15th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures

13th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms

PARMA-DITAM 2024, January 18, 2024, Munich, Germany

Edited by

João Bispo
Sotirios Xydis
Serena Curzel
Luís Miguel Sousa

OASICS

Editors

João Bispo
University of Porto, Portugal
jbispo@fe.up.pt

Sotirios Xydis
National Technical University of Athens, Greece

Serena Curzel
Politecnico di Milano, Italy

Luís Miguel Sousa
University of Porto, Portugal
lm.sousa@fe.up.pt

ACM Classification 2012
Computer systems organization → Multicore architectures; Computer systems organization → Reconfigurable computing; Software and its engineering → Compilers

ISBN 978-3-95977-307-2

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-307-2.

Publication date
March, 2024

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.PARMA-DITAM.2024.0

OASIcs – OpenAccess Series in Informatics

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Preface

This volume collects the papers presented at the 15th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures, and the 13th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2024). The workshop is co-located with the 2024 edition of the HiPEAC conference and was held on the 18th of January, 2024, that took place in Munich, Germany.

The current trend towards many-core and the emerging accelerator-based architecture requires a global rethinking of software and hardware design, which turn out to be more than ever before strongly entangled.

The PARMA-DITAM workshop focuses on many-core architectures, parallel programming models, design space exploration, tools and run-time management techniques to exploit the features and boost the performance of such (possibly heterogeneous, (re-)programmable and/or (re-)configurable) many-core processor architectures from embedded to high performance computing platforms and cyber physical systems.

The scope of the PARMA-DITAM workshop include the following topics:

- T1: Parallel programming models, languages, and applications for many-core platforms
- T2: Compiler and virtualization techniques for novel computing architectures
- T3: Run-time modeling, monitoring, adaptivity, power and memory management
- T4: Design of heterogeneous and reconfigurable many-core architectures
- T5: Methodologies, design tools, and high-level synthesis for heterogeneous architectures
- T6: Hardware/software co-design and design space exploration
- T7: Case studies, success stories and applications applying T1–T6

List of Authors

Giovanni Agosta (4)
DEIB – Politecnico di Milano, Italy

Alireza Amirshahi (2)
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Giovanni Ansaloni (2)
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

David Atienza (2)
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Marco Barbone (6)
Imperial College London, UK

Claudio Barone (1)
Pacific Northwest National Laboratory, Richland, WA, USA

Juergen Becker (5)
Karlsruhe Institute of Technology, Germany

Daniele Cattaneo (4)
DEIB – Politecnico di Milano, Italy

Stefano Cherubin (4)
NTNU – Norwegian University of Science and Technology, Trondheim, Norway

Serena Curzel (1)
Politecnico di Milano, Italy

Lev Denisov (4)
DEIB – Politecnico di Milano, Italy

Felix Eberhardt (3)
Operating Systems and Middleware Group, Hasso Plattner Institute, University of Potsdam, Germany

Fabrizio Ferrandi (1)
Politecnico di Milano, Italy

Michele Fiorito (1)
Politecnico di Milano, Italy

Georgi Gaydadjiev (6)
Delft University of Technology, The Netherlands

Michele Gazzetti (3)
IBM Research Europe, Dublin, Ireland

Roberto Giorgi (6)
University of Siena, Italy

Giovanni Gozzi (1)
Politecnico di Milano, Italy

Andreas Grapentin (3)
Operating Systems and Middleware Group, Hasso Plattner Institute, University of Potsdam, Germany

Michael Huebner (5)
BTU Cottbus – Senftenberg, Germany

Wayne Luk (6)
Imperial College London, UK

Gabriele Magnani (4)
DEIB – Politecnico di Milano, Italy

Oliver Oey (5)
Karlsruhe Institute of Technology, Germany; emmtrix Technologies GmbH, Karlsruhe, Germany

