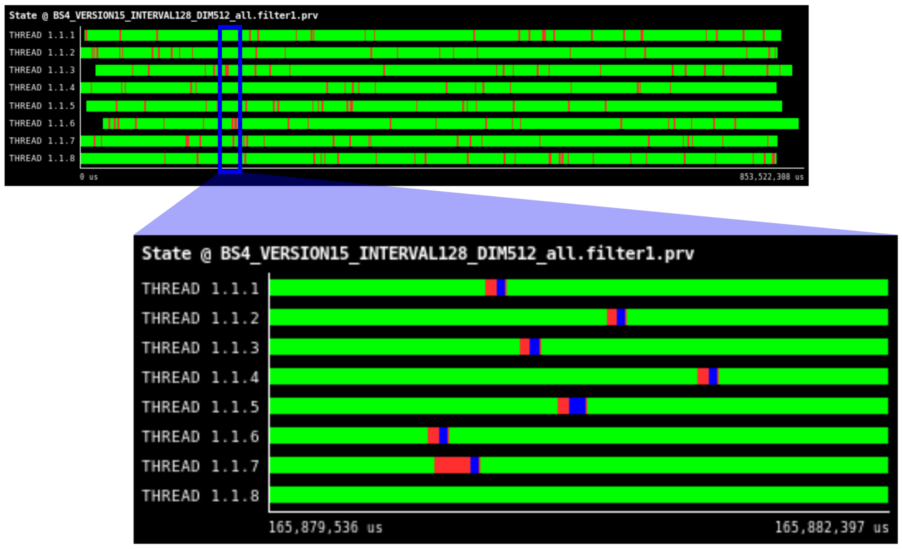


Figure 14.3: Paraver state-view showing the state each of the executing hardware threads as a function of the execution time: Green represents a running state, Red represents spinning on a mutex, Blue represents atomic execution inside critical sections, and Black represents an idle thread.

The starting point of our case-study is a naive implementation, as shown in Fig. 14.4a. The version is a text-book example of a GEMM implementation, and has been extended to use OpenMP threads to parallelize the innermost loop, while protecting the update to the `C` matrix with a critical section.

We start by profiling our initial GEMM version. This version, for a matrix of 512x512 elements, takes 853,522,308 cycles to execute. We see the visualization for this version in Figure 14.3 (top); while threads are mostly running without interference, we do notice that there is a fair amount (1.54 %) of time spent in critical sections and spinning on locks (1.57 %). This – in effect – extends the serial portion of the code, limiting the parallelism (according to Amdahl's law [1]), which is shown in detail when zooming into the trace (Figure 14.3 (bottom)): thread 7 is spinning on the lock that protects the critical section that thread 6 is currently in. We also see that the memory throughput is quite low (Figure 14.5) throughout the entire execution.

In the second version (called: *No Critical Sections*), we distribute the work differently, effectively forcing threads to work on elements in the output matrix `C` in isolation, removing the need to protect it. This is a minor (and fairly non-intrusive) change in application code, but removes all critical section- and spin-states, enabling the application to fully execute in parallel (only green states, not shown due to space-limitations) and be memory-bound. We can see in Figure 14.5 that this version has a slightly better memory throughput, which overall yields a net improvement of 1.14x in execution time over the original version with minor improvements to obtained bandwidth.

```
void matmul(DTYPE* A,
            DTYPE* B,
            DTYPE* C,
            int DIM){
  #pragma omp target parallel map(from:C[0:DIM*DIM])\
    map(to:A[0:DIM*DIM], B[0:DIM*DIM]) num_threads(8)
  {
    int my_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    for(int i=0; i < DIM; ++i){
      for(int j=0; j < DIM; ++j){
        DTYPE sum = 0;
        for(int k=my_id; k < DIM; k += num_threads){
          sum += A[i*DIM+k] * B[k*DIM+j];
        }

        #pragma omp critical
        {
          C[i*DIM + j] = sum;
        }
      }
    }
  }
}
```

(a) Simple version of GEMM.

```
void matmul(DTYPE* A,
            DTYPE* B,
            DTYPE* C,
            int DIM){
  #pragma omp target parallel map(from:C[0:DIM*DIM])\
    map(to:A[0:DIM*DIM], B[0:DIM*DIM]) num_threads(8)
  {
    int my_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    for(int i=0; i < DIM; i += num_threads){
      for(int j=0; j < DIM; ++j){
        DTYPE sum = 0;
        for(int k=my_id; k < DIM; k += VECTOR_LEN){
          VECTOR vA = *((VECTOR*) &A[(i*DIM) + k]);
          #pragma unroll VECTOR_LEN
          for(int v = k; v < k + VECTOR_LEN; ++v){
            sum += va[v-k] * B[v * DIM + j];
          }
        }

        #pragma omp critical
        {
          C[i*DIM + j] = sum;
        }
      }
    }
  }
}
```

(b) Vectorized version of GEMM.

```
void matmul(DTYPE* A,
            DTYPE* B,
            DTYPE* C,
            int DIM){
  #pragma omp target parallel map(from:C[0:DIM*DIM])\
    map(to:A[0:DIM*DIM], B[0:DIM*DIM]) num_threads(8)
  {
    int my_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    for(int i= my_id * BLOCK_SIZE; i < DIM; i += num_threads * BLOCK_SIZE){
      for(int j=0; j < DIM; j += BLOCK_SIZE){
        VECTOR C_local[BLOCK_SIZE];
        for(int k = 0, buffer = 0; k < (DIM+BS); k+=BLOCK_SIZE, ++buffer){
          VECTOR A_local[BUFFER_SIZE][BLOCK_SIZE];
          VECTOR B_local[BUFFER_SIZE][BLOCK_SIZE];
          if(k < DIM){
            for(int m=0; m < BLOCK_SIZE; ++m){
              A_local[buffer%BUFFER_SIZE][m] = *((VECTOR*) &A[(i+m)*DIM+k]);
              B_local[buffer%BUFFER_SIZE][m] = *((VECTOR*) &B[(k+m)*DIM+j]);
            }
          }

          if(buffer != 0){
            for(int x=0; x < BLOCK_SIZE; ++x){
              #pragma unroll BLOCK_SIZE
              for(int y=0; y < BLOCK_SIZE; ++y){
                DTYPE sum = C_local[i][j] * (buffer == 1 ? 0.0f : 1.0f);
                #pragma unroll BLOCK_SIZE
                for(int v=0; v < BLOCK_SIZE; ++v){
                  sum += (A_local[(buffer-1)%BUFFER_SIZE][x][v]
                          * B_local[(buffer-1)%BUFFER_SIZE][v][j]);
                }
                C_local[x][y] = sum;
              }
            }
          }
        }

        for(int k=0; k < BLOCK_SIZE; ++k){
          *((VECTOR*) &C[(i+k)*DIM+j]) += C_local[k];
        }
      }
    }
  }
}
```

(c) Double buffering version of GEMM.

Figure 14.4: Different versions of GEMM.

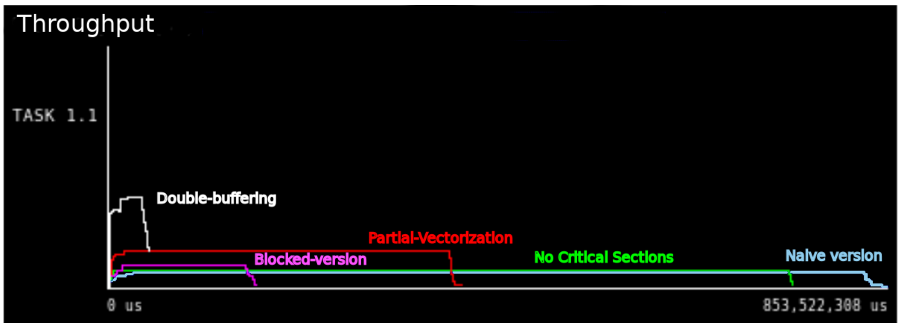


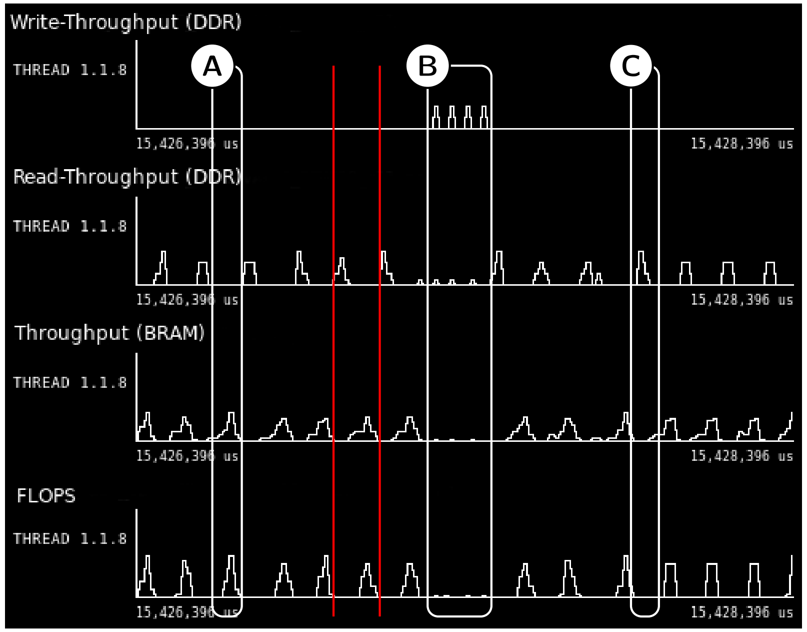
Figure 14.5: Visual comparison between the different GEMM versions and their relative bandwidth over their respective execution time (absolute metrics Paraver views are not included for space reasons).

Being memory-bound, in the third version (called: *Partial Vectorization*) (Fig. 14.4b) we partially vectorize our application. Here we vectorize the loading of the matrix `A`, and still have `B` non-vectorized (since it needs to be transposed to allow similar vectorization). The vectorization width is 128-bit, which leads to a better memory performance, which can be observed in Figure 14.5, yielding the expected improvement in achieved memory throughput with wider accesses. This also materialize in a 1.93x faster execution time over the previous version.

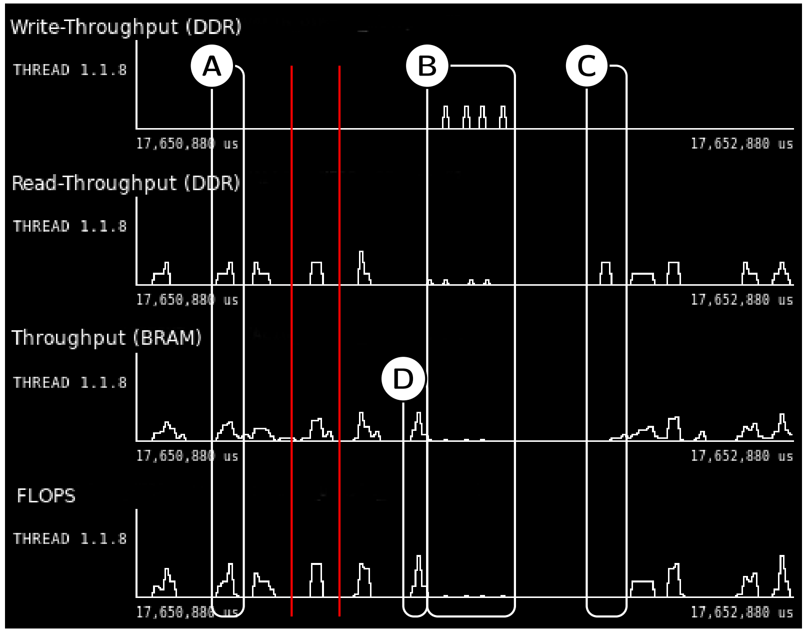
The fourth version (called: *Blocked version*), we take a step back and try to apply a commonly used method: blocking the algorithm. Blocking should be an effective strategy, as it trades the expensive external memory operations for better, high-bandwidth, on-chip BRAM bandwidth. Blocking is often used for CPUs [12], GPUs [10], and even FPGAs [8]. Blocking consists of two stages: **(i)** we first load the sub-matrices into local BRAMs, and **(ii)** we compute on the local sub-matrices. Furthermore, we vectorized the computation on the sub-matrices, leading to better compute performance.

These two discrete stages can be seen when visualizing the throughput and compute-performance (FLOP/s) trace in Figure 14.6a– we see the distinct compute phases as "spikes" in the execution trace, which are dependent on the loading of data (the upper throughput trace). The blocked version gives us a net performance improvement of 5.28x over the initial version. Interestingly, we also notice that this version yields a lower external memory throughput compared to our previous partial vectorized version (Figure 14.5). The reason for this is that we trade much of the external memory bandwidth for local memory bandwidth.

In the final version (called: *double-buffering)*, we remedy the problem of distinct load- and compute-phases. Instead, we re-write our application such that a thread performs the pre-load the *next* iterations' blocks while computing on the blocks currently available. The code for this version is seen in Fig. 14.4c. We can see the impact on performance



(a) Figure showing the impact of blocking on the Paraver graph and between two iterations of the matrix compute loop (the red line). We see that we compute only on data stored in local memory (**A**), which is followed by writing the local data back to external storage (**B**), and finally loading the new block from external memory into local storage (**C**).



(b) Figure showing the profiling double-buffering in the Paraver graph. Best contrasted against the results in Figure 14.6a. We now see that the external memory is being read from to prefetch the next block, concurrently with computing the matrix multiplication on the current local block (**A**), except for the final iteration where no prefetching is done, and only computation is seen (**D**). Finally, we also see that segment **C** is working on a different iteration (relative to the red lines, which separate individual iterations) than in Figure 14.6a, which is also due to the improved double-buffering scheme. Segment **B** remains identical between the two figures.

Figure 14.6: Impact of GEMM optimizations on Paraver graph.

of this optimization in Figure 14.6b, and we clearly see a different and more beneficial behavior compared to Figure 14.6a: the sequential read and compute behavior is removed, as the memory accesses and compute are now running in parallel. This also has a beneficial impact on achieved external memory throughput, where it reaches the highest performance out of our GEMM candidates (Figure 14.5). Overall, this version is 19x faster than our initial version.

14.5.4 *Case study: Infinite series for π calculation*

In the second case study, we show how problems with the scaling of algorithms can be analyzed using the state-view of Paraver. For this, we use an infinite series for calculating π (source shown in Fig. 14.7), which we distribute onto multiple threads. The sum-reduction of the individual results is done using a critical section. Figures 14.8, 14.9, and 14.10 show the Paraver state traces for 1, 4, and 10 million iterations respectively. For 1,000,000 iterations, the hardware only achieves 0.146 GFLOP/s. From the trace, it can be seen that the overhead of starting the individual threads by the software causes the earliest threads to be finished before last ones are even started. If we increase the iteration count to 4,000,000, all threads are starting to get executed in parallel, resulting in an increased performance of 0.556 GFLOP/s. Further increasing the number of iterations to 10,000,000 gives us a performance of 1.507 GFLOP/s. Unfortunately, since we are using only single-precision computation, further increasing the number of iterations results in numerical instability. Ignoring the instability, increasing the number of iterations to $15 \cdot 10^9$ would give us 36.84 GFLOP/s.

From this case study we can see that the bottleneck for small computational loads is located in the communication between the software and hardware, so we will look into how we can improve our interface.

14.6 RELATED WORK

The SoCLog [23] platform provides real-time acquisition of profiling data for FPGA-based System-on-Chips. The primary metric of collection is activity and the authors visualize said activity on a time-line. They show-case their work on DCT application, showing how different traces of two versions (one un-optimized and one optimized) differs in their framework. Compared to our work, which also captures activity (IDLE/RUNNING), we also incorporate many more metrics to provide a more truthful and informative picture of the execution. Curreri et al. [5] profile FPGA throughput between producer and consumer in HLS applications. The authors visualize the performance as a DAG where nodes represents producers and consumers, and edges data communication (throughput), showing both absolute and rela-

```
DTYPE pi (int steps, int threads){
  DTYPE final_sum= 0.0;
  DTYPE step = 1.0/(DTYPE) steps;
  #pragma omp target parallel map(to : step) \
    map(tofrom: final_sum) num_threads(threads)
  {
    int step_per_thread= steps/omp_get_num_threads();
    int start_i = omp_get_thread_num()*step_per_thread;
    VECTOR sum = {0.0f};
    DTYPE local_step= step;
    for (int i=0; i< step_per_thread; i+=BS_compute) {
      #pragma unroll BS_compute
      for(int j=0; j < BS_compute; j++) {
        DTYPE x = ((DTYPE)(i+start_i+j)+0.5f)*local_step;
        sum[j] += 4.0f / (1.0f+x*x);
      }
    }
    #pragma omp critical
    for(int i=0;i<BS_compute;i++) {
     final_sum+= sum[i];
    }
  }
  return final_sum;
}
```

Figure 14.7: Infinite series for π calculation, distributed across multiple threads using OpenMP.

tive performance. Compared to our work, we aggregate and abstract throughput performance in both off- and on-chip memory, showing it per-thread and per-FPGA, rather than per-consumer/producer, and we are more consistent with how HPC application display these metrics.

Additionally, we support more metrics than only throughput. Podobas et al. [25] used Paraver to reason about and demonstrate the effect of resource-sharing and arbitration on load-(im)balance in multi-threaded FPGAs similar to our work in this paper. This work extends their work by including a much richer and complex set of events. Calagar et al. [2] researched source-level profiling using their tool Inspect, which links the generated hardware (Verilog) to source-level (C/C++) constructs in order to provide understanding around what hardware is generated, and what is happening through a familiar `gdb`-like interface. Our work focuses more on the visualization of real-time performance, but in the future could also benefit from linking traces with source-level annotations in a way similar to what they proposed. The OmpSs [9] task-based eco-system supports offloading OpenMP tasks to FPGA accelerators [11], and the authors also support Paraver trace-generation during FPGA execution. Their work focuses mostly on visualization how the host processor orchestrates FPGA execution, e.g., memory transfers to-and-from the FPGA itself rather than what happens *inside* the accelerator (as this work does). Curreri et al. [4]

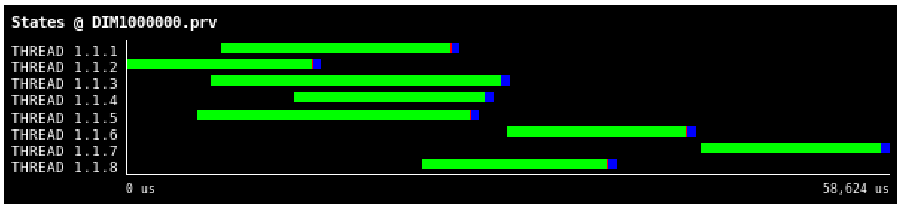


Figure 14.8: Paraver state-view (colors explained in Fig. 14.2) for π with 1 million iterations divided onto 8 threads. Here it can be seen that not all threads are executing simultaneously, resulting in a performance of 0.146 GFLOP/s.

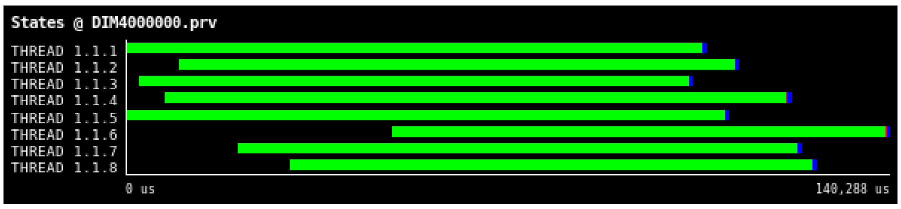


Figure 14.9: Paraver state-view (colors explained in Fig. 14.2) for π with 4 million iterations divided onto 8 threads. Compared to Fig. 14.8, the threads are now running more and more in parallel, resulting in a performance of 0.556 GFLOP/s.

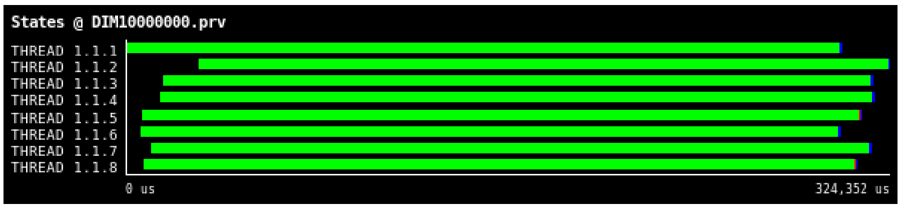


Figure 14.10: Paraver state-view (colors explained in Fig. 14.2) for π with 10 million iterations divided onto 8 threads. Compared to Fig. 14.9, most of the time is spent running all threads, resulting in a performance of 1.507 GFLOP/s.

provides a general methodology for profiling High-Level Languages (essentially HLS) for FPGAs, and create a hardware module capable of capturing performance metrics. They demonstrate their framework on a molecular dynamics application in Impulse C. Our work extends this by collecting many more diverse metrics, and we are also not limited to streams, but we also include threads. Both Intel and Xilinx offer visualization tools and profiling information for various parts of their respective HLS flow, including reporting performance bottlenecks (e.g., high initiation intervals) and what memory interfaces were synthesized for what operation (e.g., coalesced, burst, etc.), often linking performance problems back to source code. Our work extends both of these by focusing on profiling dynamic execution in a multi-threaded environment, where we also include concepts of threads, critical sections, and achievable throughput.

14.7 CONCLUSIONS

In this work, we have developed a profiling infrastructure with OpenMP-based frontend that can be included directly into an HLS toolflow that produces traces that can be used with state-of-the-art performance visualization tools. The infrastructure was included into the Nymble HLS compiler and used with the Paraver visualization tool, but is sufficiently general to be included in other HLS tools and to be used with other HPC visualization tools.

Using two different applications, we demonstrated how the performance visualization can be used to precisely analyze performance bottlenecks, and successfully optimize the performance by restructuring the HLS input code.

In the future, we plan to extend our infrastructure for communication between FPGAs in a multi-FPGA setup and evaluate how the collected traces can be used for profile-guided optimization in the HLS compiler.

REFERENCES

[1] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities." In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[2] Nazanin Calagar, Stephen D Brown, and Jason H Anderson. "Source-level debugging for FPGA high-level synthesis." In: *2014 24th international conference on field programmable logic and applications (FPL)*. IEEE. 2014, pp. 1–8.

[3] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. "LegUp: high-level synthesis for FPGA-based processor/accelerator systems." In: *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. 2011, pp. 33–36.

[4] John Curreri, Seth Koehler, Alan D George, Brian Holland, and Rafael Garcia. "Performance analysis framework for high-level language applications in reconfigurable computing." In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 3.1 (2010), pp. 1–23.

[5] John Curreri, Greg Stitt, and Alan George. "Communication visualization for bottleneck detection of high-level synthesis applications." In: *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. 2012, pp. 33–36.

[6] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. "From OpenCL to high-performance hardware on FPGAs." In: *22nd international conference on field programmable logic and applications (FPL)*. IEEE. 2012, pp. 531–534.

[7] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[8] Yong Dou, Stamatis Vassiliadis, Georgi Krasimirov Kuzmanov, and Georgi Nedeltchev Gaydadjiev. "64-bit floating-point FPGA matrix multiplication." In: *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. 2005, pp. 86–95.

[9] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. "Ompss: a proposal for programming heterogeneous multi-core architectures." In: *Parallel processing letters* 21.02 (2011), pp. 173–193.

[10] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication." In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2004, pp. 133–137.

[11] Antonio Filgueras, Eduard Gil, Daniel Jimenez-Gonzalez, Carlos Alvarez, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees Vissers. "OmpSs Zynq all-programmable SoC ecosystem." In: *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. 2014, pp. 137–146.

[12] Kazushige Goto and Robert Van De Geijn. "High-performance implementation of the level-3 BLAS." In: *ACM Transactions on Mathematical Software (TOMS)* 35.1 (2008), pp. 1–14.

[13] J. Huthmann and A. Koch. "Optimized high-level synthesis of SMT multi-threaded hardware accelerators." In: *2015 International Conference on Field Programmable Technology (FPT)*. 2015, pp. 176–183. DOI: 10.1109/FPT.2015.7393145.

[14] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch. "Hardware/Software Co-Compilation with the Nymble System." In: *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. July 2013, pp. 1–8. DOI: 10.1109/ReCoSoC.2013.6581538.

[15] J. Huthmann, J. Oppermann, and A. Koch. "Automatic high-level synthesis of multi-threaded hardware accelerators." In: Sept. 2014, pp. 1–4. DOI: 10.1109/FPL.2014.6927432.

[16] Katherine E Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. "State of the Art of Performance Visualization." In: *EuroVis (STARs)*. 2014.

[17] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. "The vampir performance analysis tool-set." In: *Tools for high performance computing*. Springer, 2008, pp. 139–155.

[18] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir." In: *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[19] Charles E Leiserson, Flavio M Rose, and James B Saxe. "Optimizing synchronous circuitry by retiming (preliminary version)." In: *Third Caltech conference on very large scale integration*. Springer. 1983, pp. 87–116.

[20] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. "PAPI: A portable interface to hardware performance counters." In: *Proceedings of the department of defense HPCMP users group conference*. Vol. 710. 1999.

[21] Ananya Muddukrishna, Peter A Jonsson, Artur Podobas, and Mats Brorsson. "Grain graphs: OpenMP performance analysis made easy." In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2016, pp. 1–13.

[22] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC." In: *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE. 2016, pp. 77–84.

[23] Ioannis Parnassos, Panagiotis Skrimponis, Georgios Zindros, and Nikolaos Bellas. "SoCLog: A real-time, automatically generated logging and profiling mechanism for FPGA-based Systems On Chip." In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2016, pp. 1–4.

[24] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. "Paraver: A tool to visualize and analyze parallel code." In: *Proceedings of WoTUG-18: transputer and occam developments*. Vol. 44. 1. Citeseer. 1995, pp. 17–31.

[25] Artur Podobas and Mats Brorsson. "Empowering openmp with automatically generated hardware." In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE. 2016, pp. 245–252.

[26] Artur Podobas and Satoshi Matsuoka. "Designing and accelerating spiking neural networks using OpenCL for FPGAs." In: *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE. 2017, pp. 255–258.

[27] Xilinx. "Vivado Design Suite User Guide: High-Level Synthesis." In: 2 (2018).

[28] Chen Yang, Tong Geng, Tianqi Wang, Rushi Patel, Qingqing Xiong, Ahmed Sanaullah, Chunshu Wu, Jiayi Sheng, Charles Lin, Vipin Sachdeva, et al. "Fully integrated FPGA molecular dynamics simulations." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–31.

[29] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. "Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL." In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2018, pp. 153–162.

Part IV

ACCELERATED SUM-PRODUCT NETWORK INFERENCE

15 AUTOMATIC MAPPING OF THE SUM-PRODUCT NETWORK INFERENCE PROBLEM TO FPGA-BASED ACCELERATORS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-Based Accelerators* by *Lukas Sommer, Julian Oppermann, Alejandro Molina, Kristian Kersting and Andreas Koch* in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. The contribution of the author of this thesis is summarized as follows.

» As the corresponding and leading author, Lukas Sommer was responsible for the design of the hardware accelerator architecture and the implementation of all its hardware components. He was also in charge of the conception and implementation of the automatic toolflow, the scheduler was contributed by Julian Oppermann. The example Sum-Product Networks used as benchmarks were prepared by Alejandro Molina. The experiments on CPU and GPU were conducted by Alejandro Molina, the experiments on the FPGA were done by Lukas Sommer. The manuscript was joint work by all authors. «

ABSTRACT

In recent years, FPGAs have been successfully employed for the implementation of efficient, application-specific accelerators for a wide range of machine learning tasks. In this work, we consider probabilistic models, namely, (Mixed) Sum-Product Networks (SPN), a deep architecture that can provide tractable inference for multivariate distributions over mixed data-sources. We develop a fully pipelined FPGA accelerator architecture, including a pipelined interface to external memory, for the inference in (mixed) SPNs. To meet the precision constraints of SPNs, all computations are conducted using double-precision floating point arithmetic. Starting from an input description, the custom FPGA-accelerator is synthesized fully automatically by our tool flow. To the best of our knowledge, this work is the first approach to offload the SPN inference problem to FPGA-based accelerators. Our evaluation shows that the SPN inference problem benefits from offloading to our pipelined FPGA accelerator architecture.

15.1 INTRODUCTION

The computational demand of many deep learning approaches, the currently predominating branch of machine learning (ML), can only be satisfied by specialized accelerators. Besides GPUs, which provide massive parallelism for regular computations, and custom processors such as Google's TPU [14], which supports operations typical for ML now, but is unable to adapt to advances in ML algorithms, FPGAs have shown promise as an energy-efficient-yet-flexible alternative [19]. Microsoft is the most prominent advocate of the FPGA-centric strategy and has widely deployed its Catapult [9] expansion boards across its data centers. While initially being used to improve the quality of results of its Bing search engine, Microsoft recently announced Project Brainwave [5], which includes an FPGA-based processor tailored to accelerate deep neural networks. Amazon and Baidu have also started to employ FPGAs in their products and services [11].

A truly intelligent system, however, should be able to deal with uncertain inputs (e.g. missing features) as well as express its uncertainty over outputs. It is, therefore, no surprise that probabilistic approaches have recently gained tremendous momentum within deep learning. Corresponding approaches such as variational autoencoders, deep generative models, and generative adversarial nets (GANs), however, have limited capabilities when it comes to *probabilistic inference*. Consider, e.g. implicit likelihood models like GANs [13]. Even when successful in capturing the data distribution, they do not allow to compute the probability of a test sample. In contrast, Sum-Product Networks (SPNs) [21] are a deep architecture that permit *exact* and *efficient* probabilistic inference. More precisely, they can compute any marginalization and conditioning query in time linear to the model's representation size, by evaluating its computational graph representation—consisting of computational nodes (addition & multiplication) and distribution nodes—in a bottom-up fashion. This computational structure and the need for efficient processing of batches of queries in different AI application scenarios make SPNs a promising candidate for FPGA-based acceleration.

Thus, the goal of the present paper is to develop a new, pipelined FPGA accelerator architecture for the sum-product network inference problem. This opens the door to local (on-chip) model inference and in future work, even model learning. The dynamic range of the probabilities and the need to represent values with very small magnitude make the use of high-precision floating-point arithmetic necessary. Besides the pipelined accelerator architecture, we also develop an automatic synthesis flow for SPNs. Our tool flow is a turn-key solution that not only encompasses the actual synthesis of a hardware datapath from an SPN description but also provides crucial parts for any practically-usable FPGA accelerator, i.e., a high-throughput memory

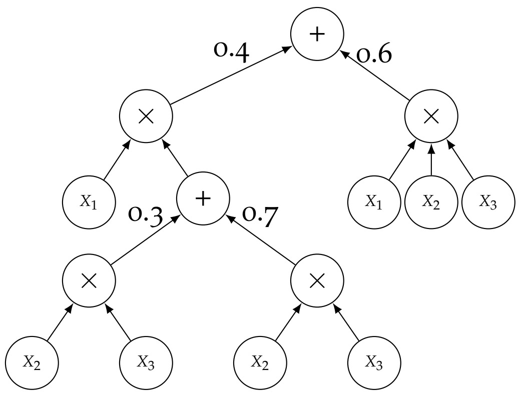


Figure 15.1: An example of a valid SPN. Here, x_1, x_2 and x_3 are random variables. The structure represents the joint distribution $P(x_1, x_2, x_3)$.

system and accompanying software APIs. In our experimental evaluation, we show that our pipelined architecture can handle different model sizes and still provide high throughput which is an essential feature of scalable inference. To the best of our knowledge, this is the first approach to offload the SPN inference problem to FPGA-based accelerators.

We proceed as follows. Section 15.2 briefly reviews SPNs, how they are learned from training data and how inference works. Afterwards, we touch upon existing approaches to accelerate probabilistic graphical models on various architectures. Section 15.4 then presents our pipelined accelerator architecture, our automatic tool flow, and the integration into a heterogeneous system. In section Section 15.5 we evaluate our accelerator and compare it to other architectures and section Section 15.6 concludes the paper and looks forward to future work.

15.2 SUM-PRODUCT NETWORKS

Probabilistic Graphical Models (PGMs) have had a broad impact on machine learning, both in academia and industry. They can solve many ML problems by simply answering probabilistic queries. Consider, e.g., predictive modeling. One trains a PGM which then answers probabilistic queries; for multi-class classification answering the query $\arg\max_c P(\text{class} = c|\text{data})$ gives us the most likely class according to our model. Alternatively, as in one of our benchmark datasets, we can ask which is the most likely plant to grow in a particular location: $\arg\max_p P(\text{plant} = p|\text{location})$. Unfortunately, inference in unrestricted PGMs is often intractable. Motivated by the importance of efficient inference for large-scale applications, a substantial amount of work has been devoted to learning probabilistic models for which inference is guaranteed to be tractable. Examples of these model classes include sum-product networks (SPNs), hinge-loss Markov random fields, and tractable higher-order potentials. Being instances of Arithmetic Circuits (ACs), see [3] for a discussion, SPNs are a deep architecture that can represent high-treewidth models [25] and facilitate *exact* inference for a range of queries in time *linear* in the network size [2, 21]. They inherit universal approximation properties from mixture models—a mixture model is simply a "shallow" SPN with a single sum node. Consequently, SPNs can represent any prediction function, very much like deep neural networks. Having exact probabilities offers an advantage not present in other PGMs and deep neural networks. One can compare the probabilities computed by different models and not only solve classification or regression problems, but also do anomaly detection at the same time while taking into account the statistical nature of the data. Furthermore, SPNs can compute measures such as Entropy, Mutual Information, Information Gain, etc. Moreover, the Mixed Sum Product Network (MSPN[1]) proposed by Molina *et al.* [18], is a non-parametric version of SPNs that opens the door for an efficient FPGA implementation based on histograms.

15.2.1 *Definition of SPNs*

Formally, an SPN is a rooted directed acyclic graph, comprising *sum*, *product*, and *leaf* nodes as seen in Fig. 15.1. The scope of an SPN is the set of random variables appearing on the network. An SPN can be defined recursively as follows: (1) a tractable univariate distribution is an SPN; (2) a product of SPNs defined over different scopes is an SPN; and (3), a convex combination of SPNs over the same scope is an SPN. Thus, a product node in an SPN represents a factorization over independent distributions defined over different random variables, while a sum node stands for a mixture of distributions defined over the same variables. From this definition, it follows that the joint distribution modeled by such an SPN is a valid probability distribution, i.e., each complete and partial evidence inference query produces a consistent probability value [20, 21]. Computationally, the number of arithmetic operations is different for the given nodes. For product nodes, we have $|\text{children}| - 1$ number of multiplications. For sum nodes, we have $|\text{children}| - 1$ number of additions and $|\text{children}|$ number of multiplications. The leave nodes require as many operations needed for a look-up table of size $|\text{domain}(\text{Variable}_i)|$. The SPNs make no restriction on reusing sub-structures as long as the consistency rules are preserved.

15.2.2 *Tractable Inference in SPNs*

To answer probabilistic queries in an SPN, we evaluate the nodes starting at the leaves. Given some evidence, the probability output

1 github.com/alejandromolinaml/SPFlow

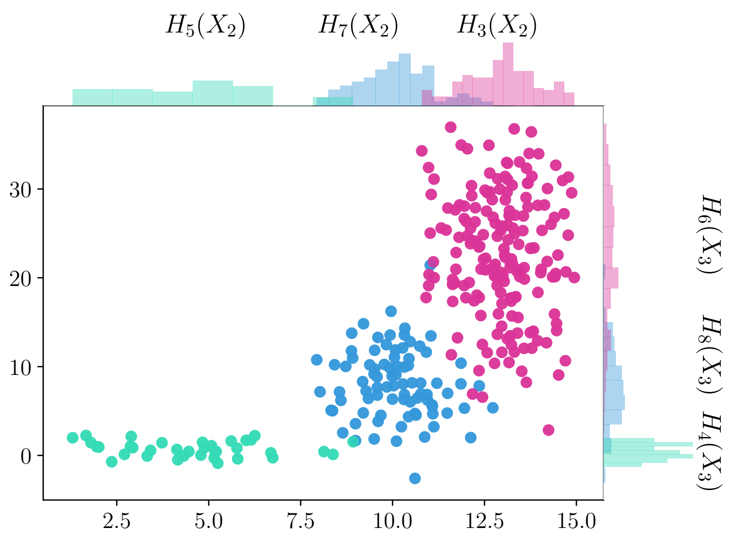


Figure 15.2: Example of a synthetic dataset, fitted using the SPN in Fig. 15.4. For illustration purposes, we present here $P(X_2, X3)$ marginalizing X_1 out. $H_i(X_j)$ represent the different histograms in the MSPN. The histograms are free to overlap, but the greedy learning algorithm attempts to represent different subsets of the data. Deeper MSPNs can represent more complex datasets.

of querying leaf distributions is propagated bottom up. For product nodes, the values of the child nodes are multiplied and propagated to their parents. For sum nodes, instead, we sum the weighted values of the child nodes. The value at the root indicates the probability of the asked query. To compute marginals, i.e., the probability of partial configurations, we set the probability at the leaves for those variables to 1 and then proceed as before. An example of such marginalization is shown in Fig. 15.2, where we obtain a new SPN that computes $P(X_2, X_3) = \sum_x P(X_1 = x, X_2, X_3)$ and show how it can be used to fit a randomly generated dataset. All these operations traverse the tree at most twice and therefore can be achieved in linear time w.r.t. the size of the SPN. In this work, we consider only datasets with discrete variables, as they can be easily implemented with fast look-up tables, however extending the histograms to the continuous case is not difficult. We also focus on joint computations as they are the basis for all other inference algorithms. These operations at the leaves are then executed in constant time, maintaining the tractability of the SPN.

15.2.3 *Learning SPNs*

While it is possible to craft the structure of a valid SPN by hand, doing so requires domain knowledge and weight learning afterward [21]. Here, we use the greedy, top-down approach of the MSPNs that directly learns both the structure and weights of (tree) SPNs at once while making few assumptions on the data.

It consists of three steps: (1) base case, (2) decomposition and (3) conditioning. In the base case, if only one variable remains, the algorithm learns a univariate normalized histogram and terminates. Here, we represent univariate distributions by normalized histograms. These histograms are then converted to look-up tables, as they can be efficiently implemented on FPGAs. We use the same implementation for the traditional CPU code to keep the experiments as similar as possible. In the decomposition step, it tries to partition the variables into independent components. This decomposition is based on a non-parametric independency test [16] that is run for every random variable $V_j \subset \mathbf{V}$ in a pair-wise fashion, creating a graph of interactions among all the variables. We then obtain the disconnected components of this graph, which indicates that variables inside a component are tightly coupled and variables among different components are independent. From these components, we induce a product node such that $P(\mathbf{V}) = \prod_j P\left(V_j\right)$ and recurse on each child.

If the base case is not applicable and the decomposition step cannot find independencies, then the algorithm partitions the training samples into clusters (conditioning). This clustering procedure first transforms the data to a higher-dimensional space so that discrete variables fit the assumptions of normality of the clustering algorithm. From the clusters, we induce a sum node, and the algorithm recurses on each cluster. The weights of the sum nodes then represent the data proportions in the clusters. This learning algorithm does not reuse sub-structures, however, the intermediate language representation used keeps track of sub-tree references and the FPGA pipeline is aware of them. Using a different algorithm or a pruning or compression step can enable this capability.

This learning algorithm is typically pre-computed on a traditional CPU. Then the resulting SPN structure can be compiled for inference into FPGAs, C++ code or even TensorFlow graphs, as shown in section 15.5.

15.2.4 *Size of SPNs*

The size of the SPN depends on the training dataset and the learning parameters. The smallest multi-variate SPN is a *product* node over all the random variables, giving a lower bound on the network size of $|\text{Variables}| + 1$. Introducing binary *sum* nodes doubles the size of the sub-graph by the number of random variables in the node. The learning algorithm then creates SPNs whose size is constrained by heuristics for early stopping and the number of independencies recovered from the data. Deeper SPNs are more expressive, but also more computationally intensive, while shallower SPNs have fewer parameters and use fewer resources. Controlling the depth of the SPN impacts how well it fits the data. A very deep structure tends to overfit, while a shallow structure does not have enough expressive power to represent the training data.

In this work, we focus on the largest SPNs that we can fit entirely in FPGAs in a fully spatial realization.

15.3 RELATED WORK

To the best of our knowledge, our work is the first automatic synthesis tool to accelerate SPN inference on FPGAs.

Previous research has studied the FPGA acceleration of other kinds of PGMs such as Bayesian Networks (BN) [1] and Markov Random Fields (MRF) [4]. These approaches are orthogonal to ours, as the inference problem, in general, is not tractable for BNs and MRFs. A common theme in BN accelerators is to design specialized processors, as inference in large tree-width models is expensive. In contrast, the arithmetic circuit (AC) representation of BNs [6] resembles a datapath similar to an SPN, which Zermani *et al.* [24] first compiled to C code and then used Vivado HLS to synthesize an IP core to be used on a Zynq SoC. Other ACs implementations for FPGAs can be found in ([8], [12]). However, ACs are restricted to binary random variables, whereas our SPN-based approach has no such restriction. Lastly, the recently presented LibSPN [22] aims to bring multi-core and GPU acceleration to SPNs by translating them to TensorFlow [10], but is not yet publicly available and does not support histogram-based representation at the leaf-nodes. For evaluation purposes, we implemented our own TensorFlow-based GPU backend.

15.4 APPROACH & IMPLEMENTATION

The goal of this work is to synthesize efficient accelerators for the inference problem in sum-product networks. In this section, we present our accelerator architecture, the compilation flow that generates the accelerator using the SPN structure learned as described in Section 15.2.3 as input and the integration of our accelerator into a heterogeneous system.

15.4.1 *System Overview*

In this paper, we aim for a fully pipelined accelerator architecture since we want to process a large number of queries efficiently and each query is independent of other queries.

The vast amount of data required for the input values of each query makes it inevitable to provide the accelerator with access to the external memory on the FPGA board. We use the open-source

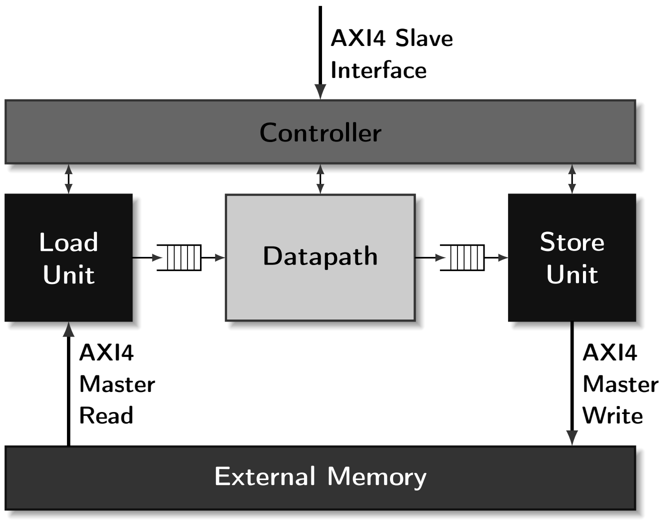


Figure 15.3: Overview of the system architecture.

TaPaSCo framework [15], which provides a standardized memory interface as well as an interface and host-side API for the integration into a heterogeneous system (cf. Section 15.4.2).

Our accelerator architecture consists of four main components, depicted in Fig. 15.3. The controller is responsible for managing the other components and provides the necessary TaPaSCo slave interface (Section 15.4.2). Load and store unit together make up the memory interface of the accelerator, described in more detail in Section 15.4.4. The datapath implements the computation represented by the sum-product network, see Section 15.4.3 for details on its construction. Load unit, datapath, and store unit are decoupled by queues to allow for the latency-insensitive, independent operation of the different components.

15.4.2 *TaPaSCo & Heterogeneous system integration*

In order to integrate our accelerator into a heterogeneous system, we use the TaPaSCo-framework [15] that has been developed to fast-track the prototyping of FPGA-based accelerators. TaPaSCo defines AXI-based, standardized interfaces, which are used to control the execution of the accelerator (slave interface) and provide the accelerator with memory access (master interface). TaPaSCo also includes an automated tool-flow to assemble multiple instances of these accelerators (*processing elements*, PEs) into a complete top-level design, called *threadpool*, which is then wrapped with the connectivity to host and memory. Regardless of its composition, every threadpool can be controlled by an unified software interface, the so-called TaPaSCO-API. This API provides basic functions to transfer data to/from the device and launch jobs on the accelerators in the threadpool.

To integrate our accelerator into the threadpool, we implement the necessary interfaces. Our controller (cf. Fig. 15.3) implements an AXI4

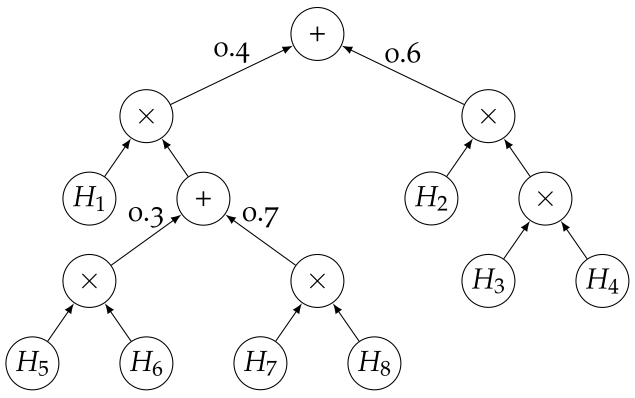


Figure 15.4: Intermediate representation of the SPN. Operators with n inputs are split into balanced trees of operators with 2 inputs. Univariate distributions are modelled by histograms H_i.

slave interface, which is used by TaPaSCo to transmit commands (e.g. the start-command) from the host by reading/writing configuration register values and control signals. Load and store unit in combination implement an AXI4 master interface, which is connected to the external memory on the FPGA board through the TaPaSCo infrastructure. The implementation of this interface is described in more detail in Section 15.4.4. The host can transfer data to/from the external memory on the device by using the TaPaSCo API. Before the execution on the FPGA starts, all sample data is transferred to the external memory on the FPGA board. After the computation has completed, all result data is transferred back to host memory.

We also use the API to control the execution of our accelerator in the FPGA. This allows us to seamlessly integrate the offloading to the FPGA accelerator into the host software for SPN inference.

15.4.3 *Compile flow & Datapath architecture*

Our automatic tool-flow that maps SPNs to FPGA accelerators starts from a textual representation of the SPN, which describes the nodes of the SPN, their individual configuration, and the connections between the different nodes in the tree-structure of the SPN.

In a first step, we parse the textual input and construct a graph-based intermediate representation of the SPN, as shown in Fig. 15.4 for the example SPN from Fig. 15.1. During the construction of the graph IR, we decompose additions or multiplications with more than two operands into balanced trees of two-input operators, as can be seen for the three-input multiplication on the right side of the example SPN.

As mentioned earlier in Section 15.2.3, the random variables with univariate distributions at the leaf nodes of the SPN tree-structure are modeled with histograms. Each histogram consists of multiple bins with corresponding probabilities, and the bins match adjacent intervals of input feature values. If two neighboring bins share the same probability value, our tool-flow merges these bins, resulting in a bin which covers the two adjacent intervals of the original bins.

The configuration of the histograms is also part of another optimization implemented in our tool-flow: The sample values stored in memory are used to index into the bins of the different histograms. Based on the largest possible value for the input features, which is determined by the upper bound of the last bin, we can optimize the input bitwidth. Across all histograms, we can calculate the minimum number of bits n, necessary to represent the highest possible input value, at compile time. All input values are then stored in memory using n bits. This optimization reduces the pressure on the memory interface, as less bandwidth is required to read the input values for each sample from memory. If, for example, the highest possible input values across all histograms was 212, we could represent all input values with only eight bit, reducing the memory bandwith requirement by a factor of four compared to a generic representation with 32 bit.