Christian Pinto (3)
IBM Research Europe, Dublin, Ireland

Andreas Polze (3)
Operating Systems and Middleware Group, Hasso Plattner Institute, University of Potsdam, Germany

Marco Procaccini (6)
University of Siena, Italy

Amin Sahebi (6)
University of Siena, Italy

Timo Stripf (5)
emmtrix Technologies GmbH, Karlsruhe, Germany

Tobias Zagorni (3)
Operating Systems and Middleware Group, Hasso Plattner Institute, University of Potsdam, Germany

High-Level Synthesis Developments in the Context of European Space Technology Research

Fabrizio Ferrandi
Politecnico di Milano, Italy

Michele Fiorito
Politecnico di Milano, Italy

Claudio Barone
Pacific Northwest National Laboratory, Richland, WA, USA

Giovanni Gozzi
Politecnico di Milano, Italy

Serena Curzel
Politecnico di Milano, Italy

Abstract

European efforts to boost competitiveness in the space services sector promote the research and development of advanced software and hardware solutions. The EU-funded HERMES project contributes to the effort by qualifying radiation-hardened, high-performance programmable microprocessors and developing a software ecosystem that facilitates the deployment of complex applications on such platforms. The main objectives of the project include reaching a technology readiness level of 6 (i.e., validated and demonstrated in relevant environment) for the rad-hard NG-ULTRA FPGA with its ceramic hermetic package CGA 1752, developed within projects of the European Space Agency, French National Centre for Space Studies and the European Union. An equally important share of the project is dedicated to the development and validation of tools that support multicore software programming and FPGA acceleration. The HERMES project selected the Bambu High-Level Synthesis tool to integrate capabilities to translate C/C++ code into Verilog/VHDL in its development ecosystem. In HERMES, Bambu has been and will be extended to support new FPGA targets, architectural models, model-based design, and input applications. The increased performance offered by FPGAs is thus made available also to software developers who do not have hardware design expertise.

2012 ACM Subject Classification Hardware → Methodologies for EDA; Hardware → High-level and register-transfer level synthesis; Computer systems organization → Architectures; Hardware → Very large scale integration design; Hardware → Reconfigurable logic and FPGAs

Keywords and phrases High-Level Synthesis, rad-hard FPGAs

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2024.1

Category Invited Talk

Funding This research has received funding from the EU Horizon 2020 Programme under grant agreement No 101004203.

1 Introduction

In space applications, the computational requirements for onboard computing are rapidly approaching the limits of what space-grade processors and microcontrollers can offer, mainly because radiation-hardened components are inherently slower than general-purpose CPUs. Hybrid platforms based on a mix of CPU and FPGA logic have become increasingly important. European institutions such as ESA, CNES, and the EU funded several efforts to develop a new generation of rad-hard FPGA platforms, including projects such as BRAVE [8], VEGAS [13], OPERA [15], DAHLIA [14], and HERMES [12]. These platforms provide improved performance with acceptable overhead in size, power consumption, and cost. However, designing such systems is a challenging task.

High-level synthesis is a process that automatically generates an optimized hardware implementation from a high-level software description. HLS tools have changed the way we think about FPGA design by automating the most complex and time-consuming aspects of the development process: by eliminating the need for manual coding in VHDL/Verilog, users can now focus on providing a program written in a well-known software language such as C/C++ along with timing and resource utilization constraints that the final design must satisfy. This approach has become increasingly popular in the design of hybrid CPU-FPGA systems. It allows designers to explore different hardware/software partitionings and optimizations quickly and to generate efficient hardware implementations that meet stringent requirements. So, HLS in the design of hybrid CPU-FPGA systems for space applications has become a powerful tool for designers to optimize the performance of their systems while meeting the strict constraints imposed by space missions. As such, it is a crucial enabling technology for developing the next generation of rad-hard FPGAs for space applications.