Before we map the tree to hardware operators, we decompose weighted adders into corresponding combinations of multipliers and adders. In the next step, we now map the SPN tree structure to the actual hardware datapath. As we aim for maximum throughput in this first version of our synthesis tool, we use a fully-spatial, statically scheduled microarchitecture for our datapaths. That is, every operation in the SPN is implemented by its own operator module on the device, and no resource-sharing occurs. For the scheduling of the fully-spatial microarchitecture, a simple as-soon-as-possible strategy suffices to obtain an optimal (concerning the latency) static schedule for the datapath. Our pipelined schedule assumes that the memory interface can provide an input sample in every clock cycle. However, depending on the number of input features and the bitwidth of the memory interface (cf. Section 15.4.4), this might not be the case. We therefore introduce a shift register into our datapath, which keeps track of valid samples in the pipeline to make sure we only write back correct output values to memory.

For the pipelined implementation of the histograms at the leaf-nodes of the SPN, we use a simple look-up table approach. The values for each index are computed at compile time, and the discrete values of the input features are used to index the look-up table. The arithmetic operations inside the tree (additions and multiplications) are realized with pipelined operators from the FloPoCo-library [7]. Regarding the precision, a worst case scenario arises from the inference of a low-probability instance such as an anomaly or an outlier data point evaluated on a shallow SPN consisting of one product over all the univariate features. Furthermore, the number of discrete values or bins in the look-up table can have a negative impact as

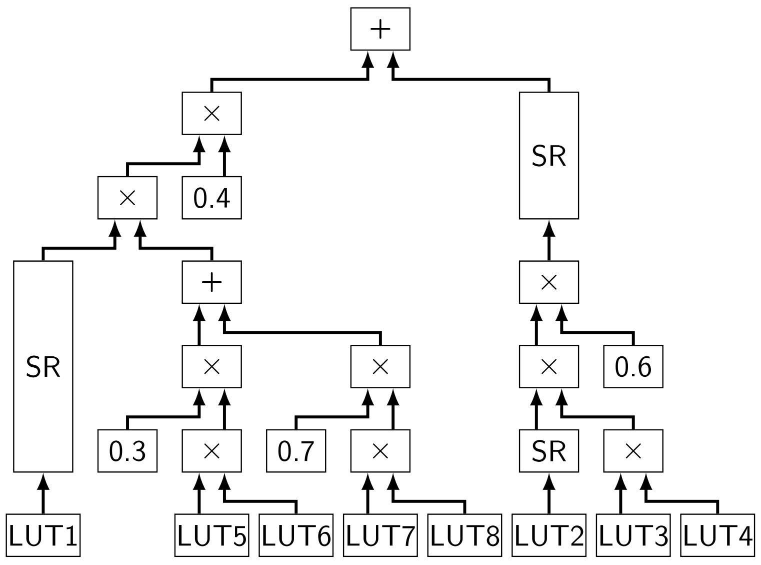


Figure 15.5: Example SPN mapped to HW operators. To balance the different branches of the tree, some intermediate results need to be preserved in shift registers. Look-up tables for the histograms are indexed with the input feature values read from memory.

they represent a normalized distribution. This has a lower bound of $\prod_{i=0}^{\#\text{features}} \min(\text{histogram}_i(x))$. To represent these values, which have a very small magnitude, the whole computation within the datapath is completely done in the FloPoCo floating point format, using 11 bits for the exponent and 52 bits for the mantissa. Aside from subnormal numbers, this format is equivalent to a double-precision floating-point format. The final conversion to IEEE-754 double precision format for use in the host is done right before the values are written back to memory. In practice, the SPNs used in the experimental section did not suffer underflows, however, bigger SPNs might have to do computations in log-space. The size of the SPNs is limited to the FPGA capabilities and implementing exp and log functions, would reduce the size of the SPNs that we could implement on-chip.

The mapping to the hardware datapath for the example from Fig. 15.1 is given in Fig. 15.5. The height of the operators in the diagram indicates their scheduled start time, with the first time step at the bottom. All look-up tables used to implement the histograms are placed in the first time step and will be indexed by the discrete values for the three input variables read from memory. As one can see from the IR-graph in Fig. 15.4, the different branches of the tree have different heights/latencies. We need to balance the different branches to match partial results from the different branches at the merge points. To this end, we insert pipelined shift registers into the datapath that work as intermediate storages for partial results (labeled *SR* in the diagram).

15.4.4 *Memory Interface*

As explained in Section 15.4.2, TaPaSCo provides each processing element, such as our accelerator, with a standardized AXI4-interface which we use to connect our accelerator to the external memory on the FPGA-board.

With our fully pipelined accelerator architecture, the goal is to feed the pipeline with a new query every clock cycle. This requires an efficient, pipelined load infrastructure, in particular as the number of bits at the input of the datapath is typically higher than at the output (due to the tree-structure of SPNs). For an efficient supply of input data, we use AXI4 burst requests to read the query input values from memory. The load unit computes the addresses for the requests, relative to a base address configurable from the host. The independence of the different AXI4 channels allows us to issue the next burst request before all data from the current one has been sent, resulting in a continuous stream of input data.

Inside the load unit, a re-alignment unit converts from the AXI4 data width (e.g. 512 bit in case of our evaluation on the VC709 board) to the input width of the datapath. Here, we have to consider three different cases: If the bitwidth of a sample matches the bitwidth of the memory interface, we can simply forward the read data. If the input bitwidth of the datapath is smaller than the memory interface bitwidth, the realignment unit buffers the read data and forwards chunks of appropriate size to the datapath. In case the bitwidth at the input of the datapath exceeds the maximum bitwidth of the memory interface, multiple beats must be buffered, before a complete sample can be forwarded to the datapath. In all three cases, the realignment unit was designed to forward a complete sample to the datapath as soon as a sufficient amount of data was read/buffered. In the latter case, the bitwidth of the memory interface limits the performance of our accelerator, and we cannot start the computation for a new sample in every clock cycle. After the computation inside the datapath has completed, we need to store the results back to memory. Here we buffer multiple output values and again use AXI4 burst requests, which can be handled more efficiently by the memory interface, to write back a batch of values.

15.5 EXPERIMENTAL EVALUATION

15.5.1 *Datasets*

For the performance evaluation, we focused on datasets of count and binary data types. For count data we used the NIPS[2] corpus, containing 1,500 documents over the 100 most frequent words. For binary

2 archive.ics.uci.edu/ml/datasets/bag+of+words

data we evaluated our implementation on a range of six different datasets as pre-processed and provided by [17] and [23]. *Accidents* is a dataset of traffic accidents in Belgium for the period 1991-2000. The *Audio* dataset consists of information about users that listened or did not listen to a set of top 100 artists. The *Netflix* dataset is a random subset of the Netflix challenge, focused on the 100 most frequently rated movies and whether a user rated a movie. The anonymized *MSNBC* data contains information about whether a user visited a top-level MSNBC page during a particular browsing session. The National Long Term Care Survey (NLTCS) dataset contains variables that measure whether a person can perform a set of daily living activities. The *Plants* dataset indicates whether a given plant can be found in a particular location.

15.5.2 *FPGA implementation*

For the evaluation of our FPGA implementation, we target a Xilinx VC709 evaluation board, containing a Virtex7-device (xc7vx690) and 4 GiB of RAM. The heterogeneous system that we use for the performance evaluation of the FPGA in Section 15.5.4 combines the FPGA-board with an Intel i7 6700K CPU. We used Xilinx Vivado 2017.4 for the FPGA implementation with a target frequency of 200 MHz. The results are given in Table 15.1. Column *Cols* gives the number of inputs to the datapath, *Adds* and *Muls* give the number of two-input adders and multipliers in the datapath and *Depth* indicates the depth of our computation pipeline. Besides the achieved clock frequency, we also state the resource requirements on the FPGA, for brevity those numbers are given relative to the entire FPGA in percent[3]. We encounter a relatively low demand for BRAMs and a moderate usage of registers. For the other numbers, the resource requirements roughly correspond to the size of the network. This can be seen from the NIPS-examples in particular: Here, we add an increasing number of input features (i.e. words) to the SPN, resulting in a bigger network and thus in a higher resource consumption.

Most of the examples achieve the target frequency, but in some examples the routing to the DSP-blocks used for the realization of the floating-point multiplication in FloPoCo becomes critical. We were able to improve the routing by adding one or two more pipeline stages to the multiplier; however, we see a notable degradation of the operating frequency for *NIPS80* and *Netflix*.

3 The absolute number of resources available are 433200 (LUT), 866400 (Register), 108300 (Slices), 1470 (BRAM) and 3600 (DSP), respectively.

Table 15.1: FPGA implementation results, all numbers are post-place&route.

Dataset	Cols	Adds	Muls	Depth	Freq. (MHz)	LUT (%)	Reg (%)	Slices (%)	BRAM (%)	DSP (%)
Accidents	111	27	217	99	200	61.49	32.92	75.87	4.66	36.17
Audio	100	12	275	99	200	72.26	37.65	89.32	4.66	45.83
Netflix	100	11	231	115	190	67.60	36.12	87.84	4.66	38.50
MSNBC200	17	30	165	245	200	51.64	27.24	66.01	4.66	27.50
MSNBC300	17	17	102	163	200	36.82	21.76	46.52	4.66	17.00
NLTCS	16	27	152	150	200	47.91	26.72	61.99	4.66	25.28
Plants	69	14	256	206	200	68.89	34.88	86.80	4.66	42.67
NIPS5	5	1	10	38	200	17.96	8.87	24.12	4.80	1.67
NIPS10	10	3	25	76	200	21.20	11.01	29.55	4.80	4.33
NIPS20	20	7	56	82	200	28.84	14.67	38.93	5.27	9.67
NIPS30	30	10	87	94	199	36.47	18.31	46.27	4.86	14.83
NIPS40	40	16	122	94	200	44.79	23.67	57.67	5.14	21.17
NIPS50	50	16	143	100	200	51.78	25.06	62.97	5.48	24.50
NIPS60	60	13	156	94	200	54.71	27.84	68.15	5.88	26.67
NIPS70	70	14	180	106	200	61.97	28.90	75.58	5.61	30.50
NIPS80	80	32	265	143	180	79.72	38.99	96.17	5.95	44.17

Number of LUTs, FFs, BRAM slices and DSP blocks are given as percentage for brevity. The total number of available resources are 433200, 866400, 1470 and 3600 respectively.

15.5.3 *CPU & GPU Implementation*

To have a complete performance comparison, we compiled the same SPNs used in the FPGAs to both C++ and TensorFlow [10]. Both implementations were executed on a Linux workstation with an AMD Ryzen 1950X Processor, 128GB of RAM and an NVIDIA 1080Ti GPU with 11GB of memory. We implemented the C++ version via code generation, writing inlined functions with look-up tables for the leaves. The rest of the SPN was expressed as a single function of additions and multiplications of the leaf functions. We compiled the generated C++ source code using GCC 7.2.0 and the flag -O3 and created two versions, one with the flag -ffast-math enabled, called **CPUF**, and one without it, called **CPU**. Both C++ implementations load all the data in memory and evaluate each complete dataset 1000 times. We report the average time for each instance of the dataset. For the **GPU** version, we implemented a TensorFlow graph of additions and multiplications. For the leaves, we used a look-up table implemented as a *tf.gather* operation over placeholders containing the data. We then executed each complete dataset 1000 times and measured execution times including data-transfer to the GPU, just as for the heterogeneous system with the FPGA.

15.5.4 *Performance evaluation*

To compare the performance of our FPGA implementation with the CPU and GPU, we report two different execution times for the FPGA: One only for the actual SPN computation, measured using a performance counter (**Cycle Counter**) inside the accelerator, with the corresponding actual time shown as (**FPGAC**). The second time (**FPGA**) is measured on the host side and includes the time for data transfer to/from the device memory and the launch of HW-execution. The performance results are given in Table 15.2, *Rows* gives the number of samples processed for each example. Besides the execution time in microseconds, we also report the throughput in samples per microsecond (e.g. **T-GPU**).

The GPU performance clearly shows that the naive TensorFlow parallelization model is not very suitable for the tree-structure of the SPNs. The analysis of the traces shows that most lanes are only used for a few operations and are idle most of the time. Additionally, a lot of inter-lane communication takes place, as the computation of the tree is spread across multiple lanes. This a general problem of the TensorFlow GPU-model, which is tailored more towards neural networks, and not linked to a specific GPU. The comparison of **CPU** and **CPUF** shows that, except for the two smallest networks, the CPU execution profits from the compilation with the *fastmath*-flag. Comparing the performance of CPU and the FPGA performance counter, one can

Table 15.2: Performance comparison. Best end-to-end throughputs (T), excluding the cycle counter measurements, are denoted bold.

Dataset	**Rows**	**CPU** (μs)	**T-CPU** (rows/ μs)	**CPUF** (μs)	**T-CPUF** (rows/ μs)	**GPU** (μs)	**T-GPU** (rows/ μs)	**FPGA Cycle Counter**	**FPGAC** (μs)	**T-FPGAC** (rows/ μs)	**FPGA** (μs)	**T-FPGA** (rows/ μs)
Accidents	17009	2798.27	6.08	2162.59	7.87	63090.94	0.27	17249	86.25	197.22	696.00	**24.44**
Audio	20000	4271.78	4.68	3683.71	5.43	78253.46	0.26	20317	101.59	196.88	761.00	**26.28**
Netflix	20000	4892.22	4.09	4098.88	4.88	67172.39	0.30	20322	106.95	187.00	654.00	**30.58**
MSNBC200	388434	15476.05	25.10	12713.55	30.55	62349.42	6.23	388900	1944.50	199.76	5008.00	**77.56**
MSNBC300	388434	10060.78	38.61	9418.29	41.24	50558.06	7.68	388810	1944.05	199.81	4933.00	**78.74**
NLTCS	21574	791.80	27.25	687.25	31.39	35544.39	0.61	21904	109.52	196.99	566.00	**38.12**
Plants	23215	3621.71	6.41	3521.04	6.59	67004.41	0.35	23592	117.96	196.80	778.00	**29.84**
NIPS5	10000	25.11	**398.31**	26.37	379.23	8210.32	1.22	10236	51.18	195.39	337.30	29.65
NIPS10	10000	83.60	**119.61**	84.39	118.49	11550.82	0.87	10279	51.40	194.57	464.30	21.54
NIPS20	10000	191.30	52.27	182.73	**54.72**	18689.04	0.54	10285	51.43	194.46	543.60	18.40
NIPS30	10000	387.61	25.80	349.84	**28.58**	25355.93	0.39	10308	51.80	193.06	592.30	16.88
NIPS40	10000	551.64	18.13	471.26	**21.22**	30820.49	0.32	10306	51.53	194.06	632.20	15.82
NIPS50	10000	812.44	12.31	792.13	12.62	36355.60	0.28	10559	52.80	189.41	720.60	**13.88**
NIPS60	10000	1046.38	9.56	662.53	**15.09**	40778.36	0.25	12271	61.36	162.99	799.20	12.51
NIPS70	10000	1148.17	8.71	1134.80	8.81	46759.26	0.21	14022	70.11	142.63	858.60	**11.65**
NIPS80	10000	1556.99	6.42	1277.81	7.83	63217.99	0.16	14275	78.51	127.37	961.80	**10.40**

see that, aside from NIPS5, the pipelined computation in the FPGA outperforms the CPU implementation regarding execution time and throughput. These values also demonstrate the effectiveness of our pipelining: Especially for the cases with binary input data, we are able to achieve an almost perfect pipelining, where we process a sample per clock cycle (equivalent to a throughput of 200 samples (rows) per μs). From the NIPS-examples, one can also see that the accelerator is memory bound: With a larger number of input values, more data has to be loaded from memory, and the pipeline cannot be fed every clock cycle.

The comparison of the performance counter and the total FPGA execution time including interaction in the heterogeneous system shows that there is significant overhead for data transfers to/from FPGA memory and the HW-launch. However, the FPGA is still able to outperform the CPU for all binary examples and most of the larger NIPS-examples (NIPS50, NIPS70, NIPS80), demonstrating the potential of offloading the inference in SPNs to the FPGA, in particular for larger SPNs. Note that for platforms having true shared memory between the CPU and the FPGA, such as the Xilinx Zynq devices, or the Intel HARP2 systems, these explicit data transfers between CPU and FPGA memory can be completely avoided and the full speed-up (based on the **FPGAC** measurement) realized.

15.6 CONCLUSION AND FUTURE WORK

We have presented the first FPGA-based accelerator architecture for the inference problem in sum-product networks (SPNs), a deep architecture for probability distributions. Our automatic synthesis flow generates a fully-pipelined accelerator from an input description of the SPN and also provides a software interface for interaction with the accelerator in a heterogeneous system. The accelerator architecture features pipelined access to the external memory on the FPGA-board and double-precision floating-point computation. The results of our experimental evaluation demonstrate that the pipelined computation in the FPGA can outperform CPU- and TensorFlow-GPU-implementations, almost processing a complete input sample per cycle for many examples.

There are several interesting avenues for future work. One could extend our synthesis flow and hardware implementation for resource-sharing of operators, in order to be able to map bigger networks. One could also further optimize the arithmetic operators, e.g., by using log-space computations, a very common arithmetic optimization in the ML-domain. Another interesting research avenue is pre-compiling a randomly generated structure and doing weight optimization in the FPGA, with the aim of having a full implementation of a PGM in a chip. If the random structure is large enough it could be retrained on different domains to fit any kind of data. This could also account for domains with concept drifting. Finally, other potential usage scenarios should be explored, such as computing *mutual information*, *maximum a posteriori estimation* and *approximate queries* within databases. These scenarios require fast processing of many input combinations but require less data-transfer from host to FPGA.

REFERENCES

[1] JD Alves, JF Ferreira, J Lobo, and J Dias. "Brief survey on computational solutions for Bayesian inference." In: *Workshop on Unconventional computing for Bayesian inference*. 2015.

[2] Jessa Bekker, Jesse Davis, Arthur Choi, Adnan Darwiche, and Guy Van den Broeck. "Tractable Learning for Complex Probability Queries." In: *Proc. of NIPS*. 2015.

[3] Arthur Choi and Adnan Darwiche. "On Relaxing Determinism in Arithmetic Circuits." In: *Proceedings of ICML*. 2017, pp. 825–833.

[4] Jungwook Choi and Rob A. Rutenbar. "Video-Rate Stereo Matching Using Markov Random Field TRW-S Inference on a Hybrid CPU+FPGA Computing Platform." In: *IEEE Trans. Circuits Syst. Video Techn.* (2016).

[5] Eric Chung, Jeremy Fowers, et al. "Accelerating Persistent Neural Networks at Datacenter Scale." In: *Hot Chips 29: A Symposium on High-Performance Chips*. 2017.

[6] Adnan Darwiche. "A differential approach to inference in Bayesian networks." In: *J. ACM* 50.3 (2003), pp. 280–305.

[7] Florent de Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo." In: *IEEE Design & Test of Computers* (July 2011).

[8] Pouya Dormiani, David Omoto, Pavan Adharapurapu, and Milos D Ercegovac. "A design of online scheme for evaluation of multinomials." In: *Advanced Signal Processing Algorithms, Architectures, and Implementations XV*. Vol. 5910. International Society for Optics and Photonics. 2005.

[9] Adrian M. Caulfield *et al.* "A cloud-scale acceleration architecture." In: *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. 2016.

[10] Martín Abadi *et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[11] Karl Freund. *Microsoft: FPGA Wins Versus Google TPUs For AI*. https://www.forbes.com/sites/moorinsights/2017/08/28/microsoft-fpga-wins-versus-google-tpus-for-ai. Accessed April 5, 2018. 2017.

[12] Johannes Geist, Kristin Y Rozier, and Johann Schumann. "Runtime observer pairs and Bayesian network reasoners on-board FPGAs: flight-certifiable system health management for embedded systems." In: *International Conference on Runtime Verification*. Springer. 2014, pp. 215–230.

[13] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generative Adversarial Nets." In: *Proceedings of NIPS*. 2014, pp. 2672–2680.

[14] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In: *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017*. ACM, 2017, pp. 1–12.

[15] J. Korinth, D. d. l. Chevallerie, and A. Koch. "An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures." In: *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. May 2015. URL: https://git.esa.informatik.tu-darmstadt.de/tapasco/tapasco.

[16] David Lopez-Paz, Philipp Hennig, and Bernhard Schölkopf. "The randomized dependence coefficient." In: *Advances in neural information processing systems*. 2013, pp. 1–9.

[17] Daniel Lowd and Jesse Davis. "Learning Markov network structure with decision trees." In: *Data Mining (ICDM), 2010 IEEE 10th International Conference on*. IEEE. 2010, pp. 334–343.

[18] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Sriraam Natarajan, Floriana Esposito, and Kristian Kersting. "Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains." In: (2018).

[19] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, et al. "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017*. ACM, 2017, pp. 5–14.

[20] Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro Domingos. "On Theoretical Properties of Sum-Product Networks." In: *Proc. of AISTATS*. 2015.

[21] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: a New Deep Architecture." In: *Proc. of UAI* (2011).

[22] Andrzej Pronobis, Avinash Ranganath, and Rajesh Rao. "LibSPN: A Library for Learning and Inference with Sum-Product Networks and TensorFlow." In: *Principled Approaches to Deep Learning Workshop*. 2017.

[23] Jan Van Haaren and Jesse Davis. "Markov Network Structure Learning: A Randomized Feature Generation Approach." In: *AAAI*. 2012, pp. 1148–1154.

[24] Sara Zermani, Catherine Dezan, Hanen Chenini, Reinhardt Euler, and Jean-Philippe Diguet. "FPGA implementation of Bayesian network inference for an embedded diagnosis." In: *2015 IEEE Conference on Prognostics and Health Management, ICPHM 2015*. IEEE, 2015, pp. 1–10.

[25] Han Zhao, Mazen Melibari, and Pascal Poupart. "On the Relationship between Sum-Product Networks and Bayesian Networks." In: *Proc. of ICML*. 2015.

16 HIGH-THROUGHPUT MULTI-THREADED SUM-PRODUCT NETWORK INFERENCE IN THE RECONFIGURABLE CLOUD

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *High-Throughput Multi-Threaded Sum-Product Network Inference in the Reconfigurable Cloud* by *Micha Ober, Jaco Hofmann, Lukas Sommer, Lukas Weber and Andreas Koch* in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. The contribution of the author of this thesis is summarized as follows.

» *The extension of the TaPaSCo framework to the Amazon AWS EC2 F1 FPGA platform was designed and implemented by Micha Ober, with the help of Jaco Hofmann. Lukas Sommer was in charge of extension of the software interface library for block-wise, multithreaded execution via OpenMP and for the conception and implementation of the source-to-source compiler used for fast inference on CPU and GPU. The required enhancements of the SPN hardware accelerator's memory access was joint work by Jaco Hofmann and Lukas Sommer. The experiments on the AWS EC2 F1 FPGA platform were conducted by Micha Ober, Lukas Sommer performed the experiments on CPU and GPU. The manuscript was joint work by all authors.* «

ABSTRACT

Large cloud providers have started to make powerful FPGAs available as part of their public cloud offers. One promising application area for this kind of instances is the acceleration of machine learning tasks.

This work presents an accelerator architecture that uses multiple accelerator cores for the inference in so-called Sum-Product Networks and complements it with a host software interface that overlaps data-transfer and actual computation.

The evaluation shows that, the proposed architecture deployed to Amazon AWS F1 instances is able to outperform a 12-core Xeon processor by a factor of up to 1.9x and a Nvidia Tesla V100 GPU by a factor of up to 6.6x.

16.1 INTRODUCTION

FPGAs have received increasing attention as a potential platform for the implementation of application-specific accelerators for datacenter-workload in recent years. As a consequence, large cloud providers have started to make FPGAs available in their public cloud offers, such as Amazon AWS with the F1-instances.

Next to genomics research, financial analysis and high-throughput image and video-processing, FPGAs in the cloud are also used to solve a wide range of machine learning problems. Starting with projects such as Microsoft's Brainwave [1, 4], FPGAs have established themselves as a platform for the acceleration of machine learning tasks, next to GPUs and custom ASICs such as Google's TPU [5].

While much of the existing work on the acceleration of ML tasks on FPGAs has been devoted to neural networks, e.g., for speech recognition or image classification using convolutional neural networks (CNN), a completely different problem is tackled in prior work such as [12, 13], namely the inference in so-called *Sum-Product Networks* (SPN).

Sum-Product Networks are one of the first models from the class of Probabilistic Graphical Models that can provide *tractable* inference, and, in contrast to neural networks, compute *exact* probability values. This difference also poses interesting challenges to the hardware implementation with regard to the arithmetic precision, and many optimization techniques often employed for the mapping of neural networks to FPGA accelerators, such as weight quantization, cannot be applied to Sum-Product Networks [13].

Yet, the evaluation in the prior work has shown some promising results, with a pipelined datapath architecture and memory interface outperforming CPUs and a Tensorflow-based GPU-implementation of SPN-inference [12].

This work aims at extending the existing framework to efficiently map and run the inference in Sum-Product Networks on the publicly available FPGA cloud-offer Amazon AWS F1.

We present the following contributions:

- An extension to the open-source SoC generation framework TaPaSCo [6] to support Amazon AWS F1. TaPaSCo is then used to automatically generate all infrastructure necessary to run the proposed accelerators on F1 and provides convenient software interfaces.

- The existing SPN accelerator generator is extended to allow for concurrent execution of *multiple* accelerators, promising better resource utilization through overlapping processing and memory transfers.

- The software infrastructure is improved to incorporate the parallel execution and preloading of data.

The paper is structured as follows: In Section 16.2, background on Sum-Product Networks is given. Section 16.3 shows the existing framework for the acceleration of SPN inference on FPGAs. Section 16.4 describes the accelerators, the implementation of support for Amazon AWS F1 instances in TaPaSCo, and the extensions to the existing framework and software interface. In Section 16.5, three different SoC-architectures are evaluated with regards to their FPGA implementation. Additionally, the FPGA's performance is compared with a CPU- and GPU-based implementation of SPN-inference. Finally, Section 16.6 concludes this paper and gives an outlook to future work.

16.2 sum-product networks

Sum-Product Networks (SPN) [9] are a type of models from the class of *Probabilistic Graphical Models* (PGM), which have received increasing attention in recent time. PGMs capture statistical information and relations over the variables in a dataset. By using probabilistic queries, PGMs can then be used to solve a wide range of machine learning problems, such as classification or regression. They can also be used to derive statistical properties, such as marginals or conditionals, for a given dataset and input values. For instance, on the NeurIPS corpus, they can be used to compute the probability of different words to occur with a certain frequency in a text.

In contrast to neural networks, which are the currently dominating models in the machine learning (ML) domain, PGMs are capable of computing *exact* probability values. However, earlier approaches to probabilistic graphical models, such as Bayesian Networks or Markov Random Fields, suffer from the fact that the inference in unrestricted PGMs is intractable in general.

But Sum-Product Networks overcome this limitation and combine the ability to compute exact probabilities with *tractable* inference in linear time with respect to the network size [8]. SPNs not only allow to efficiently solve classification or regression problems, but can also be used to calculate additional properties of the underlying probability distribution, such as entropy or mutual information. Examples for the use of SPNs in real-world applications include classification of the characters in a handwritten sequence [11], or path planning algorithms for mobile robots [10]. Because SPNs compute exact probabilities, they are also able to express uncertainty over their output in such applications (e.g. in sequence labeling), a capability not present on neural networks.

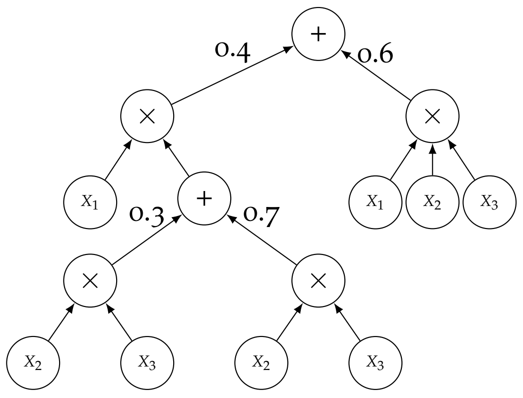


Figure 16.1: Example of a valid Sum-Product Network, capturing the joint probability distribution of the variables x_1, x_2 and x_3.

16.2.1 *Model Representation*

Sum-Product Networks capture the joint probability distribution over a set of random variables as a rooted, directed acyclic graph (DAG), with three different kinds of nodes:

- Leaf nodes represent univariate distributions. For an efficient mapping to the FPGA, these can be represented using *histograms*.
- Product nodes correspond to a factorization over independent distributions, and are therefore always defined over different scopes (i.e., random variables).
- Weighted sum nodes represent a mixture of multiple distributions over the same scope as a convex combination.

An example for a valid Sum-Product Network, which captures the joint probability $\mathcal{P}(x_1, x_2, x_3)$ for three random variables, is given in Fig. 16.1.

16.2.2 *Learning*

Similar to many other machine learning models, the structure and parameters of a Sum-Product Network can be learned from training data. As a brief introduction, a short description of the top-down learning algorithm proposed by Molina, Vergari, Di Mauro, Natarajan, Esposito, and Kersting [7] follows. Next to this algorithm, a number of other approaches to learn the structure and/or parameters from data exists.

The algorithm's recursive base case is reached, if only a *single* variable is left. In this case, the algorithm learns a histogram representing the univariate distribution of this variable.

If more than one variable is still left to process, the algorithm tries to find independent sets of variables using a pair-wise, parametric independency test. If the test succeeds, a product node is created and the algorithm recurses on the independent sets. If the independence test fails, the training data is partitioned using clustering. This induces a weighted sum node, where the weights correspond to the normalized, proportional size of the associated clusters.

This work focuses on the *inference* process in a given, previously trained and optimized SPN. Therefore, learning of the SPN structure can happen offline on a traditional CPU.

16.2.3 *Inference*

As stated earlier, SPNs can be used to compute a range of different probabilistic queries to solve different ML problems, such as multi-class classification. However, independent of the actual probabilistic query, the inference process boils down to a bottom-up evaluation of the SPN graph with given (partial) evidence. By indexing the histogram, the probability value associated with the given evidence for the input variable can be determined. These probability values are then propagated upwards through the tree. At product nodes, the probabilities for the different child nodes are multiplied. In case of a sum-node, the probabilities are first multiplied with the associated weight and then summed up. Eventually, at the root node of the SPN, the inference process will yield a single probability value as the answer to our probabilistic query.

This work focuses on the computation of joint probabilities, which corresponds to a single evaluation of the SPN with full evidence per input sample. However, the datapath architecture could easily be extended to support other kinds of probabilistic queries on SPNs, for example to compute marginals, where the histograms for the marginalized variables are replaced by the value 1. With the value for joint probability and the result from a marginalized evaluation, conditional probabilities can easily be computed as $\mathcal{P}(\mathcal{Y}|\mathcal{X}) = \frac{\mathcal{P}(\mathcal{Y},\mathcal{X})}{\mathcal{P}(\mathcal{X})}$.

16.3 PRIOR WORK

To the best of our knowledge, the work we presented in [12, 13] is the only work to date on the acceleration of SPN inference on FPGAs. The goal of the prior work was the acceleration of the computation of the joint probability over the variables represented by an SPN by its evaluation, and was done by processing small batches of input samples with high throughput.

To this end, an automatic flow that maps textual representations of SPNs to fully custom datapaths is already available as a result of the prior work. The generated datapath for an SPN captures the computation tree represented by the SPN and is completely pipelined by using a fully spatial implementation, where each operation in the

SPN tree directly corresponds to a hardware operator in the datapath. For the hardware operator implementation the FloPoCo framework [2] is used, with a format similar to IEEE-754 double precision (apart from subnormals).

The automatically generated datapath is supplied with data through a pipelined memory infrastructure. The memory infrastructure uses AXI4 burst requests to supply the datapath with a continuous stream of input data from the DDR3-memory attached to the FPGA and writes back results to memory. In both the load- and store-unit, a MIMO (Multi-in-Multi-out) unit is used to translate between the external AXI4 datawidth, and the internal input- and result bitwidth of the datapath, respectively.

The open-source TaPaSCo-framework [6] is used to integrate the accelerator core into a heterogeneous system. The automatic SoC composition capability of TaPaSCo can be used to automatically connect the accelerator(s) to the FPGA's external memory and the PCIe-based host interface. Using the TaPaSCo software API, execution of the accelerator(s) can be controlled and data, such as, input & results, can be transferred between host memory and FPGA external memory.

16.4 EXTENSION FOR HIGH-THROUGHPUT INFERENCE

This work builds on the automatic mapping flow from [12] to generate pipelined datapaths for SPNs. In the current work, generated datapaths are coupled with a completely new memory infrastructure, designed from the ground up to support the simultaneous execution on *multiple* accelerator cores, and concurrent data-transfers from/to the host. Beyond that, to incorporate the necessary infrastructure, the open-source TaPaSCo framework has been extended to support Amazon AWS F1 instances, and provided with a new multi-threaded SW-interface for simultaneous SPN inference.

Accordingly, the following sections describe the two necessary steps: (1) Extend TaPaSCo with AWS capabilities. (2) Ensure multi-threading compatibility by implementing a new, high-performance memory infrastructure to feed the datapath.

16.4.1 *Extending the TaPaSCo Framework for the Reconfigurable Cloud*

Amazon's first generation FPGA instances actually run in a virtualized environment, with only limited access to the hardware. Amazon uses the partial reconfiguration feature to split the area of the Virtex UltraScale+ FPGA into an user accessible area (called custom logic) and the so-called *Shell*. The Shell is provided by Amazon as an encrypted design checkpoint and acts as an interface between the custom logic and the external peripherals such as the PCI Express interface or memory. A DMA engine for data transfer between host and FPGA is provided by the Shell in the form of the Xilinx XDMA IP core. However, TaPaSCo's own DMA engine is used here, as it achieves better throughput and requires fewer driver changes to be used.

The Shell provides up to 16 MSI based interrupts that can be used by the developer. TaPaSCo, however, requires more than 16 interrupts for optimal performance. A custom interrupt controller is thus used to translate between the needs of TaPaSCo and the interfaces of the AWS Shell. The interrupt controller has a FIFO buffer on each interrupt input and an additional input for ACK signals coming from the Shell. Furthermore, interrupts are ACKed from the host to avoid situations where the interrupt service routine is already active, which would result in dropped interrupts.

Each FPGA on an F1 instance has access to up to four channels of ECC DDR4 memory, each with a size of 16 GiB. One channel is provided by a MIG memory controller inside the Shell, while the MIGs for the other three channels are placed inside the custom logic. The developer can decide how many memory channels are enabled, offering the possibility to trade-off between a larger number of memory channels, and the available area on the device.

Amazon does provide an Amazon Machine Image (AMI) for FPGA development, which includes all required Xilinx tools and licenses. For users, owning the required licenses, on-premise development is supported as well. In both cases, the design flow ends with a design checkpoint containing both the custom logic and the Shell. This design flow differs in many ways from any existing flow in TaPaSCo, as it includes writing out checkpoints for the custom logic, adding the checkpoint of the Shell to the project, linking the custom logic and Shell together, or adding a top-level wrapper provided by Amazon. The Amazon design flow has now been integrated into TaPaSCo, so a final checkpoint is created together with a mandatory manifest file, which are then both sent to Amazon servers. There, in an automated process, the submitted checkpoint is verified to not contain any combinatorial loops or unrouted nets. On success, the bitstream can be referenced by a globally unique identifier, which can in turn be used to load the bitstream into a device.

As Amazon limits the power draw of the FPGA to 85 watts by gating the clock to the custom logic should this limit be exceeded, clocking was another important topic when integrating the F1 platform into TaPaSCo. The Shell provides up to three different clock groups with different, but not arbitrary frequencies. As the design space exploration of TaPaSCo requires the design frequency to be varied in 5 MHz steps, an additional clock generator (MMCM) is used. The placement of that MMCM is tricky, however, as it cannot be placed near the Shell's MMCM, as this area is not accessible to user logic. This suboptimal placement of the clocking resources put some additional constraints on the design to optimize timing. For example the internal logic of all AXI Interconnects is always clocked by the *Shell's* main clock, even if the *design clock* is faster.

16.4.2 *Improvements of the Accelerator Architecture*

The existing memory infrastructure in the previous work was clearly designed to provide the maximum memory throughput in a scenario with only a single accelerator and where the execution of a job would *not* start before the previous job had completed. As a result, the prior load infrastructure would completely occupy the AXI4-bus by having a high number of transfers with maximum burst length in flight, ensuring that a continuous stream of input data gets supplied to the datapath. On the other side of the accelerator, the data store unit did not buffer many results, but would instead use long-lasting burst requests, occupying the AXI4 write-channel for a long time.

This behavior is problematic, as the accelerator presented in this work executes inference on *multiple* SPN instances *simultaneously*, which requires *overlapping* of data transfer with accelerator execution.

First, the large number of incomplete memory transactions can lead to deadlocks with the simultaneous execution of multiple accelerators: If one core is blocked from writing because another core is occupying the *write* channel, it will not be able to complete potentially incomplete *read* transactions due to back-pressure in the datapath. If the core occupying the write channel is blocked from reading due to the first core's outstanding read request, it won't be able to compute the results necessary to complete the write request, effectively resulting in a deadlock.

The occupation of the write channel is problematic for another reason: TaPaSCo uses a DMA engine to transfer data from the host memory to the FPGA's external memory. However, if the memory interface's write channel is occupied by the accelerator for a long time span, the DMA engine cannot execute the transfer, forcing computation and data transfer to effectively execute sequentially.

To overcome this problem, a completely new memory infrastructure for the accelerator is implemented. In the course of the re-design, the input MIMO unit was moved to the datapath, in order to unify the interface of the accelerator core across different Sum-Product Networks.

The new load interface has fewer incomplete transfers in flight, and now uses an internal buffer that allows to store a substantial amount of input data. A new request is only issued if all data that will be loaded in the course of this request can be buffered internally. In this manner, it can be ensured that even if in the presence of back-pressure from the store interface, all incomplete load requests can be completed to avoid deadlocks. Additionally, this buffer allows the accelerator

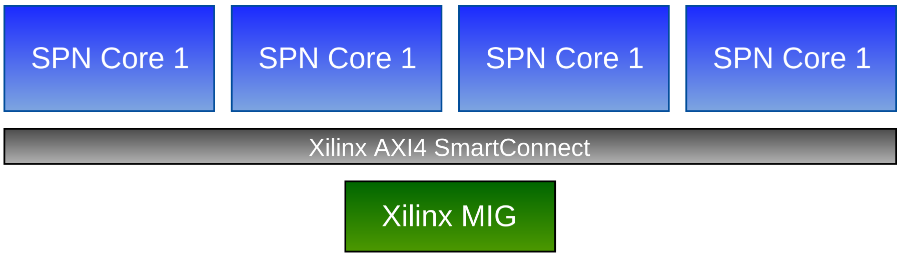


Figure 16.2: Multi-core architecture with single memory interface.

to continue its computation for some time, even if the current load request incurs some delay.

The store interface is also redesigned to be able to buffer results for a complete burst request internally. Only after enough results for a complete burst request have been calculated, a new write request is issued. Because all write data is available, the write request can be completed in the shortest time possible, not only avoiding deadlock, but also allowing the TaPaSCo DMA engine to transfer data in parallel to the computation in the accelerator cores.

16.4.3 *SoC Architecture*

The TaPaSCo-framework, with the extensions described in Section 16.4.1, is able to automatically construct and generate SoC-designs for the F1 FPGA instances provided by Amazon AWS. The AXI4 slave interface of the accelerator core, which is used for control signalling (e.g., launch execution) is attached to the PCIe-based host interface. The AXI4 master interfaces of each core are automatically connected to the memory interface of the FPGA's external DDR-memory via a shared bus. As described in the previous section, the SPN accelerator core uses a single AXI4 master interface to read and write data and results from/to external memory.

The obligatory Shell environment provided by Amazon as part of the AWS F1 HDK by default contains a single interface to FPGA external memory. For the baseline architecture a single SPN accelerator core is connected to this interface via AXI4, allowing the accelerator core to read and write data from/to the external memory. As described in the previous section, the load/store interface of the accelerator is re-designed in a way that allows to operate the DMA-unit used by TaPaSCo to transfer data from host memory to FPGA memory concurrently to the accelerator.

Although this baseline architecture is already a fully functional accelerator for SPN inference, further improvements to the throughput can be made: The vast amount of hardware resources available on the Xilinx Virtex UltraScale+ devices, with which the AWS F1 instances are equipped, allows multiple instances of the SPN accelerator cores, which can compute multiple inferences simultaneously. In the multi-

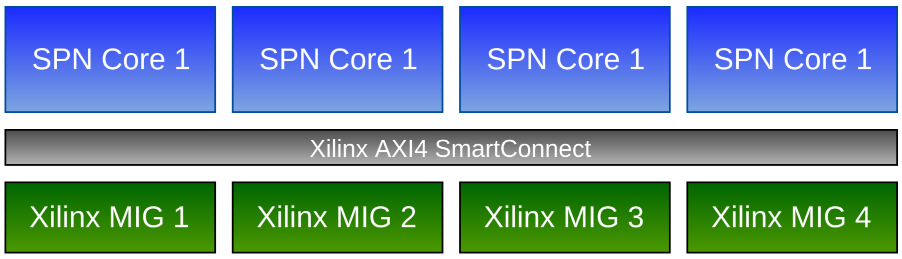


Figure 16.3: Multi-core architecture with four independent memory interfaces.

core architecture, depicted in Fig. 16.2, up to four SPN accelerator are connected to the memory interface via a shared bus. Due to the changes made to the memory infrastructure (see Section 16.4.2), the different cores can now operate concurrently on the same bus.

However, depending on the number of input values of the SPN and the memory bandwidth requirement that goes along with that (remember that the datapath architecture is fully pipelined and can process a new sample in every clock cycle), the single interface to memory can become a bottleneck. To overcome this limitation, and fully exploit the memory bandwidth provided by the four independent memory banks on the Virtex UltraScale+ device, and additional three *more* memory interfaces can be used. Just as in the multi-core architecture, up to four SPN accelerator cores are connected to the four memory interfaces via an AXI4 SmartConnect, depicted in Fig. 16.3. Note that as long as the four accelerator cores access distinct memory regions located on the different memory banks, they can be served *simultaneously* through the SmartConnect, so this bus infrastructure will not become a bottleneck.

16.4.4 *Multi-threaded Software-Interface*

The existing host software uses the TaPaSCo software API to launch the entire computation in a single, large job. However, to fully exploit the available throughput, the software interface needs to reflect the multi-core architecture in the FPGA-hardware and launch multiple, independent jobs that execute concurrently on the different SPN accelerator cores.

Although it would be sufficient to launch four independent jobs in different threads, there are more improvements to be had: In the prior work, the overhead for transferring data between host- and FPGA-memory turned out to be relatively large and contributed significantly to the overall execution time.

Therefore, a goal is to extend the software interface to allow overlapping of computation on the SPN accelerator cores and data transfer to reduce the overall execution time of SPN inference. The computation is split into independent blocks (chunks), which can be independently

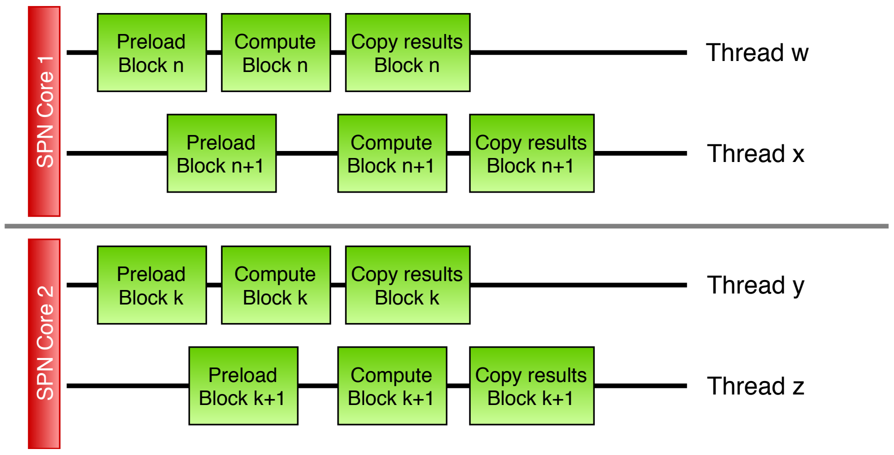


Figure 16.4: Block-wise execution overlapping actual computation and data-transfer between host- and FPGA-memory.

preloaded. Fig. 16.4 shows how computation and data transfer are overlapped with this change: While block n is computed on the FPGA SPN accelerator core 1, the input data for block $n+1$ is concurrently transferred to the FPGA memory by another thread. As soon as the computation of block n is completed, the computation for block $n+1$ can start, while the results for block n are copied back to host memory. At the same time, the same execution scheme can be used for another block of samples on another SPN accelerator core.

This execution scheme is implemented based on the TaPaSCo software API using OpenMP on the host, the block size and number of threads are configurable parameters for the host software. As described in the previous section, it is important that the different SPN accelerator cores operate on *distinct* memory regions located on different memory banks. To ensure this behaviour, each host thread has fixed device memory addresses to evenly distribute the workload across the different SPN accelerator cores in the multi-core architectures (cf. Fig. 16.2 & Fig. 16.3).

16.5 EVALUATION

16.5.1 *Benchmarks*

The approach is evaluated through a set of eight different benchmark SPN models derived from the NeurIPS corpus [3], which has also been used before for evaluation purposes in [12, 13].

The SPN networks derived from the NeurIPS corpus capture statistical properties about the number of occurrences (frequency) of different words in the texts contained in the corpus. Using inference in this networks, it is for example possible to compute the probability that certain words each occur with a specific frequency in a text.

The increasing number of inputs of the networks in the benchmark make them particularly interesting for the evaluation: Not only does the required memory bandwidth increase with the number of inputs, but also the size of the SPN networks themselves increases, leading to more computational demand and, with the fully spatial implementation of the datapaths, also more FPGA resource consumption.

All the following performance evaluations have been performed using a dataset containing ten million samples per benchmark.

16.5.2 *FPGA Implementation*

The FPGA resource usage and implementation results are evaluated by implementing each of the three different SoC-architectures described in Section 16.4.3 for each of the benchmarks on the xcvu9pflgb2104-2 FPGA, with which the Amazon AWS F1 instances are equipped. Xilinx Vivado version 2018.3 and TaPaSCo version 2019.10 (pre-release), extended as described in Section 16.4.1, are used. All reported numbers are taken from the post-place&route reports generated by Vivado. The resource usages for each SoC-architecture and each benchmark are given in Fig. 16.5. For brevity, the numbers are given relative to the number of available resources. For reference, the VU9P FPGA has in total 148×10^3 CLB, 2×10^3 BRAM and 7×10^3 DSP.

Even with a single memory interface and only one SPN accelerator core, the design already requires a significant amount of logic resources (CLB), partially also due to the obligatory AWS Shell and basic platform components (e.g., PCIe interface). However, there is still sufficient headroom left to implement multiple SPN accelerator cores. The operating frequencies for this configuration reach up to 410 MHz.

The resource usage increases noticeably for the first multi-core architecture, which uses up to four SPN accelerator cores coupled with a single memory interface. For the two largest networks, NIPS70 and NIPS80, only three accelerator cores fit onto the FPGA device. The DSP usage correlates linearly to the number of SPN accelerator cores, but for the other two kinds of resources, the relative increase is much smaller: Logic resources (CLB) required increase just by a factor 1.5x to 2.13x, and only between 1.2x and 1.4x more BRAMs are used. The higher resource usage also results in a noticeable degradation of the operating frequency, with this configuration the highest achievable frequency is 370 MHz.