The High-Level Synthesis flow begins with a compilation step to analyze data dependencies and loops in the input program, optimize the code, and generate a Control and Data Flow Graph (CDFG). The CDFG is then subjected to three core steps - resource allocation, scheduling, and binding - to define the structure of the output hardware. These steps involve assembling functional, storage, and communication units taken from a library of RTL components. Further optimization and analysis passes are carried out in the front-end, middle-end, and back-end of the tool to generate efficient accelerator designs. The result is HDL code ready to be used in a downstream FPGA or ASIC design tool for further analysis, logic synthesis, and deployment. In the past, the shorter development time offered by HLS tools used to come at the cost of reduced efficiency in the generated designs. However, with the availability of several commercial and open-source tools today, that is no longer the case. These tools can generate efficient designs that are competitive in terms of speed and resource utilization with hand-optimized RTL code.

This paper shows how the Bambu HLS tool [3] has been extended in the context of the HERMES project [12] (Qualification of High-pErformance pRogrammable Microprocessor and dEvelopment of Software ecosystem). The integration of the capabilities of Bambu in the space development ecosystem allows space application developers with no hardware design expertise to exploit the performance offered by FPGAs. The paper is divided into three additional sections. The first section presents Bambu, an open-source HLS tool adopted by the HERMES project. The second section discusses our most recent extensions to Bambu, which aim to enhance the tool's usability in an industrial context for space applications. The third section provides an overview of representative applications and corresponding experimental results, followed by the paper's conclusion.

2 The Bambu Open-Source High-level Synthesis tool

Bambu is a command-line tool developed at Politecnico di Milano providing support to designers for the HLS of complex applications. Most C/C++ constructs are supported, including function calls, access to arrays and structs, parameters passed by reference or copy, pointer arithmetic, dynamic resolution of memory accesses, and module sharing. The flow resembles a software compilation process: it begins with a high-level specification and generates low-level code through a series of analysis and optimization steps. Like a standard

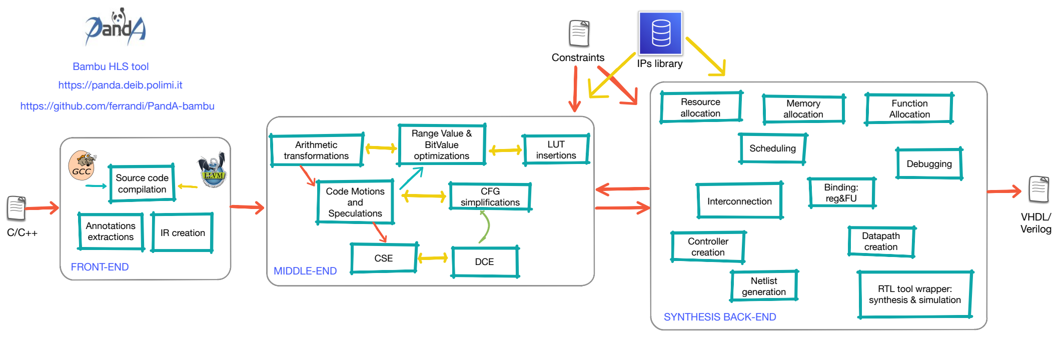


Figure 1 Bambu high-level synthesis flow.

software compilation flow, Bambu has three phases (Figure 1): front-end, middle-end, and back-end. In the front-end, the input code is parsed and translated into an intermediate representation (IR) that is used in the following parts of the flow. In the middle-end, target-independent analyses and optimizations are performed. The back-end performs the actual synthesis of Verilog/VHDL code ready for simulation, logic synthesis, and implementation through external tools.

Bambu front-end. Within Bambu, the user can choose several different front-end compilers, such as GCC and Clang. If GCC is selected, a plugin extracts the call graph and the Control Data Flow Graph of the functions under analysis from GCC's internal IR. Similarly, a Clang plugin extracts the same information from the input and serializes it into a textual format that is easy to parse. Bambu then parses all the compiler serialized information plus all the annotations to build a Static Single Assignment in-memory IR. This approach decouples the compiler front-end code from the rest of the HLS process. Localizing all the changes in a GCC or LLVM/Clang plugin allows rapid and easy integration of many different versions of the compilers. Bambu supports GCC versions from 4.9 to 8, and LLVM/CLANG versions from 4.0 to 16.