When using four memory interfaces in the multi-core architecture, the available resources limit the number of SPN accelerators that can be implemented for some examples: For NIPS50 and NIPS60 only three cores fit the device and for NIPS70 and NIPS80 only two cores can be placed. Regarding the resource usage, the increase compared to the baseline architecture with one core and one memory interface

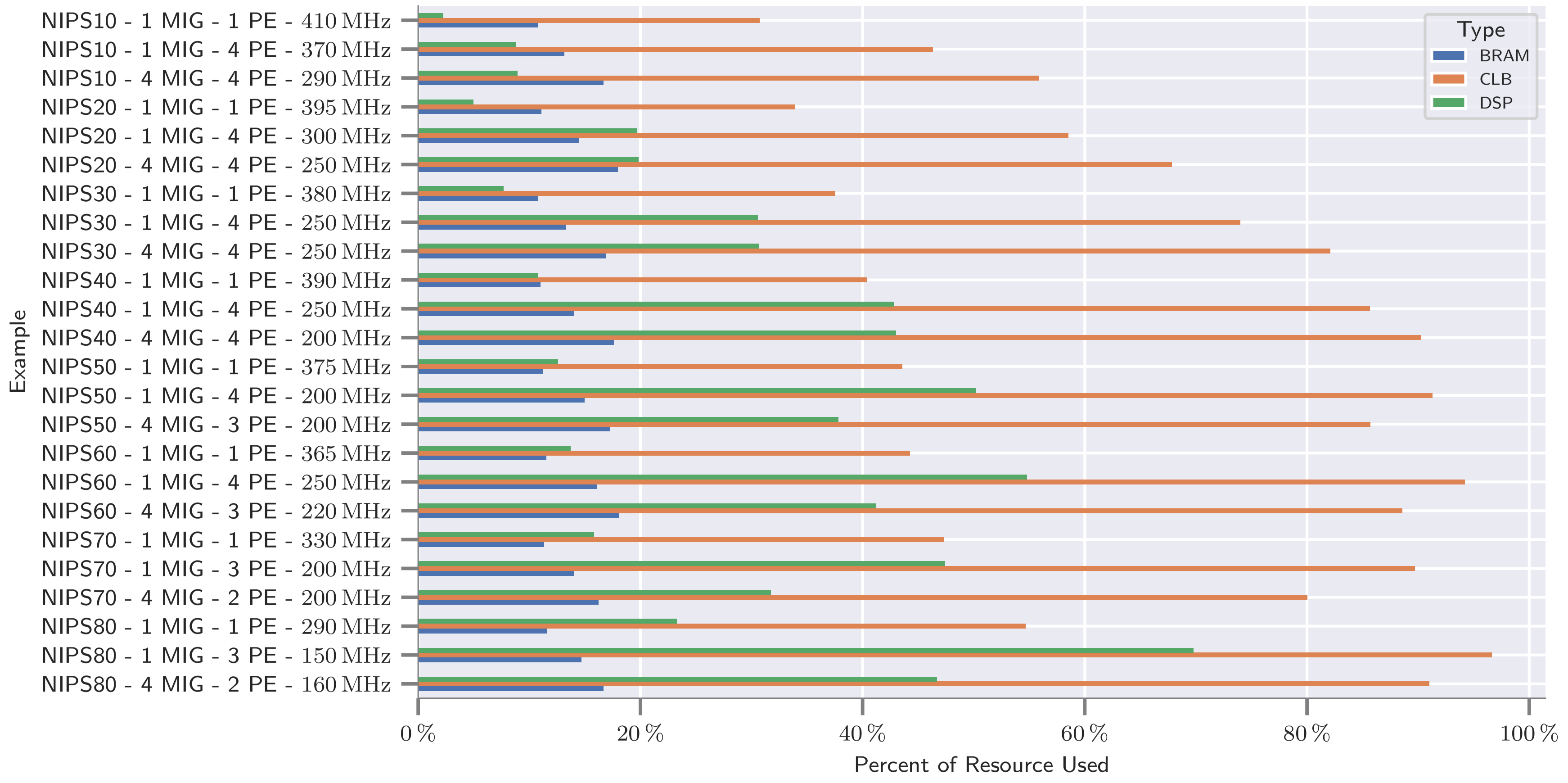


Figure 16.5: Utilization of the main FPGA resources on the AWS F1 instances. The designs are mainly limited by available CLBs. DSPs and BRAM play a minor role in the overall resource requirements. A high CLB utilization results in difficult routing and a tendency for lower clock frequencies.

is comparable to the first multi-core architecture. The number of DSPs used scales linearly with the number of SPN accelerator cores, CLBs increase by factor 1.6x to 2.2x and BRAM usage increases by 1.43x to 1.6x. The direct comparison of the two multi-core architectures shows that the additional memory interfaces only consume a relatively small number of resources.

However, with four memory interfaces, three of them are implemented in the custom logic. Here, timing closure can become problematic. The achievable frequency drops by 30%-50% across all benchmarks, partly because of the high resource usage (more than 80% of logic resources for most examples) and the IO-restricted locations for the memory interfaces.

16.5.3 *Performance Evaluation*

In this section, the performance of the three different architectures is compared to CPU- and GPU-based implementations. Furthermore, the FPGA-implementations have different choices for block-size and number of threads to determine the optimal configuration (cf. Section 16.4.4). For brevity detailed charts have been omitted. For most benchmarks, a block-size of 400,000 samples per block, and three to four host software threads per SPN accelerator result in the best performance.

16.5.3.1 *CPU Baseline*

For the CPU baseline, a custom C++ compilation flow is used, sharing some of the infrastructure (intermediate representation, parser) of the datapath generator. The flow automatically generates optimized C++ code for each example and invokes the compiler (gcc) with flags `-O3` and `-ffast-math`. To efficiently parallelize the workloads, the dataset is split into blocks, which can be processed concurrently using OpenMP with twelve threads on a 12-core Xeon E5-2680 v3 CPU of the Lichtenberg high-performance computing cluster.

16.5.3.2 *GPU Baseline*

The evaluation in prior work [12] showed that Tensorflow, tailored towards standard neural networks, is not able to efficiently map the inference in SPNs to GPUs. Therefore, for a fair comparison, a custom, optimized CUDA compilation flow, again based on the common infrastructure we developed, is used. Within each block, the processing of samples is parallelized across the available GPU processing units. Eight threads on the host CPU are utilized to parallelize processing of blocks. The evaluation is done on the top-end Nvidia Tesla V100 GPUs available in the Amazon AWS cloud with CUDA version 9. The evaluation shows that our custom SPN-to-CUDA compilation flow

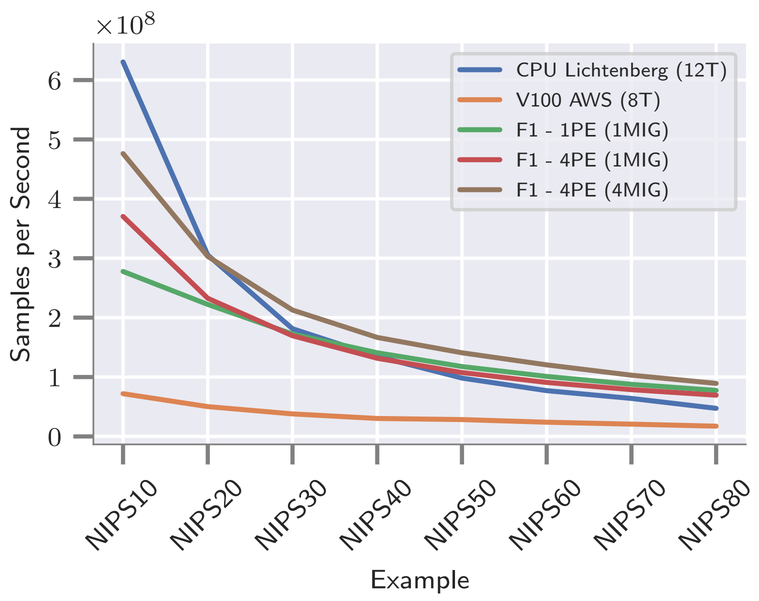


Figure 16.6: Samples processed per second by the CPU, GPU and the FPGA architecture for increasing problem sizes. For very small problem sizes, the CPU is faster as the transfer overhead to PCIE-based accelerators is dominant. However, for larger problem sizes, the FPGA pulls ahead. The GPU cannot keep up with either of the other two competing approaches.

improves the throughput by a factor of up to 109x (geo.-mean 96x) over the original Tensorflow-based mapping.

16.5.3.3 *Performance Comparison*

Fig. 16.6 shows the throughput of the CPU- and GPU-baseline and the three different SoC-architectures.

Even though the performance of the GPU-baseline improves up to factor of 109x over the old Tensorflow-based baseline, the GPU is still left behind and provides the least throughput. The computational density of the SPN inference in the examples is not sufficient to compensate for the data-transfer overheads and fully exploit the GPU's computational power. This can also be seen in our observation that the GPU performance is almost independent of the SPN network size.

In contrast, the CPU performance *is* highly dependent on the network size, and decreases as the number of operations in the SPN's tree increases. Still, for small examples, the CPU provides the best throughput of all architectures, since it does not incur data transfer overheads.

The baseline FPGA-architecture with a single SPN accelerator core and memory interface is outperformed by the CPU for the examples up to NIPS30, but for *larger* networks, the pipelined processing yields significant speedups of up to factor 1.6x (geo.-mean 1.01x) over the CPU, and 4.6x over the GPU (geo.-mean 4.3x).

Despite using up to four cores with only a single memory interface, the first multi-core architecture is also able to outperform the CPU by a factor of up to 1.47x (geo.-mean 0.99x) for larger examples and the GPU by a factor of up to 5.15x (geo.-mean 4.23x) on all examples. The overall performance is slightly lower compared to the single-core architecture due to the operating frequency drop (cf. Section 16.5.2) and the saturation of the memory interface. Therefore, the DMA transfers of data from/to host and computation effectively happen sequentially, leading to a degradation in performance, in particular for the larger examples.

With up to four SPN cores and four distinct memory interfaces in the second multi-core architecture, the FPGA is already on par with the CPU for NIPS20 and even more significant speedups can be observed for all bigger networks. The speedup reaches as high as 1.9x over CPU (geo.-mean 1.28x) and 6.6x over GPU (geo.-mean 5.47x).

When directly comparing the baseline FPGA-architecture with the second multi-core architecture with up to four SPN cores and four distinct memory interfaces, it can be seen that the performance advantage *decreases* for the larger networks. Whereas the relative speedup of the multi-core architecture is 1.7x for NIPS10, it decreases to only 1.15x for NIPS80. This is due to the fact that four SPN cores do not fit on the device for the larger examples (cf. Section 16.5.2), and the high penalty on the lowered operating frequency incurred when using four distinct memory interfaces. Note that this is most likely an artifact specific to the AWS F1 system architecture. Still, this multi-core architecture is the best performing FPGA accelerator architecture.

16.6 CONCLUSION & OUTLOOK

This work presents an accelerator architecture, based on a framework developed in prior work, for the efficient inference of so-called Sum-Product Networks on FPGA instances in the cloud based Amazon AWS F1 instances.

The open-source TaPaSCo-framework has been extended to support Amazon AWS F1 instances, enabling automatic SoC generation based on the proposed accelerator including integration into a heterogeneous system.

To make use of the resources available on the FPGA-instances, three different architectures have been investigated that allow concurrent processing on multiple accelerator cores. This required a re-design of the accelerator memory interface compared to prior work. Lastly, the accelerator architecture is complemented with a multi-threaded host software interface that is able to efficiently overlap data-transfer and actual computation in order to improve end-to-end execution times.

The evaluation shows that the developed architectures are able to outperform a 12-core Xeon CPU and a Tesla V100 GPU by up to a factor of 1.9x and 6.6x, respectively.

In the future, we plan to extend the mapping toolflow with the ability to share operators between multiple operations in the SPN graph, allowing us to map even bigger networks to FPGAs.

The extension to the TaPaSCo-framework developed in the course of this work will also become an official part of the TaPaSCo open-source release on Github [14]. Just as any other platform in the TaPaSCo-framework, the AWS support will allow to automatically compose a SoC-design around any accelerator core with a suitable AXI4-interface and connect to it from the host CPU via the TaPaSCo software API.

ACKNOWLEDGEMENTS.

The authors would like to thank Xilinx Inc. for supporting their work by donations of hard- and software. Calculations for this research were conducted on the Lichtenberg high performance computer of TU Darmstadt.

Finally, the authors would like to thank Kristian Kersting and Alejandro Molina, for much appreciated discussions of the subject, as well as extensive insight into the matter of Sum-Product Networks.

REFERENCES

[1] Eric Chung, Jeremy Fowers, et al. "Accelerating Persistent Neural Networks at Datacenter Scale." In: *Hot Chips 29: A Symposium on High-Performance Chips*. 2017.

[2] Florent de Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo." In: *IEEE Design & Test of Computers* (July 2011).

[3] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[4] Karl Freund. *Microsoft: FPGA Wins Versus Google TPUs For AI*. https://www.forbes.com/sites/moorinsights/2017/08/28/microsoft-fpga-wins-versus-google-tpus-for-ai. Accessed April 5, 2018. 2017.

[5] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In: *44th Annual International Symposium on Computer Architecture, ISCA 2017*. ACM, 2017.

[6] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems." In: *Applied Reconfigurable Computing*. 2019.

[7] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Sriraam Natarajan, Floriana Esposito, and Kristian Kersting. "Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains." In: (2018).

[8] Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro Domingos. "On Theoretical Properties of Sum-Product Networks." In: *Proc. of AISTATS*. 2015.

[9] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: a New Deep Architecture." In: *Proc. of UAI* (2011).

[10] Andrzej Pronobis, Francesco Riccio, and Rajesh PN Rao. "Deep spatial affordance hierarchy: Spatial knowledge representation for planning in large-scale environments." In: *ICAPS 2017 Workshop on Planning and Robotics*. 2017.

[11] Martin Ratajczak, Sebastian Tschiatschek, and Franz Pernkopf. "Sum-Product Networks for Sequence Labeling." In: *CoRR* (2018). arXiv: 1807.02324.

[12] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-Based Accelerators." In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. Oct. 2018, pp. 350–357. DOI: 10.1109/ICCD.2018.00060.

[13] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. "Automatic Synthesis of FPGA-based Accelerators for the Sum-Product Network Inference Problem." In: *ICML 2018 Workshop on Tractable Probabilistic Models (TPM)*. 2018.

[14] *TaPaSCo framework on Github*. https://github.com/esa-tu-darmstadt/tapasco. 2019.

17 COMPARISON OF ARITHMETIC NUMBER FORMATS FOR INFERENCE IN SUM-PRODUCT NETWORKS ON FPGAS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs* by *Lukas Sommer, Lukas Weber, Martin Kumm and Andreas Koch* in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. The contribution of the author of this thesis is summarized as follows.

» *As the corresponding and co-leading author, Lukas Sommer was responsible for the design and implementation of the hardware accelerator's load-/store infrastructure and controller and the initial conception of the automatic toolflow. He was also in charge of the source-to-source compiler, which was used for the fast software-based design-space exploration of the arithmetic formats and for fast inference on CPUs and GPUs. The logarithmic number system arithmetic hardware operators were designed and implemented by Lukas Weber, who was also responsible for the integration of the Posit hardware operators into the framework. The custom floating point hardware operators were contributed by Martin Kumm. The FPGA performance experiments and design-space explorations were performed by Lukas Weber. The CPU and GPU experiments, as well as the FPGA power measurements, were conducted by Lukas Sommer. The manuscript was joint work by all authors.* «

ABSTRACT

Probabilistic Graphical Models (PGM) have recently received increasing attention for various machine learning tasks and approaches for their acceleration on FPGAs have been presented.

In this work, we investigate three different arithmetic formats, namely customized floating-point, Posit and logarithmic number systems with regard to their suitability for the inference in PGMs, specifically so-called Sum-Product Networks (SPN). Based on results from an automatic design-space exploration developed in this work, we implement hardware arithmetic operators for each format, optimized for SPN inference.

Our evaluation shows that the choice of the most area-efficient solution depends on the relation between the numbers of adders to multipliers in the network. Up to 57% and 68% of Slice and DSP

reductions, respectively, could be obtained compared to previous work. With regard to performance, all formats achieve similar results and outperform CPU and GPU-based implementations of SPN inference by factors up to 12x and 4.6x, respectively.

17.1 INTRODUCTION

Next to GPUs and custom ASICs, such as Google's TPU, FPGAs have established themselves as a succesful implementation platform for the acceleration of machine learning (ML) tasks, in particular for inference. Besides numerous works on the acceleration of the inference in neural networks, for example convolutional neural networks (CNN) for computer vision applications, new approaches to accelerate inference in *probabilistic models* on FPGAs have recently been presented.

One such approach for the inference in so-called *Sum-Product Networks* (SPN) was developed in [26, 27, 29]. Compared to neural networks, Sum-Product Networks, which belong to the class of *tractable* Probabilistic Graphical Models (PGM), can better deal with missing input features and, as SPNs compute *exact* probability values, are also able to express uncertainty over their outputs.

However, this ability also poses new challenges to the implementation of such networks on FPGAs. In [26, 27], the authors used a double-precision floating-point format to preserve accuracy. Such an arithmetic format is expensive to implement on FPGAs. Therefore, in this work, we seek to optimize the hardware arithmetic operators to *reduce* resource usage, while *preserving* sufficient accuracy. To this end, we will investigate three different arithmetic formats, namely "traditional" but customized floating point, logarithmic number system (LNS) and Posit, with regard to their suitability for FPGA-based accelerators for SPN inference.

We exploit an automatic and efficient design-space exploration (DSE) flow, based on software-only emulation of the arithmetic formats for SPN inference, to determine the minimal bit-widths required to preserve accuracy with each of the formats prior to hardware generation.

Based on the findings from our DSE, we then implement hardware arithmetic operators for each of the three investigated arithmetic formats, optimized for the inference in Sum-Product Networks on FPGAs. The optimized arithmetic operators are used to generate fully pipelined datapaths, which are integrated into a SoC-design providing the host-CPU software interface. In our extensive evaluation, we investigate which arithmetic format is most suited for SPN inference on FPGAs and compare the performance of the generated datapaths with CPU and GPU-based implementations of SPN inference.

17.2 SPN BACKGROUND

Sum-Product Networks [22] belong to the class of *probabilistic models*, which can be used for a range of different machine learning tasks. As they are also able to take the statistical nature of the data into account, and deal well with uncertainty and missing features, this class of models has received increasing attention recently.

After a probabilistic model has been trained from data, different machine learning problems, such as classification and regression, can be solved by using probabilistic queries on the trained model. An example for such a query would be to determine which news-article a user is most likely interested in, based on information on whether or not he or she has looked at other articles before.

In comparison with other probabilistic models and other ML-techniques, such as deep neural networks, SPNs exhibit a number of interesting characteristics, that makes them attractive for use in a range of different applications. For example, SPNs have already been used succesfully for sequence labeling [24], i.e., classifying the characters in a handwritten sequence, or in path planning algorithms for mobile robots [23].

One very important property of SPNs for their practical usage is the *efficiency* of the inference: While in general, inference for unrestricted PGMs is *intractable*, the inference in SPNs is guaranteed to be *linear* w.r.t. the number of nodes [3, 22]. This *tractable* inference is key to efficiently answering probabilistic queries in practical applications.

Another interesting property of SPNs is their expressiveness: From mixture models, which can easily be represented by a shallow Sum-Product Network with a single sum-node, SPNs inherit the *universal approximation property* [20]. This means that Sum-Product Networks can represent *any prediction* function, similar to deep neural networks.

One of the most interesting properties about Sum-Product Networks, that also makes SPNs stand out from other ML-techniques such as deep neural networks, is the *precision* of the inference process. Whereas neural networks generally compute *approximate* values, Sum-Product Networks are instances of Arithmetic Circuits [30] and therefore facilitate the computation of *exact* probability values. Beyond more precise answers to queries, this also offers the advantage that the inference process can be combined with *anomaly detection* by comparing the respective probabilities from different SPNs, and also better account for the statistical nature of the data.

In this work, we focus on the *inference* process in a pre-trained SPN. In this case, the learning has taken place offline on a traditional CPU-based machine.

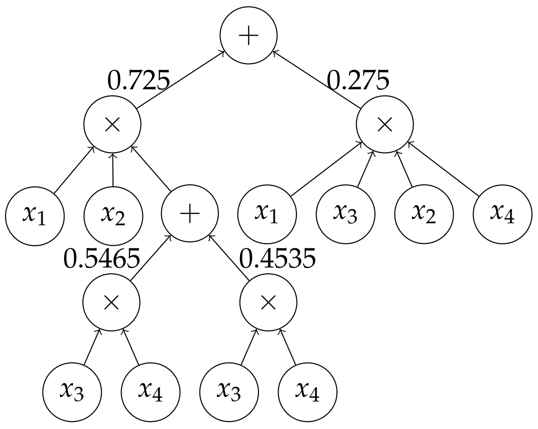


Figure 17.1: Example of a valid SPN representing the joint probability $\mathcal{P}(x_1, x_2, x_3, x_4)$.

17.2.1 *Model Representation*

A Sum-Product Network captures the joint probability $\mathcal{P}(X, Y, Z)$ over a set of variables $\{X, Y, Z\}$ in the form of a rooted, directed acyclic graph (DAG). An example for a valid SPN over the variables $\{x_1, x_2, x_3, x_4\}$ can be found in Fig. 17.1. The graph representation of SPNs is composed from three different kinds of nodes, with some additional restrictions to guarantee the validity of the SPN:

- Leaf nodes represent univariate distributions over a single variable. In this work, based on the approach proposed by Molina et al. [19], we represent these univariate distributions by histograms for an efficient mapping to the FPGA.

- Factorizations over independent distributions are represented by product nodes in the graph. The child nodes of a product node are defined over different scopes, i.e., each sub-tree uses a distinct set of variables.

- Mixtures over distributions defined over the same set of variables are represented by sum-nodes, where each child node is additionally associated with a weight. The child nodes of a sum node are defined over the same scope, i.e., the same set of variables appears in each subtree.

17.2.2 *Inference*

The inference process depends on the kind of probabilistic query that should be answered. Common to all kinds of inference is the bottom-up evaluation of the SPN graph, eventually yielding a probability value at the root of the graph.

The most basic kind of inference in an SPN is the joint computation, yielding the joint probability for given input values, i.e., full evidence.

In the first step, the leaf nodes are queried with the value of the associated input variable, yielding a probability value. In this work, the univariate distributions at leaf nodes associated with an input variable are modeled using histograms, which are simply indexed with the input value. The resulting probability values are then propagated upwards through the tree. At product nodes, the child node values are multiplied with each other. When a sum node is reached, the child node values are first multiplied with the corresponding weight and then summed up.

Marginalization [20] of variables is another possible kind of query that can be answered by inference. To this end, the leaf nodes associated with the marginalized input variables are replaced by the probability 1. The remaining leaf nodes are just queried with the associated input values from the partial evidence. The rest of the inference process is identical to the joint computation. Through the combination of joint computation and marginalization, it is also possible to compute conditional probabilities using the following equation, where the numerator of the fraction corresponds to the joint computation and the denominator can be computed by marginalization of Y: $\mathcal{P}(Y|X) = \frac{\mathcal{P}(Y,X)}{\mathcal{P}(X)}$.

In this work, we focus on joint computation, but the datapath architecture can easily be extended to support other kinds of inference, such as marginalization.

In prior work, accelerators for the inference in other Probabilistic Graphical Models such as Bayesian Networks (BN) [1] or Markov Random Fields (MRF) [5] were developed. However, as discussed in the previous section, the inference in these kinds of PGMs differs significantly from Sum-Product Networks and the techniques used in these works cannot be applied to SPNs without further ado.

To the best of our knowledge, the only approach to accelerate SPN inference on FPGAs was presented in [26, 27]. In this work, we seek to extend the automatic toolflow from this work with three different arithmetic formats.

17.3 ARITHMETIC NUMBER FORMATS

17.3.1 *Fixed Point*

Fixed-point arithmetic can be implemented very efficiently in FPGAs. Yet, we do not consider fixed-point further in this work, because with SPNs, very small numbers can still represent significant results. In [19], the authors reported on relevant *log-likelihoods* as small as -144 and a first analysis of the dynamic range of the results of our benchmark networks showed that the smallest numbers are as small as $1.85 \cdot 10^{-88}$.

As each number can also be as large as one, at least 292 bits would be necessary to encode this number. A binary multiplier of corresponding size would require over 200 DSP-slices and is thus not a viable option.

The fact that such small numbers can still be significant for the outcome of the SPN and the result of the ML-task is also the reason why we use comparisons in *log-space* to compute the deviation from reference results in the rest of this work.

17.3.2 *Floating Point*

As motivated above, for applications requiring a large dynamic range, the word length w of fixed-point numbers may get excessively large. *Floating point* (FP) numbers provide a much wider dynamic range, at the cost of a reduced precision, for the same number of bits. An FP number X according to the IEEE 754 standard is represented as $X = (-1)^s \times 1.f \times 2^{e-e_0}$, where s is the sign bit (0 for positive, 1 for negative), f is the fraction and e is the exponent field. The exponent field e is a w_e bit unsigned integer that represents the signed exponent $e - e_0$, where e_0 is called the bias defined as $e_0 = 2^{w_\mathrm{e}-1} - 1$. As the FP format is normalized such that the leading bit of the significant is equal to '1', only the fractional bits of the mantissa are stored in w_m bits.

17.3.3 *Posit*

The Posit arithmetic format is a comparably young format, introduced in 2017 as an implementation of type-3 unum (universal number) arithmetic [12]. The Posit format is characterized by two parameters, the total number of bits in the format w and the number of bits used to represent the exponent w_es.

As shown in Fig. 17.2, the Posit number representation is composed of four parts.

Negative numbers are encoded as 2's complement where the most significant bit (s) indicates the sign of the number.

The next component, the so-called *regime*, distinguishes Posit from traditional floating point formats. The regime is represented using a variable run-length (or thermometer) encoding, i.e., a sequence of bits with identical value terminated by a bit of the opposite value, where the length of the sequence represents the encoded value. As an example, the sequence 0001 encodes the value -3, whereas the sequence 110 encodes the value 2.

The third component, the *exponent*, is encoded as a binary number using a fixed size of w_es bits. In contrast to IEEE754 floating point, the exponent only encodes *positive* numbers and no bias is used.

The last component is the *mantissa*, which is stored just as in IEEE754 floating point, with an implicit leading 1 omitted. The mantissa occu-

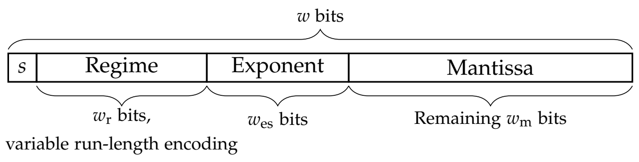


Figure 17.2: Posit binary format.

pies the remaining w_m bits, that are left after the run-length encoding of the regime and the fixed-size exponent.

Because the length of the regime is only limited by $w - 1$, the mantissa and also the exponent may not be present at all.

Given the sign bit s, a regime value r, the exponent e and the mantissa f, the number represented in Posit can be computed as follows: $(-1)^S \times \mathrm{useed}^r \times 2^e \times 1.f$, where $\mathrm{useed} = 2^{2^{w_\mathrm{es}}}$. As an example, with $w = 7$ and $w_\mathrm{es} = 2$, the bit-sequence 0␣01␣11␣10 encodes the decimal value $(-1)^0 \times (2^{2^2})^{-1} \times 2^3 \times 1.10_2 = 0.75$.

Multiple previous works have developed Posit arithmetic hardware operators for FPGAs. While [15] and [21] found that Posit incurred a significant area overhead over traditional floating point, the operators developed in [4] required resources comparable to FP implementations and for the particular application investigated in this work, floating point could be replaced with a smaller bit-width and more area-efficient Posit format. As only the operators from [15] are available open-source, we build on this library for the implementation of the Posit hardware operators in this work. We extend the operators to meet our requirements as described in Section 17.5.3.

17.3.4 *Logarithmic Number System*

Originally, Logarithmic Number Systems (LNS) were developed as an alternative to floating point numbers. The general idea behind LNS is that instead of storing a real number as a combination of an integer exponent and a fixed-point number, only the *logarithm* $\log_2(A) = E_A$ is stored as a fixed-point *exponent*. In general purpose applications, LNS-numbers are then encoded as follows: $A = -1^{S_A} \times 2^{E_A}$ and a flag is used for zero values [6, 14].

Due to the logarithmic nature of the encoding, all calculations are performed in a *logarithmic scale*. Thus, logarithmic properties apply and $\log_2(a \times b) = \log_2(a) + \log_2(b)$, greatly simplifying multiplicative calculations.

In contrast to this, additive arithmetic operations become more complex. Assuming that $x > y$ holds, addition and subtraction are given by $\log_2(x \pm y) = \log_2(x) + \log_2(1 \pm 2^{(\log_2(y) - \log_2(x))})$. The second part of the equation is usually implemented through a helper function h, and the allowed interpolation error determines how this function is implemented in hardware. In this work, we adapt the approach from [29], which was optimized for SPNs and uses a quadratic spline interpolation for h.

17.4 DESIGN-SPACE EXPLORATION USING SOFTWARE EMULATION

A fair comparison of the three arithmetic formats considered requires that the individual parameters of the different formats (e.g. overall bitwidth) are *optimized* as much as possible.

To this end, prior work such as [25] has often used theoretical worst-case analyses based on error-models for fixed- and floating-point arithmetic operators. However, these analyses tend to *overestimate* the error that occurs during actual computation. Besides that, error analysis models for Posit and LNS are not readily available and many of the application-specific optimizations to the hardware operator implementations described in Section 17.5 cannot easily be modelled in such error models.

Therefore, we take a different approach: Using a C++-based software emulation of the individual SPN and the different arithmetic formats, the design-space is traversed to determine the best *viable* configuration for each arithmetic format on a per-benchmark basis. We use the available benchmark data to run the software emulation with each configuration and only accept a configuration, if it maintains a given error-threshold. As the representativeness of the training data is key to the ML training itself, the design-space exploration will yield configurations that work for all relevant input combinations. This approach is also common when quantizing neural networks [13].

17.4.1 *Implementation*

Using a graph-based intermediate representation and an abstract syntax tree (AST) infrastructure, we generate C++ code emulating the behavior of each of the different arithmetic formats in hardware as closely as possible.

The design-space of possible configurations is then automatically traversed. For each configuration, we generate and compile the C++ code and run the SPN inference on a CPU. If the maximum error does not exceed the configurable error threshold, we accept the configuration.

The performance of the DSE can be improved significantly by investigating multiple configurations in parallel and additionally parallelizing the CPU-based execution using OpenMP. This way, we could reduce the time required to determine the correct floating-point configuration for the largest benchmark instance *NIPS80* from 656 seconds to only 138 seconds.

Table 17.1: Configuration of the three arithmetic formats for each of the benchmarks maintaining an error of 1×10^{-6}.

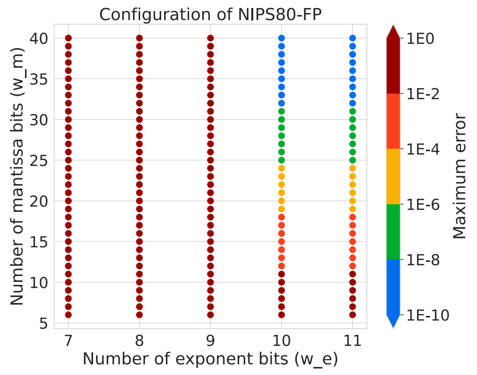
	FP		**Posit**		**LNS**		
Benchmark	w_e	w_m	w	w_es	w_I	w_F	h-Error
Accidents	8	26	36	4	7	32	21.5
Audio	9	28	36	4	8	30	20.5
MSNBC 200	8	26	32	4	7	31	19.5
MSNBC 300	8	24	32	4	7	31	20.5
Netflix	9	26	36	4	8	30	20.5
NLTCS	7	26	32	3	6	30	19.5
Plants	8	28	36	5	7	31	20.5
NIPS5	7	24	30	3	5	26	18.5
NIPS10	7	24	32	3	6	27	20.0
NIPS20	8	24	34	3	7	29	19.5
NIPS30	8	26	34	4	7	29	19.5
NIPS40	9	26	34	4	7	30	19.5
NIPS50	9	26	34	5	8	30	19.5
NIPS60	9	26	36	5	8	30	19.5
NIPS70	9	26	36	5	8	30	20.5
NIPS80	10	26	36	5	9	31	19.5

17.4.2 *Accuracy Results*

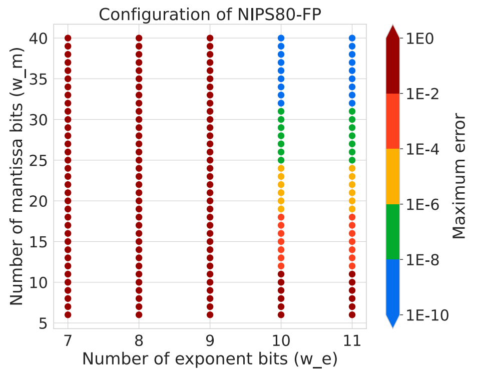
For the following accuracy evaluation, an error threshold of 1×10^{-6} was used. Note that we compute the error in log-space to determine the error independently from the magnitude of the values. For each of the arithmetic formats, different parameters can be chosen: For floating point, the number of bits in the mantissa (w_m) and the exponent (w_e) can be configured. The Posit format is parameterized by the total number of bits (w) and the number of bits used for the exponent (w_es). The LNS format can be configured by three parameters: The number of integer (w_I) and fraction (w_F) bits in the fixed-point format of the exponent and the maximum error allowed for the interpolation (Error) of the helper function h used in LNS-addition.

The configurations identified through our design-space exploration for each benchmark can be found in Table 17.1. The plots in Fig. 17.3 show how the maximum error develops across different configurations for each arithmetic format in the NIPS80 benchmark, the largest instance in our benchmark set.

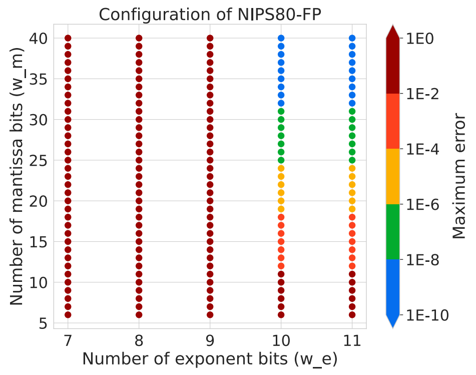
For floating point, a minimum number of exponent bits (w_e) is required to be able to represent small but significant values in the



(a) Floating point.



(b) Posit.



(c) LNS with $w_I = 9$.

Figure 17.3: Development of the maximum error depending on the configuration of the arithmetic formats. Best viewed in color.

Table 17.2: Comparison of the per-operator resource requirements and pipeline depth, using configurations for *NIPS80* (cf. Table 17.1).

Format	Op.	Slice	DSP	BRAM	pipeline depth
FP	Adder	106	0	0	5
	Mult.	86	2	0	5
Posit	Adder	374	0	0	7
	Mult.	340	4	0	12
LNS	Adder	757	20	1.5	64
	Mult.	36	0	0	3

first place. Beyond that, a certain number of mantissa bits (w_m) is required to represent numbers sufficiently accurate so the error will not accumulate beyond the error threshold.

With Posit, a minimum number of bits for the exponent (w_{es}) and the total size of the format (w) is required. However, if the size of the exponent is increased *beyond* that minimum number, the total number of bits *also* has to be increased, otherwise the number of bits remaining for the mantissa (max. $w - w_{es} - 3$) is no longer sufficient. So for Posit, the sweet spot is reached when w_{es} is just large enough to encode all relevant exponents.

The direct comparison of floating-point and Posit shows, that the total bitwidth of the formats is typically relatively close. This result aligns with the findings in [8]. The probabilistic values computed inside the SPN tree are very small, and lie outside of the *golden range* identified in [8]. In that range, relatively small Posit formats can be used to replace significantly larger floating-point formats.

The LNS format will only produce correct results, if the number of integer bits (w_I) is sufficiently large to represent all relevant exponents, therefore the plot in Fig. 17.3c shows the development of the error depending on the fraction bits (w_F) of the exponent and the interpolation error of the addition helper function h for $w_I = 9$. The number of fraction bits must be sufficiently large to represent numbers with a certain accuracy and, at the same time, the allowed interpolation error of h must be sufficiently small so the LNS addition does not introduce excessive error.

17.5 IMPLEMENTATION OF HARDWARE ARITHMETIC OPERATORS

Based on the findings from the automatic DSE presented in the previous section, specialized hardware arithmetic operators for SPN inference were developed. This section details the implementation for each arithmetic format. An overview of the resource requirements of the

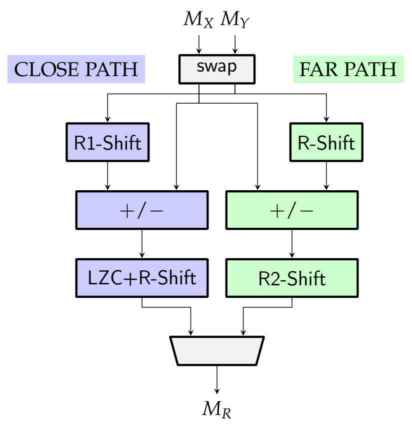


Figure 17.4: FP Adder dual path mantissa processing

individual operators can be found in Table 17.2. The operators were designed as drop-in replacement for the operators in [26] to enable reuse of the automatic toolflow in this work.

17.5.1 *Floating Point*

The floating point implementations used in this work are based on the FloPoCo tool [9] which was extended for the specifics of SPN. All of our extensions have been made publicly available in the FloPoCo git repository [7]. Note that subnormal numbers are not supported in FloPoCo as they are very costly to implement and the loss in dynamic range can be easily compensated by adding one additional mantissa bit.

17.5.1.1 *Floating Point Adder*

Addition in FP is a much more time and resource consuming operation compared to FP multiplication. The basic algorithm to perform a floating point addition requires the following computation steps: 1) computing the exponent difference, 2) alignment of the operands, 3) mantissa addition, 4) alignment and rounding of the result, and, 5) handling of special values. All these computations lie on the critical path where the large bit shifters required for the two alignment steps are among the most demanding. Also, faithful rounding does not help much for addition. However, a well-known technique to reduce the delay is the dual-path (DP) architecture [11]. The observation here is that two cases exist that can be treated separately: 1) when subtracting two numbers with similar magnitude, only a *small* operand shift is necessary while a *full* result shifter is required; 2) in all other cases, the operand shift has to be *large* while a *small* shift for the result is sufficient. In the DP adder, the computations for both cases are computed *in parallel*, and the correct result is selected at the end.

Table 17.3: Direct comparison of the FP adder before and after their optimization using the configurations for *NIPS80* (cf. Table 17.1).

Operator	Slice	DSP	PD	Freq. [MHz]
FP Adder Single Path	138	0	8	384
FP Adder Dual Path	184	0	7	274
FP Adder (only pos. args.)	106	0	5	389

Figure 17.4 shows the data path for processing the mantissa, omitting the control signals for brevity. The first case is called the *close-path* (shown on the left in blue) and the second case the *far-path* (shown on the right in green). While for the *operand* alignment only a 1-bit right shift (R1-Shift) is necessary in the close-path, a full right shifter (R-Shift) is necessary in the far-path. In contrast, the *result* of the close-path requires a leading zero counter (LZC) and full right shifter, while the the far-path only requires a 2-bit shift (R2-Shift) for normalization and rounding.

To implement SPNs, we can make use of the dual-path idea by exploiting the fact that all values in SPNs are restricted to be *positive* and only *additions* occur. Hence, the close-path in a dual-path architecture will *never* be active in an SPN. To this end, we extended the dual-path implementation of the `FPAdd` operator in FloPoCo with an option to optimize the adder only for positive numbers, which omits all components from the close-path as well as the output multiplexer.

To gauge the effects of this optimization, we performed a synthesis experiment on the single operators (using the same setup later described in Section 17.6.2). The results are given in Table 17.3, showing the logic resources, the pipeline depth (PD) as well as the max. clock frequency. As there are two options for the FP adder in FloPoCo, a single path and a dual path, we synthesized both. As expected, the dual path has one pipeline stage less compared to the single path, but at the expense of a larger chip area. Remarkably, our optimization for only positive operands (listed as "only pos. args.") leads to a slice reduction of 23.2% and 42.4% compared to the single and dual path options, respectively, while reducing the pipeline depth by 3 and 2 cycles at the same time.

17.5.1.2 *Floating Point Multiplier*

The computation of an FP multiplication is much simpler compared to addition: 1) the mantissas are multiplied, 2) the exponents are added, and, finally 3) the result is normalized and rounded. This normalization requires only a small shift by one bit position and can usually be merged with the output MUX that is necessary for the special values. Besides this, the rounding mode has the most influence on the used resources. In contrast to correct rounding, faithful rounding requires only about half the number of bits plus some guard bits of the mantissa multiplication result [2]. Hence, a truncated integer multiplier can be used for the mantissa which requires less chip area. Therefore, the FP multipliers in this work use faithful rounding based on the work in [2].

17.5.2 *Logarithmic Number System*

For the implementation of the LNS hardware operators, we employ the implementation of Weber et al., presented in [29]. They developed pipelined and parameterized LNS adders and multipliers targeted towards SPNs.

As discussed earlier, multiplication in the logarithmic space can be implemented as a simple binary addition, and consequently consumes less than half (36 vs. 86, cf. Table 17.2) of the slices compared to the floating-point multiplier, and no DSPs.

On the other hand, the much more complex calculation for addition in logarithmic space, for which a quadratic spline interpolation was used in [29], results in a larger chip area for the logarithmic adder, which consumes 757 slices and 20 DSPs, compared to 106 slices and no DSPs for FP.

17.5.3 *Posit*

For the implementation of the Posit hardware operators, we build upon PACoGen [15], an open-source project providing Posit basic arithmetic operators. These implementations are generally only realized as combinatorial circuits.

To ensure a fair comparison between the arithmetic formats regarding operating frequency, we introduced pipelining into the existing, parameterized implementation. The resulting multiplication operator requires almost five times the logic resources (340 vs. 86 slices), and, even though we adopted the optimal DSP allocation scheme from [17], twice the number of DSPs (4 vs. 2), as the floating-point multiplier. In case of the addition, the additional decoding logic for the regime and the higher internal precision cause the Posit adder to use significantly more resources than its floating-point counterpart (374 vs. 106 slices, cf. Table 17.2).

17.6 EVALUATION

17.6.1 *Benchmarks*

In order to be able to compare the performance and FPGA resource usage directly to [26], we use the same set of benchmarks. The set contains two kinds of benchmarks: Count-based examples, which are taken from the NeurIPS corpus [10] and capture information about the frequency of words in texts, and examples with binary input variables, which were pre-processed by [18] and [28] and capture statistical data, such as usage statistics of services. More detailed information on the individual benchmarks can be found in [26].

17.6.2 *FPGA Implementation Results*

We first compare the resource usage of the three different arithmetic formats for the benchmark set, using the configurations from Table 17.1. Xilinx Vivado 2019.1 and TaPaSCo 2019.10 (pre-release) are used to generate bitstreams for a Xilinx Virtex 7 FPGA device (xc7vx690), all numbers given here are taken from the post-place&route reports. We use the automatic design-space exploration feature of TaPaSCo [16] to determine the best possible frequency. All bitstreams are tested in actual hardware on a Xilinx VC709 development board, verifying that the configurations determined by our DSE (cf. Section 17.4) maintain the given error bound of 1×10^{-6}.

The FPGA implementation results are given in Table 17.4. For brevity, numbers are given relative to the entire FPGA, the absolute number of resources available are 108,300 (Slices), 1,470 (BRAM) and 3,600 (DSP), respectively.

Through our automatic design-space exploration to determine the minimum viable configuration and the optimization to the floating point operators described in Section 17.5.1, the resource usage compared to the results reported in [26], decreases by up to 57% in logic slices (avg. 38.5%) and up to 68% in DSP (avg. 62.9%). Additionally, the clock frequency increases by up to 75 MHz (avg. 46.6 MHz). The decrease in resource consumption is also depicted in Fig. 17.5.

The comparison between customized floating-point (CFP) and Posit shows that the latter requires significantly more logic (avg. +53%) and, except for benchmarks *Audio* and *Plants*, which contain a low number of adders in comparison to the number of multipliers, also twice the number of DSPs. The BRAM utilization is almost identical, the frequency is typically lower for Posit (avg. 30 MHz less) and the pipelines are notably deeper. Overall, one can conclude that Posit is less suitable for SPN inference than floating-point, probably because the numbers involved in SPN inference lie outside of the *golden range* (cf. Section 17.4.2), where Posit could make up for the additional decoding logic by using much narrower bitwidths. However, the Posit-based arithmetic still outperforms the double-precision arithmetic used in [26] by up to 22.6% in slices (avg. 9.5%) and 36% in DSP (avg. 34.2%).

When compared with floating-point, LNS requires slightly more slices (avg. +7.57%) and significantly (avg. +56%) more BRAM, which,

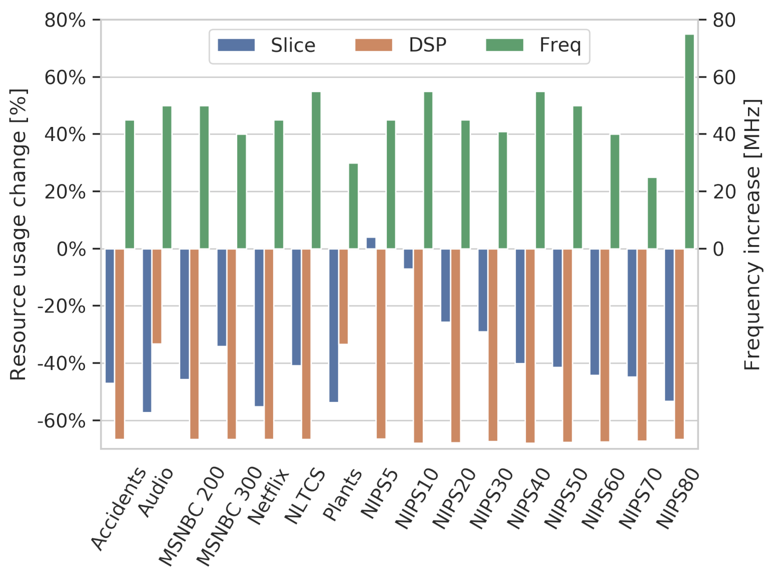


Figure 17.5: Improved resource and maximum frequency for floating-point arithmetic in comparison with prior work [26].

however, is not a critical resource in our case. The frequencies are comparable, with winners in both formats. The pipelines are much deeper, mainly due to the long latency (64 cycles) of the LNS adder. The DSP usage comparison between floating-point and LNS is highly dependent on the multiplier/adder-ratio (given as *M/A* in Table 17.4) of the examples. Only if there are roughly nine times more multipliers than adders, LNS outperforms floating-point with regard to the DSP usage (NIPS10 is an outlier, probably due to the very low DSP usage in both formats). Overall, it seems that LNS is only suitable for such SPNs with a much higher number of multipliers than adders. Yet, the LNS-based arithmetic is able to outperform the FloPoCo double-arithmetic results from [26] by up to 57.5% in slices (avg. 33.7%) and 86% in DSP (avg. 66%), in particular for examples with only a few adders.

To further validate our results, we also tested relaxed error conditions, namely 1×10^{-4} and 1×10^{-2}, for benchmarks *Accidents* and *Audio*, which were chosen because of their very different adder/multiplier-ratio. We have to omit detailed results for brevity here, but overall, the relation between LNS- and floating-point format found in the evaluation for 1×10^{-6} *persists* for relaxed error conditions: LNS is only able to save resources in comparison to floating-point, if the SPN contains very few adders compared to the number of multipliers.

Table 17.4: FPGA implementation results for all benchmarks, using the configurations from Table 17.1. Best values bold.

		Slices [%]			DSP [%]			BRAM [%]			Frequency [MHz]			Pipeline Depth		
Benchmark	**M/A**	CFP	Posit	LNS	CFP	Posit	LNS	CFP	Posit	LNS	CFP	Posit	LNS	CFP	Posit	LNS
Accidents	8	**40.19**	70.81	41.31	**12.06**	24.11	15	**3.71**	3.71	7.52	**245**	200	222	**73**	161	169
Audio	22.9	**38.22**	77.02	38.63	30.56	30.56	**6.33**	**3.71**	3.71	5.75	**250**	200	210	**73**	161	169
MSNBC 200	5.5	**35.82**	53.5	41.69	**9.17**	18.33	15.83	**3.71**	3.71	6.77	**250**	215	245	**143**	299	577
MSNBC 300	6	**30.61**	43.59	38.35	**5.67**	11.33	8.97	**3.71**	3.71	6.77	240	225	**255**	**89**	213	431
Netflix	21	39.3	67.98	**37.33**	12.83	25.67	**5.81**	**3.71**	3.71	5.75	**235**	215	220	**68**	149	166
NLTCS	5.6	**36.64**	51.38	40.51	**8.44**	16.89	13.5	**3.71**	3.71	6.46	255	220	**270**	**98**	206	342
Plants	18.3	**40.14**	74.78	41.17	28.44	28.44	**7.39**	**3.71**	3.71	6.09	230	205	**250**	**128**	278	385
NIPS5	10	**25.11**	26.59	26.17	0.56	1.11	**0.44**	3.74	**3.71**	3.84	**245**	240	240	**24**	58	60
NIPS10	8.3	**27.44**	30.71	28.09	1.39	2.78	**1.33**	3.74	**3.71**	4.18	**255**	240	**255**	**42**	96	182
NIPS20	8	**28.93**	37.88	31.28	**3.11**	6.22	3.5	**3.81**	4.01	4.69	245	200	**270**	**46**	108	203
NIPS30	8.7	**32.85**	44.87	35.43	**4.83**	9.67	5	**3.74**	4.01	4.93	240	220	**250**	**63**	132	209
NIPS40	7.6	**34.48**	50.35	39.42	**6.78**	13.56	7.56	**3.91**	4.08	5.95	255	215	**265**	**63**	132	224
NIPS50	8.9	**36.86**	54.99	42.42	**7.94**	15.89	8.44	**3.98**	4.15	6.09	**250**	215	245	**68**	144	227
NIPS60	12	**38**	59.91	40.34	8.67	17.33	**6.86**	**4.32**	4.46	6.19	**240**	215	235	**63**	132	224
NIPS70	12.9	**41.75**	67.86	43.12	10	20	**7.39**	**4.05**	4.35	6.97	**225**	203	210	**73**	156	230
NIPS80	8.3	**44.83**	82.84	47.52	**14.72**	29.44	20.44	**3.98**	4.63	7.72	**255**	195	230	**83**	211	306

Number of LUTs, FFs, BRAM slices and DSP blocks are given as percentage for brevity. The total number of available resources are 433200, 866400, 1470 and 3600 respectively.

Table 17.5: Power consumption of the datapath.

	Power Consumption [Watt]			
Benchmark	[26]	CFP	Posit	LNS
Accidents	12.493	**3.069**	5.267	3.721
Audio	18.427	5.518	8.358	**2.818**

17.6.3 *Power Evaluation*

Next to the required chip area, we are also interested in the impact of the arithmetic format onto power consumption.

In order to investigate the power consumption of the different arithmetic formats, we consider only the datapath itself, leaving out the memory infrastructure and TaPaSCo platform infrastructure. We again run synthesis and P&R for the Xilinx VC709 board using Vivado 2019.1. Afterwards, we use Mentor Questasim 2019.2 to run a *post-implementation timing simulation* to capture signal activity information from a run with actual inference input data. Using this activity information, we then use the Vivado 2019.1 power analysis for an estimate of the power consumption of the datapath.

As the post-implementation timing simulation can take several days for larger circuits, we again limit our investigation to the two benchmark instances *Accidents* and *Audio*, that we selected for the reasons described in the previous section. In addition to the three arithmetic formats investigated in this work, we also conduct the measurement for the double-precision FloPoCo-format from prior work [26].

The results from the power analysis (Table 17.5) align with our findings for the chip area in the previous sections: In the benchmark instance *Accidents*, where the customized floating-point was the most area-efficient format, it also requires the least power, followed by LNS. For the benchmark instance *Audio*, where LNS was the most area-efficient format due to the low number of adders in the SPN, LNS also requires the least power. Just as before, Posit is not able to keep up with the two other formats with regard to power usage.

Compared to the double-precision format from prior work, the SPN-optimized arithmetic formats developed in this work are able to save significant amounts of power.

17.6.4 *Performance Evaluation*

In this section, we evaluate the performance of three arithmetic formats implemented on the FPGA and compare it to a CPU and GPU-based implementation of SPN inference.

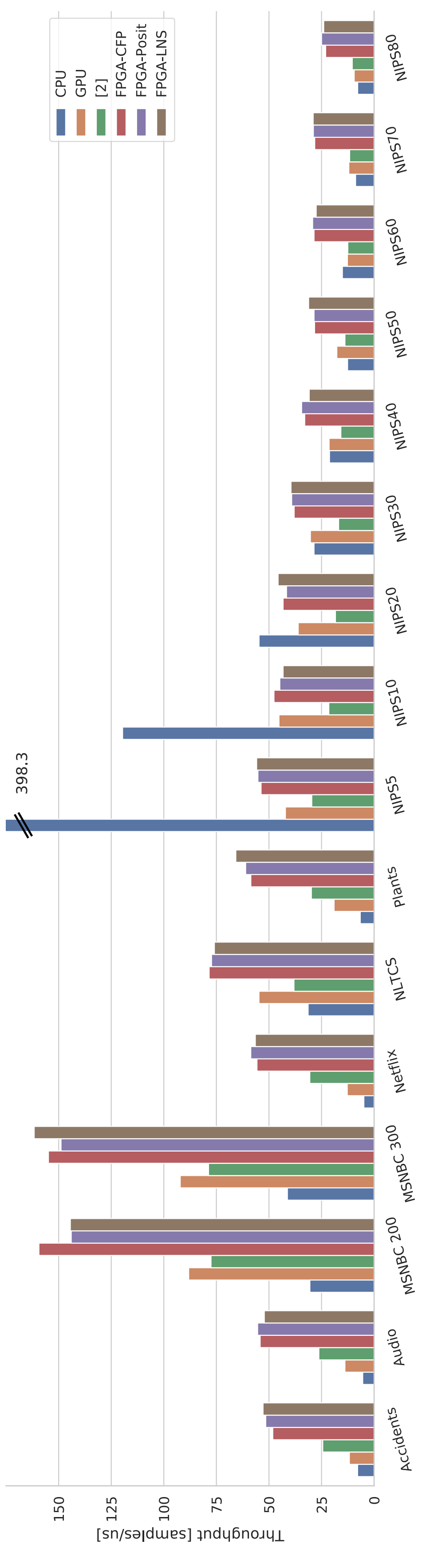


Figure 17.6: Throughput of the CPU, GPU and FPGA-implementations in samples/μs. Each group represents an example SPN. The single outlier is the CPU throughput for example NIPS5 which amounts to 398.8 samples/μs.

17.6.4.1 *CPU & GPU Baseline*

Based on the compiler infrastructure that we created for the design-space exploration (cf. Section 17.4), we additionally built a custom compilation flow mapping an SPN description to optimized C++ and CUDA-code, both using double-precision floating-point arithmetic. In both cases, we compiled using `-O3` and `-ffast-math` to enable aggressive compiler optimizations. Our C++ compilation flow on an AMD Ryzen 1600X performs on par with the CPU-baseline from [26], and our CUDA compilation flow is able to *outperform* the original Tensorflow-based GPU-mapping from [26] by a factor of up to 90x on a Nvidia 1080Ti GPU.

17.6.4.2 *Performance Comparison*

For the comparison, we run the inference on the VC709 development board, coupled with an AMD Ryzen 1600X. Our measurements of the throughput in Fig. 17.6 also include the time required to *transfer* the data between host and FPGA.

For the three smallest count-based samples (NIPS5-20), the CPU provides the best throughput. For these small networks the overhead for data-transfer to the accelerator (GPU or FPGA) clearly dominates the execution time. With our optimized CUDA compilation flow, the GPU provides better throughput than the CPU for the remaining benchmarks, in particular for the binary examples.

Despite the large differences in the pipeline-depth (cf. Table 17.4), the performance for the three arithmetic formats implemented on the FPGA varies only slightly. Overall, all three versions deliver very similar performance (with an overall difference of less than 2%). Compared to the previous FPGA implementation in [26], the new formats provide better throughput (geo.-mean. 2.1x speedup). This is partly due to the higher operator frequencies, but also caused by improvements to the underlying TaPaSCo framework.

All three formats significantly *outperform* the CPU. Except for the three benchmarks mentioned earlier, the speedup reaches as high as factor 12x (geo.-mean 2.5x). The three FPGA versions also provide significantly *higher throughput* than the GPU-based implementation, here, the speedups reach up to 4.6x (geo.-mean. 2.1x).

Again, note that our measurements include the PCIe data-transfer to the FPGA memory. On shared-memory systems such as Zynq MPSoC, the speedup over the CPU and the GPU would reach up to 37x and 14x, respectively.

17.7 CONCLUSION & OUTLOOK

In this work, we have investigated three different arithmetic formats with regard to their suitability for Sum-Product Network Inference on FPGAs. We have developed an automatic design-space exploration framework, which allows us to efficiently identify the minimum bitwidth required for each of the formats to maintain a given error margin. Based on the findings from the DSE, hardware arithmetic operators, optimized for SPN inference, for each of the formats were implemented.

Our evaluation shows that customized floating-point is the most resource-efficient format for SPN inference, and is only outperformed by a logarithmic number system format for SPNs with *very few* adders compared to the number of multipliers. All three investigated arithmetic formats deliver almost identical performance and significantly outperform CPU and GPU-based implementations of SPN inference, by factors up to 12x and 4.6x, respectively.

In future work, we will investigate how the hardware arithmetic operators can be optimized further, e.g., by using fused operators.

ACKNOWLEDGEMENTS

The authors would like to thank Xilinx Inc. for supporting their work by donations of hard- and software. Calculations for this research were conducted on the Lichtenberg high performance computer of TU Darmstadt.

Finally, the authors would like to thank Kristian Kersting and Alejandro Molina, for much appreciated discussions of the subject and insights into the matter of Sum-Product Networks.

REFERENCES

[1] JD Alves, JF Ferreira, J Lobo, and J Dias. "Brief survey on computational solutions for Bayesian inference." In: *Unconventional comp. for Bayesian inference*. 2015.

[2] Sebastian Banescu, Florent de Dinechin, Bogdan Pasca, and Radu Tudoran. "Multipliers for Floating-Point Double Precision and Beyond on FPGAs." In: *SIGARCH Computer Architecture News* 38.4 (Sept. 2010), pp. 73–79.

[3] Jessa Bekker, Jesse Davis, Arthur Choi, Adnan Darwiche, and Guy Van den Broeck. "Tractable Learning for Complex Probability Queries." In: *NIPS*. 2015.

[4] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers. "Parameterized Posit Arithmetic Hardware Generator." In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. Oct. 2018.

[5] Jungwook Choi and Rob A. Rutenbar. "Video-Rate Stereo Matching Using Markov Random Field TRW-S Inference on a Hybrid

CPU+FPGA Computing Platform." In: *IEEE Trans. Circuits Syst. Video Techn.* (2016).

[6] J. Detrey and F. de Dinechin. "A VHDL library of LNS operators." In: *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*. Nov. 2003.

[7] Florent de Dinechin. *FloPoCo Project Website*. URL: http://flopoco.gforge.inria.fr.

[8] Florent de Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. "Posits: The Good, the Bad and the Ugly." In: *Proceedings of the Conference for Next Generation Arithmetic 2019*. CoNGA'19. New York, NY, USA: ACM, 2019, 6:1–6:10. ISBN: 978-1-4503-7139-1. (Visited on 07/04/2019).

[9] Florent de Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo." In: *IEEE Design & Test of Computers* 28.4 (2011), pp. 18–27.

[10] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[11] Paul Michael Farmwald. "On the design of high performance digital arithmetic units." PhD thesis. Stanford University, 1981.

[12] John L. Gustafson and Isaac T. Yonemoto. "Beating Floating Point at its Own Game: Posit Arithmetic." In: 4 (Apr. 2017). ISSN: 2313-8734.

[13] Song Han, Huizi Mao, and William J Dally. "Deep Compression: Compressing Neep Neural Networks with Pruning, Trained Quantization and Huffman coding." In: (2015).

[14] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, and K. S. Hemmert. "A comparison of floating point and logarithmic number systems for FPGAs." In: *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*. Apr. 2005, pp. 181–190. DOI: 10.1109/FCCM.2005.6.

[15] M. K. Jaiswal and H. K. H. So. "PACoGen: A Hardware Posit Arithmetic Core Generator." In: *IEEE Access* 7 (2019), pp. 74586–74601. ISSN: 2169-3536.

[16] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems." In: *Applied Reconfig. Comp.* 2019.

[17] M. Kumm, J. Kappauf, M. Istoan, and P. Zipf. "Resource Optimal Design of Large Multipliers for FPGAs." In: *2017 24th Symp. on Computer Arithmetic*. July 2017.

[18] Daniel Lowd and Jesse Davis. "Learning Markov network structure with decision trees." In: *Data Mining (ICDM), 2010 IEEE 10th International Conf.* 2010.

[19] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Floriana Esposito, Siraam Natarajan, and Kristian Kersting. "Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains." In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2018.

[20] Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro Domingos. "On Theoretical Properties of Sum-Product Networks." In: *Proc. of AISTATS*. 2015.

[21] A. Podobas and S. Matsuoka. "Hardware Implementation of POSITs and Their Application in FPGAs." In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2018.

[22] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: a New Deep Architecture." In: *Proc. of UAI* (2011).

[23] Andrzej Pronobis, Francesco Riccio, and Rajesh PN Rao. "Deep spatial affordance hierarchy: Spatial knowledge representation for planning in large-scale environments." In: *ICAPS 2017 Workshop on Planning and Robotics*. 2017.

[24] Martin Ratajczak, Sebastian Tschiatschek, and Franz Pernkopf. "Sum-Product Networks for Sequence Labeling." In: *CoRR* (2018). arXiv: 1807.02324.

[25] Nimish Shah, Laura I. Galindez Olascoaga, Wannes Meert, and Marian Verhelst. "ProbLP: A Framework for Low-precision Probabilistic Inference." In: *56th Annual Design Automation Conference*. DAC '19. Las Vegas, NV, USA: ACM, 2019. ISBN: 978-1-4503-6725-7.

[26] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-Based Accelerators." In: *36th Intl. Conf. on Computer Design (ICCD)*. Oct. 2018, pp. 350–357.

[27] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. "Automatic Synthesis of FPGA-based Accelerators for the Sum-Product Network Inference Problem." In: *ICML 2018 Workshop on Tractable Probabilistic Models (TPM)*. 2018.

[28] Jan Van Haaren and Jesse Davis. "Markov Network Structure Learning: A Randomized Feature Generation Approach." In: *AAAI*. 2012, pp. 1148–1154.

[29] Lukas Weber, Lukas Sommer, Julian Oppermann, Alejandro Molina, Kristian Kersting, and Andreas Koch. "Resource-Efficient Logarithmic Number Scale Arithmetic for SPN Inference on FPGAs." In: *International Conference on Field-Programmable Technology (FPT)*. 2019.

[30] Han Zhao, Mazen Melibari, and Pascal Poupart. “On the Relationship between Sum-Product Networks and Bayesian Networks.” In: *Proc. of ICML*. 2015.

18 SPNC: FAST SUM-PRODUCT NETWORK INFERENCE

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *SPNC: Fast Sum-Product Network Inference* by *Lukas Sommer, Cristian Axenie and Andreas Koch* in *2021 International Workshop on IoT, Edge, and Mobile for Embedded Machine Learning (ITEM)*. The contribution of the author of this thesis is summarized as follows.

> *» As the corresponding and leading author, Lukas Sommer was responsible for the design of the MLIR-based compilation flow, the corresponding SPN-specific dialects, and mapping and lowering strategies. He was also in charge of the implementation of the complete compilation flow and the interface to the open-source library SPFlow, and performed the experiments. The text was joint work by Cristian Axenie and Lukas Sommer, with feedback from Andreas Koch. «*

ABSTRACT

Sum-Product Networks have received increasing attention from academia and industry alike, but the software ecosystem is comparably sparse. In this work, we enhance the ecosystem with an open-source, domain-specific compiler that allows to easily and efficiently target CPUs and GPUs for Sum-Product Network inference. The implementation of the compiler is based on the open-source MLIR framework.

Using a real-world application of Sum-Product Networks, a robust speaker identification model, we showcase the performance improvements our compiler can achieve for SPN inference on CPUs and GPUs.

18.1 INTRODUCTION

Probabilistic models are receiving increasing attention from both academia and industry, being a complementary alternative to more widespread machine learning approaches such as (deep) neural networks (NN). Probabilistic models can handle the *uncertainty* found in real-world scenarios better, and are also, in contrast to NNs, able to *express uncertainty* over their output.

However, in contrast to neural networks, for which a rich ecosystem with a variety of frameworks, libraries and compilers, such as Tensorflow's XLA, or Facebook's Glow, is available, the ecosystem for probabilistic models such as Sum-Product Networks (SPN) is comparatively sparse, due to them being a relatively young class of models.

One of the most popular libraries for research with Sum-Product Networks is SPFlow by Molina et al. [3], which provides a programmatic representation of Sum-Product Network models and allows to learn their structure and parameters from data. SPFlow also allows to perform inference on the models obtained through learning, but is implemented in pure Python and can therefore not leverage the full feature set of CPUs or GPUs. However, exploiting all available hardware features is crucial for the deployment of SPN models on embedded-grade devices and for efficient inference in real-world applications, e.g., to fulfill real-time requirements.

Therefore, in this work, we enhance the SPN ecosystem by developing *SPNC*, an *open-source* domain-specific, multi-platform compiler for performing fast Sum-Product Network inference on CPUs and GPUs, and present the following contributions:

- Based on the MLIR framework [2], we develop custom high-level intermediate representations capturing the semantics of Sum-Product Networks in the compiler (Section 18.3.1).

- We define efficient strategies to map Sum-Product Network inference to CPUs with vector extensions and CUDA GPUs. The mapping strategies make use of the underlying hardware's specific features for efficient inference (Section 18.3.2 & Section 18.3.3).

- Using a real-world application of Sum-Product Networks, we evaluate our approach in detail, and compare it to the currently available framework (Section 18.5).

- We develop a Python interface to our compiler, which seamlessly integrates with SPFlow [3] and allows to target CPUs and GPUs with ease (Section 18.4).

Furthermore, we provide necessary background information on Sum-Product Networks and the open-source MLIR framework in the next section, and discuss related works in Section 18.6.

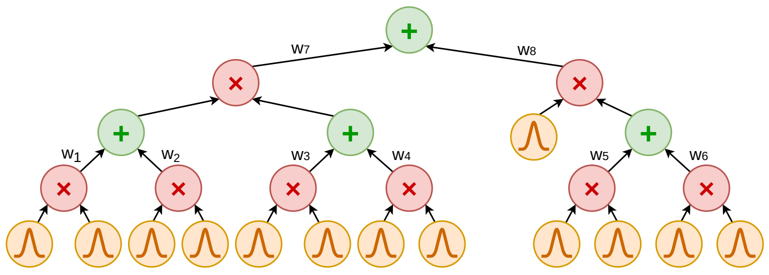


Figure 18.1: Example of a Sum-Product Network graph.

18.2 BACKGROUND

18.2.1 *Sum-Product Networks*

SPNs [7] are a relatively young class of probabilistic graphical models (PGM). In principle, such class of models can be considered a unified approach combining Bayesian network representation formalism and Markov random field computation. Such a computational configuration enables SPNs to efficiently reason under incompleteness and uncertainty, which is a challenging task in many real-world scenarios [11]. Unfortunately, inference requires intense computation that introduces a long delay. This is a core motivation of our work.

Additionally, in contrast to most neural network architectures, SPNs are also able to quantify uncertainty over the output. An example for this property can be found in [6], where SPNs, when confronted with out-of-domain images, indicate this through a low likelihood for the output class, in contrast to the multi-layer perceptron undergoing the same test. An overview of other practical usage examples of SPNs can be found in the survey by Paris et al. [5].

Sum-Product Networks capture the joint probability of a set of variables (i.e., features) in the form of a directed acyclic graph (DAG). Regardless of the application and the underlying data, the DAG is always composed from three different types of nodes. At the bottom of the DAG, so-called *leaf nodes* capture the univariate probability distribution of a single variable/feature. Depending on the type of data (e.g., continuous vs. discrete), and underlying distribution of the data, different probability distributions can be used, e.g., Gaussian distribution for continuous variables, or a categorical distribution for discrete values. The so-called *scope* of a leaf node is the single variable associated with it. Further up in the graph, a combination of product nodes and weighted sum nodes is used to capture the joint probability distribution. Product nodes represent factorizations of independent variables. For the SPN to be valid, the scopes of the different child nodes of a product node must be *disjoint*. Weighted sum nodes, on the other hand, represent a mixture of distributions and the scopes of all child nodes must be *identical* in a valid SPN. The structure of the SPN depends on the distribution of the underlying data, and can either be learned from data, or be hand-crafted, just followed by parameter learning. An overview of SPN learning algorithms can be found in [5]. A small example of an SPN graph is shown in Fig. 18.1. As a core functional principle, the SPN decomposes complex multivariate "global" functions by exploiting the way in which the global function factors into a product of simpler "local" functions of a subset of the variables [1]. SPN can be used to solve machine learning tasks, such as classification, by performing inference on the underlying DAG. In general, SPNs support multiple different types of inference. In this study, we are focusing on two of these types, namely joint probability inference and marginal inference. Joint probability inference is used to obtain the joint probability given *full* evidence (i.e., a value for each variable). To this end, the evaluation of the SPN DAG starts by evaluating the distribution of the leaf nodes, given the value of the variable associated with each of them. After that, the values are propagated upwards through the DAG, performing multiplication or weighted addition at the product and sum nodes, until a final probability value is obtained at the root node of the SPN. Marginal inference, on the other hand, is used when only *partial* evidence is available. Leaf nodes for which no evidence is available are set to 1, the remaining ones are evaluated just as in joint inference, and the propagation of values through the SPN is performed analogously to the description above.

The compiler developed in this study aims to accelerate the *inference* in Sum-Product Networks by efficiently mapping them to different hardware targets. Learning of the SPN is assumed to have taken place beforehand, using a standard Sum-Product Network framework such as SPFlow [3].

18.2.2 *MLIR*

The implementation of SPNC in this work is heavily based on the open-source MLIR framework [2][1]. Therefore this section presents a brief overview of MLIR.

MLIR aims to facilitate the implementation of compilers by providing an *extensible* framework for the implementation of multi-level IRs. The main reason for adding multiple levels of abstractions into compilers is that an early lowering to a *low-level* intermediate representation such as LLVM IR loses too much of the high-level structure of the program, which later on must be reconstructed using often fragile approaches based on the low-level IR in order to perform transformations, e.g., on loops. Capturing additional information and potentially domain-specific semantics in one or multiple high-level IRs enables the compiler to perform more powerful program transformations.

1 https://mlir.llvm.org

Because MLIR provides common components for the implementation of IRs, such as pass managers and common transformations, users can focus on the design of the IR itself.

In order to not impose too many constraints on the semantics of different IRs, MLIR defines a minimal set of *generic* abstractions that must be used by all IRs. Similar to most modern compilers, MLIR uses the static single assigment (SSA) form, with *operations* (short: Ops) consuming and producing *values*. All values are typed, with the type system being extensible, while also defining a number of common types. So-called *attributes*, which are also typed, can additionally be used to attach compile-time information to operations.

Operations, types, and attributes are organized in so-called *dialects*, which do not add any semantics, but are a mere logical unit for the organization of the IR. Dialects can be mixed in the same logical unit, and so-called *lowerings* translate *between* different dialects, with intra-dialect transformations also being available. Typically, a progressive, step-wise lowering from a high-level dialect to lower-level dialects is used to compile for a specific target, e.g., a CPU. To enable the implementation of common transformations, MLIR uses the notion of *traits* and *interfaces* that can be attached to operations, and provides generic interfaces for transformations such as constant folding.

MLIR's extensible nature allows us to design *custom* high-level IRs. We use these to represent Sum-Product Networks in the compiler developed in this work, while we can rely on the *common* infrastructure and dialects provided with MLIR to efficiently target different hardware platforms.

18.3 approach

The aim of SPNC is to automatically compile Sum-Product Networks and probabilistic queries operating on them to executable kernels. Compiling individual SPNs allows to employ *all* hardware features available on the target platform for fast inference, for example vector extensions present on most modern CPUs. Currently, SPNC supports two main targets:

- **CPUs:** Being based on MLIR and LLVM, SPNC can target any CPU for which a backend is present in LLVM. Vector extensions are currently supported on x86 (AVX, AVX2, AVX-512) and Arm (Neon Advanced SIMD) CPUs.
- **GPUs:** The flow currently supports Nvidia CUDA GPUs, but the generic GPU abstractions of MLIR would allow to target other GPUs with comparably few changes.

The compilation flow for both targets is based on MLIR. To this end, two SPN-specific MLIR dialects have been designed and implemented, which will be described in Section 18.3.1. Starting from these dialects, the target-specific lowerings will create an executable, using the flows described in Section 18.3.2 and Section 18.3.3. The user interface and some implementation details are described in Section 18.4.

18.3.1 *MLIR Dialect Design*

The first of the two SPN-specific dialects that SPNC employs during compilation, called **HiSPN**, captures the DAG structure of a Sum-Product Network and the information about the query to perform on a high level of abstraction. It was designed to closely match the representation used internally by the SPFlow framework [3], similar to how an abstract syntax tree captures a general-purpose programming language on a high level of abstraction.

In the HiSPN dialect, an abstract probability type is used for values inside the SPN DAG, allowing SPNC to delay the decision on the actual data type used for computation and take graph characteristics into account.

The second SPN-specific dialect, called **LoSPN**, represents the actual computation that needs to be performed to process the requested query on the SPN. The top-level unit in this dialect is a *Kernel*, comprising one or multiple *Tasks*. A Task does not only represent (parts of) the SPN DAG structure, including weighted sum, product, and leaf nodes, but also contains information about which inputs values will need to be accessed and which outputs will be produced as intermediate or final result. In contrast to HiSPN, LoSPN uses a concrete type for the values. To represent the computation in log-space, which commonly used to avoid arithmetic underflow in Sum-Product Network inference, a SPN-specific data type was added to the LoSPN dialect.

The lowering from HiSPN to LoSPN is currently identical for both flows, targeting CPU and GPU. In this step, the necessary computation for the query is derived from the SPN DAG structure and query information captured by HiSPN and is the lowered into the operations of the LoSPN dialect. After lowering, the LoSPN representation undergoes a number of transformations, including steps such as common subexpression elimination (CSE), followed by the target-specific lowerings to dialects provided by the MLIR framework described in the next two sections.

18.3.2 *CPU Compilation Flow*

The compilation flow for the CPU starts with the serialized SPN model (cf. Section 18.4), an overview is shown in Fig. 18.2. After deserialization to the HiSPN dialect, lowering to LoSPN, and the transformations on the LoSPN dialect have been performed, the IR is again lowered to dialects provided as part of the MLIR framework. The Kernel and the Tasks in the LoSPN dialect are lowered to functions,

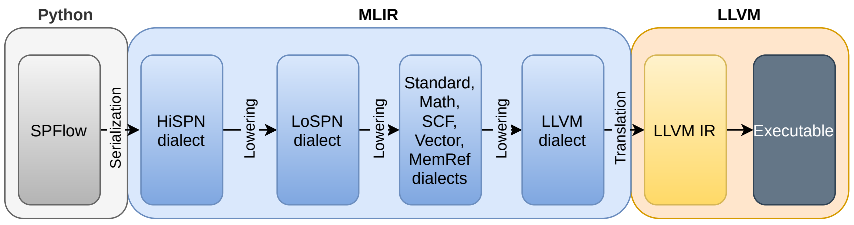


Figure 18.2: CPU Compilation Flow.

with the Kernel function calling the functions for the individual Tasks and the Task functions iterating multiple inputs for batch processing.

The operations contained inside each Task are lowered to a combination of different dialects (Note that MLIR allows to mix operations from different dialects in the same function/module):

- **Standard dialect:** Contains operations such as simple addition or multiplication on arbitrary data-types, including vectors.
- **Math dialect:** Elementary math functions, such as the `exp` and `log` function, are represented by operations from this dialect.
- **SCF dialect:** Operations from this dialect represent structured control flow, e.g. for-loops.
- **MemRef dialect:** Contains facilities to handle memory, e.g., allocation or store/load operations.
- **Vector dialect:** Vector specific operations, e.g., vector lane shuffling.

The combination of the Vector dialect and the Standard operation's ability to handle vector data-types lets the compiler exploit the CPU's SIMD extensions, if present, for maximum efficiency. In contrast to a generic loop vectorization, a domain-specific compiler such as SPNC can, thanks to the MLIR framework, leverage high-level information to generate more efficient code, e.g., by employing a combination of simple vector loads and shuffles instead of expensive gather loads.

After some transformation passes provided by the MLIR framework, all dialects are lowered to the LLVM dialect and then translated to LLVM IR, so LLVM can produce the final executable. As part of this process, the executable is also linked with vector libraries, providing optimized implementations of elementary math functions (e.g., `exp`) for vector code. The currently supported vector libraries are Intel SVML and Libmvec for x86 CPUs, and ARM Optimized Routines for ARM Neon.

Although it is technically possible to perform within the MLIR framework, we have decided to implement *multi-threading* in the runtime-component (cf. Section 18.4) rather than directly in the generated code. This allows to adopt the threading behavior dynamically, e.g., when executing multiple compiled kernels concurrently.

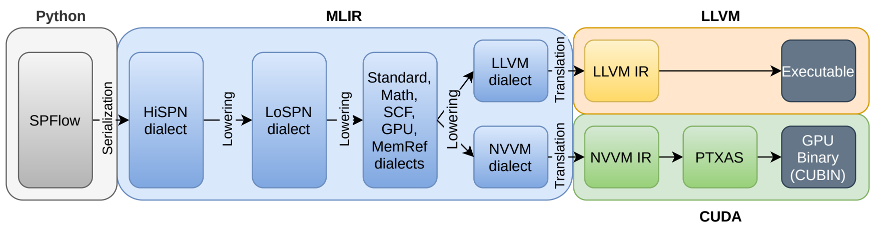


Figure 18.3: GPU Compilation Flow.

18.3.3 *GPU Compilation Flow*

Similar to the CPU compilation flow, the GPU compilation flow also starts from the serialized SPN model and performs the same steps up to the lowering of LoSPN to dialects from the MLIR framework. Here, for the GPU, the Kernel is lowered into a function, which will remain on the host CPU and will be responsible for GPU/CPU data transfers and the invocation of the Tasks, which, in contrast to the CPU flow, are lowered into *device* functions executing on the GPU.

For the operations inside the Tasks, a similar combination of MLIR-provided dialects is used, with one notable difference: Instead of the Vector dialect, the **GPU** dialect is used to represent the SIMT execution model, with operations for access to block and thread identifiers and for representation of GPU device functions and runtime functions for memory & execution management.

After that step, the GPU- and host portion of the IR are separated into two compilation units. While the flow for the host portion via LLVM is very similar to the CPU flow, eventually resulting in an executable, the GPU portion of the code is translated in multiple steps to NVVM IR, PTX assembly and a GPU binary (CUBIN format). This GPU binary is then loaded at runtime by the host function to execute inference on the GPU.

An overview of the overall compilation flow for the GPU is shown in Fig. 18.3.

18.4 PYTHON INTERFACE & IMPLEMENTATION

In order to make the compiler and the execution of the compiled binaries via the runtime component accessible to machine learning experts working with the SPFlow library, SPNC offers a Python-based interface to the compiler and runtime. In this manner, machine learning experts can create the SPNs using their familiar tools from the SPFlow library and feed their results to the compiler.

Fig. 18.4 shows an usage example of the Python interface, the example SPN is taken from SPFlow's documentation. Location (1) in the code shows how inference is usually performed in SPFlow, by invoking `log_likelihood`.

```python
import numpy as np
from spn.structure... import ...

# Create an example SPN
p0 = Product(children=[Categorical(p=[0.3, 0.7], scope=1),
                       Categorical(p=[0.4, 0.6], scope=2)])
...
spn = Sum(weights=[0.4, 0.6], children=[p2, p4])

# Create some random test data
...
test_data = np.c_[a, b, c].astype("float32")

# Perform inference using SPFlow
from spn.algorithms.Inference import log_likelihood
spflow_results = log_likelihood(spn, test_data)            # Location (1)

# Compile for CPU and perform inference
from spnc.cpu import CPUCompiler
cpu_results = CPUCompiler().log_likelihood(spn, test_data)  # Location (2)

# Compile for CUDA GPU and perform inference
from spnc.gpu import CUDACompiler
gpu_results = CUDACompiler().log_likelihood(spn, test_data) # Location (3)
```

Figure 18.4: Python interface usage example.

The other two locations show the invocation of SPNC for compilation and execution on the CPU (2) or CUDA GPU (3). In both cases, the invocation of `log_likelihood` on the compiler will first compile the SPN using the respective flow described in Section 18.3.2 and Section 18.3.3 and then execute inference using the compiled kernel. A small runtime component part of SPNC is responsible for loading the compiled kernel and executing inference. In case of CPU execution, the runtime is also responsible for multi-threaded execution using OpenMP. The Python interface also supports separate compilation and execution, so an SPN only needs to be compiled once to repeatedly perform inference.

Similar to SPFlow, the compiled kernels also support marginalized inference by passing `NaN` as input value for marginalized variables.

The Python interface is implemented using Pybind11[2]. As Pybind11 has full support for numpy arrays, input data for execution can simply be provided as numpy arrays, and the result data will likewise be returned as a numpy array. For efficient exchange of SPN models between the Python interface and the compiler, implemented in C++,

2 https://github.com/pybind/pybind11

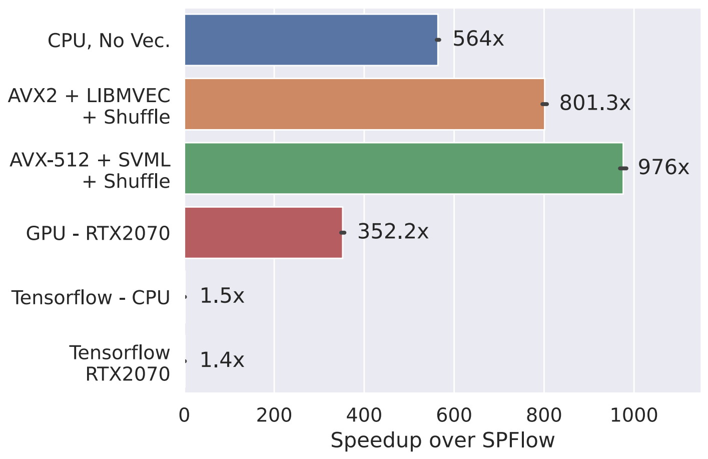


Figure 18.5: Performance comparison for clean speech samples on non-embedded systems, given as speedup over execution in SPFlow.

a binary serialization based on the open-source Cap'n Proto[3] library was implemented.

18.5 EVALUATION

To demonstrate SPNC's ability to target different heterogeneous systems, we are evaluating it on two different systems: As an example of an embedded-grade device, a Nvidia Jetson AGX Xavier device with 6-core ARM v8 CPU and Volta GPU will be used. As a non-embedded device, a machine with an AMD Ryzen 9 3900XT CPU equipped with 32 GB RAM and an Nvidia RTX 2070 Super GPU with 8 GB RAM will be used. As the Ryzen processor does not support AVX-512, experiments for AVX-512 will be performed on a dual-socket system with two Intel Xeon Platinum 9242 CPUs and 384 GB RAM.

As a real-world application of SPNs, an SPN-based automatic speaker identification from [4] is used as example application. Based on the open-source release by Nicolson et al.[4], we evaluate two different scenarios, namely the clean speech samples (245567 samples) and noisy speech samples with marginalization (1227835 samples). A sample comprises 26 features, each encoded as single-precision floating point value. We use computation in log-space to avoid deviation from the original result, using single-precision floats as the underlying data type. The implementation by Nicolson et al. contains an SPN *per speaker*, so a set of 628 different SPNs is used for evaluation.

In all experiments using our compiler, we measure the execution time from Python, i.e., the execution time always also includes the invocation overhead of the Python interface in addition to the actual execution time. We track compilation time and execution time sepa-

3 https://capnproto.org/
4 https://github.com/anicolson/SPN-ASI

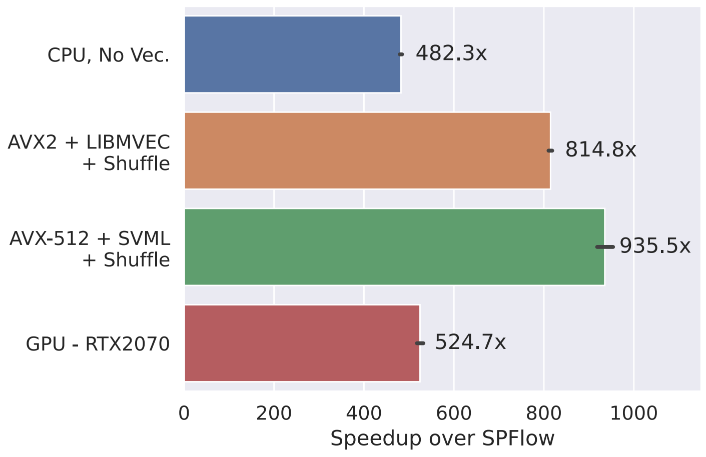


Figure 18.6: Performance comparison for noisy speech samples on non-embedded systems, given as speedup over execution in SPFlow.

rately (also for Tensorflow). The average compilation time across all platforms for CPU is 7 seconds (max. 33s) and for GPU 2s (max. 5s). The translation of the SPFlow graph to a Tensorflow graph, which is provided by the SPFlow framework, takes 18 seconds on average (max. 61s).

18.5.1 *Non-Embedded Systems*

Figs. 18.5 and 18.6 show the performance comparison for the non-embedded systems, the numbers are given as speedup over the inference execution with SPFlow.

The speedup achieved by translating the SPFlow graph to a Tensorflow graph is relatively low on both CPU (geo.-mean 1.5x) and GPU (1.38x), as the graph is still broken down into individual operations that are launched through the Tensorflow runtime. Marginalization is currently not supported by the Tensorflow translation in SPFlow, therefore no bars are shown for Tensorflow in Fig. 18.6.

SPNC on the other hand achieves speedups of 564x and 482x by compiling for the CPU and multithreaded execution, without employing vector extensions. If the vector extensions and vector libraries for elementary functions (Libmvec for AVX-2 and Intel SVML for AVX-512) are used additionally, the speedup increases to 801/814x and 976/935x, respectively. The compilation for the GPU also achieves a significant speedup of 352x and 524x, but data movements between host and device in both cases make up for more than 60% of the execution time, so even though the execution on the GPU itself is very fast, the data movement overhead, which is not present when compiling for CPU, limits the speedup.

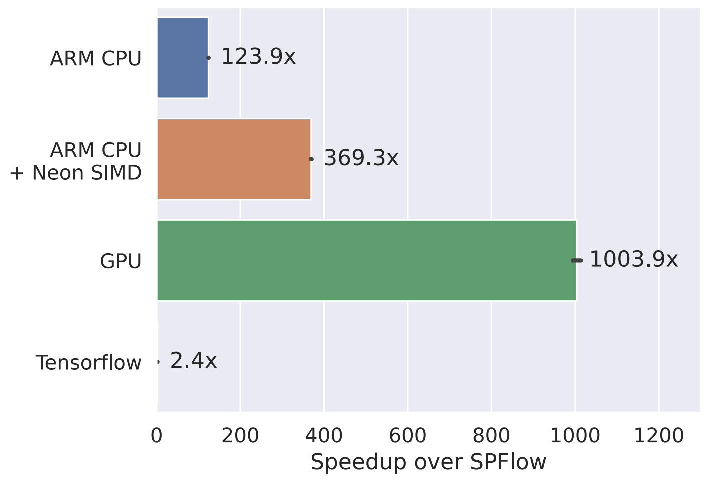


Figure 18.7: Performance comparison for noisy speech samples on embedded systems, given as speedup over execution in SPFlow.

18.5.2 *Embedded System*

Figs. 18.7 and 18.8 show the same comparison for the embedded-grade Jetson Xavier platform. As there are fewer CPU cores available than on the Ryzen/Xeon CPU, the speedup achieved by compilation for CPU is smaller compared to Figs. 18.5 and 18.6, but still reaches 124x (clean) and 58x (noisy) compared to SPFlow. When using the Neon Advanced SIMD extensions, the speedup increases by 2.9x/2.3x to 369x and 133x. In contrast to Figs. 18.5 and 18.6, the GPU compilation on the Xavier platform provides better performance than the CPU compilation. This is due to the fact that GPU and CPU *physically* share the same memory and no memory transfers between host CPU and GPU are necessary. With the memory transfers eliminated, our GPU compilation achieves speedups of 1004x and 784x.

Another important aspect on embedded systems is memory usage: For the noisy speech samples, it is not possible to process all samples in one batch with SPFlow, as the SPFlow inference runs out of memory (16 GB) and the input has to be processed in multiple blocks sequentially. The compiled kernels are much more memory-efficient and allow to process all samples in a single invocation.

For the Tensorflow comparison on this platform, the Tensorflow package officially provided by Nvidia for Jetson platforms is used. Similar to Figs. 18.5 and 18.6, the translation provides a speedup over SPFlow (2.36x), but is still significantly slower than the compiled executables.

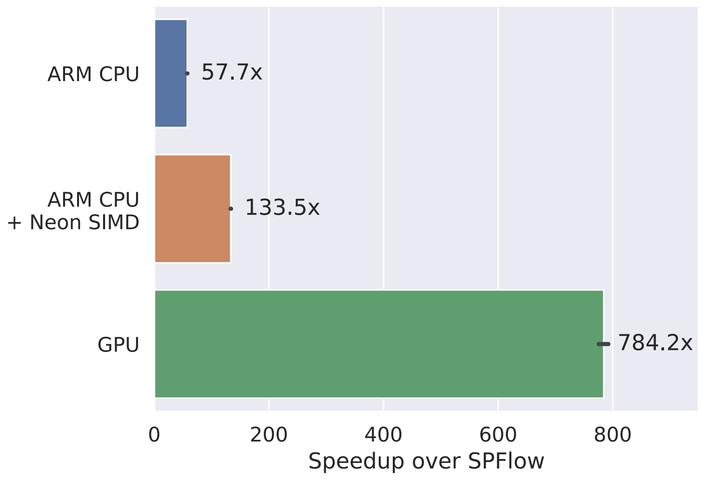


Figure 18.8: Performance comparison for noisy speech samples on embedded systems, given as speedup over execution in SPFlow.

18.6 RELATED WORK

To the best of our knowledge, the compiler presented in this work is the *first* compiler for Sum-Product Networks, enabling efficient inference on multiple hardware platforms.

For creation, training, inference, and experimentation with Sum-Product Networks, a number of libraries have been proposed over the years. The two most popular ones, according to the survey conducted by Paris et al. [5], are SPFlow [3] and libspn [8].

SPFlow allows users to either programmatically create an SPN or learn it, including its structure, from data. It also supports inference on the obtained SPN, either in pure Python, or, for a limited number of cases, through a translation to a Tensorflow graph and execution of that graph. As our evaluation has shown, our compiler significantly outperforms both variants.

Libspn also allows to perform parameter learning and inference for SPNs, again through translation to a Tensorflow graph, which has yielded suboptimal performance in our evaluation in Section 18.5.

Another interesting approach to efficient training and inference for SPNs is through tensorization of the SPN graph, as shown in [6] or [12]. However, these implementations are limited to weight learning, with the structure of the SPNs being subject to additional constraints, whereas our compiler can process SPNs with *arbitrary* DAG structure.

In previous work [9, 10], we have developed a custom, FPGA-based inference accelerator for Sum-Product Networks. However, as the automatically generated accelerator uses a fully spatial hardware layout, the maximum size of SPNs that can be mapped to the FPGA is limited by the available hardware resources to sizes significantly smaller than the SPNs evaluated in this work, and the flow currently does not support Gaussian distributions.

18.7 CONCLUSION

In this work, we have presented SPNC, a domain-specific compiler for fast inference in Sum-Product Networks. The implementation of SPNC is based on the open-source MLIR framework, which facilitates the implementation of domain-specific compilers.

SPNC was designed to seamlessly integrate with SPFlow, a popular open-source library for SPN construction, learning, and representation, through its Python interface.

In our evaluation, using an SPN-based robust automatic speaker identification as a real-world example of Sum-Product Networks, we have demonstrated how SPNC can target different heterogeneous systems and can achieve a speedup over SPFlow of a factor of up to 978x when compiling for CPUs with vector extensions, and up to a factor of 1003x when targeting CUDA GPUs.

AVAILABILITY

SPNC is available as open-source software under the Apache v2 License on Github[5]. In the releases section on Github, pre-built packages for Linux systems can be found for download and installation via Python `pip`.

REFERENCES

[1] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. "Factor graphs and the sum-product algorithm." In: *IEEE Transactions on information theory* 47.2 (2001), pp. 498–519.

[2] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Arnaud Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation." In: *CGO 2021*. 2021.

[3] Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. *SPFlow: An Easy and Extensible Library for Deep Probabilistic Learning using Sum-Product Networks*. 2019. arXiv: 1901.03704 [cs.LG].

[4] Aaron Nicolson and Kuldip K. Paliwal. *Sum-Product Networks for Robust Automatic Speaker Identification*. 2020. arXiv: 1910.11969 [eess.AS].

[5] Iago Paris, Raquel Sanchez-Cauce, and Francisco Javier Diez. *Sum-product networks: A survey*. 2020. arXiv: 2004.01167 [cs.LG].

5 https://github.com/esa-tu-darmstadt/spn-compiler

[6] Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Xiaoting Shao, Martin Trapp, Kristian Kersting, and Zoubin Ghahramani. "Random sum-product networks: A simple but effective approach to probabilistic deep learning." In: *Proceedings of UAI*. 2019.

[7] Hoifung Poon and Pedro Domingos. "Sum-product networks: A new deep architecture." In: *2011 IEEE Intl. Conf. on Computer Vision Workshops (ICCV Workshops)*. 2011.

[8] Andrzej Pronobis, Avinash Ranganath, and Rajesh PN Rao. "LibSPN: A library for learning and inference with Sum-Product Networks and TensorFlow." In: *Principled Approaches to Deep Learning Workshop*. 2017.

[9] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-based Accelerators." In: *IEEE Intl. Conf. on Computer Design (ICCD)*. IEEE. 2018.

[10] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. "Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs." In: *2020 IEEE 28th Annual Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM)*. 2020.

[11] Indar Sugiarto, Cristian Axenie, and Jörg Conradt. "FPGA-based hardware accelerator for an embedded factor graph with configurable optimization." In: *Journal of Circuits, Systems and Computers* 28.02 (2019), p. 1950031.

[12] Jos van de Wolfshaar and Andrzej Pronobis. "Deep Generalized Convolutional Sum-Product Networks for Probabilistic Image Representations." In: *arXiv:1902.06155* (Sept. 2019). (Visited on 01/28/2020).