Bambu middle-end. Starting from the GCC/Clang IR, Bambu rebuilds data structures, such as the Call Graph and the Control Data Flow Graphs, and builds additional data structures, such as the Program Dependence Graphs. Next, it applies a set of device-independent analyses and transformations. Some of these steps are commonly used in a software compilation flow (e.g., data flow analysis, loop recognition, dead code elimination, constant propagation, LUT expression insertion, etc.). Multiplications and divisions by constant values are transformed into expressions that use only shifts and adders to reduce area utilization and improve timing. The resulting expression structure depends on the target device and technology since adders and multipliers may have different performances on different devices. Differently from general-purpose software compilers, designed to target a processor with a fixed-sized data-path (usually 32 or 64 bits), a HLS compiler can exploit custom-sized operators (e.g., a multiplier with the minimum number of I/O bits) and registers. Consequently, we can select the minimal number of bits required for the specific algorithm's operations and value storage, which leads to less area, less power, and shorter critical paths. At this stage, Bambu also performs Bitwidth and Range Analysis, aiming at reducing the number of bits required by data-path operators. This analysis is crucial during the optimization process because it impacts all non-functional requirements (e.g., performance, area, power) of a design without affecting its behavior.

Bambu synthesis back-end. In this phase, Bambu performs the actual architectural synthesis of the specification. The synthesis process works on each function separately, and the resulting architecture reflects the structure of the call graph. A single function is implemented through at least two sub-modules: the control logic and the data-path. Control logic modeled as a Finite State Machine handles the routing of the data values and the temporal execution of the operations. The data-path is a custom mux-based architecture with optimized data types to reduce the number of flip-flops and bit-level multiplexers, implementing all the operations and memories required during the function execution. The following paragraphs describe the sequence of steps that Bambu follows to generate control and data-path modules.

Function Allocation. Function Allocation associates the high-level functions with specific resources available in the technology library associated with the target device. The technology library coming with Bambu integrates standard functions described in Verilog or VHDL, standard system libraries such as `libc` and `libm`, and designer-defined components written in Verilog or VHDL. Bambu supports function pointers and sharing of (sub)modules across module boundaries [7]. Sharing of functions is achieved using function proxies that act as intermediaries between function calls in the original specification and shared modules. This method of sharing results in significant area savings when dealing with complex call graphs, without any notable impact on execution delays.

Memory Allocation. Memory Allocation refers to the storage of aggregate variables such as arrays and structures, global variables, and the implementation of dynamic memory allocation. Bambu adopts an architecture for memory accesses that supports a wide range of cases. Statically analyzing the memory accesses, Bambu builds a hierarchical data-path where memories can be classified as read-only, local, with aligned or unaligned memory accesses, or those requiring dynamic resolutions. The memory interconnection defines multiple buses connecting load/store components to their respective memories. Dual-port BRAMs or memory controllers with complex parallel channels are supported by replicating such memory interconnections as needed. The same memory infrastructure can connect to external components (e.g., scratchpads, caches, and DRAMs) or directly to the bus to access off-chip memory. Supporting protocol-based accesses (e.g., FIFO or stream-based access) is obtained by generating specific components that replace load/store units.

Resource Allocation. Resource allocation associates operations not mapped on a function to resource units (RU) available in the resource library. During the middle-end phase, the specification is inspected to identify the characteristics of the operations: these include the type of the operation (e.g., addition, multiplication, etc.) and the types of the operands (e.g., integer, float, etc.). Floating-point operations are supported through the HLS of a soft-float library containing basic soft-float operators [4] or alternatively by exploiting the FloPoCo software [2], a generator of arithmetic Floating-Point Cores. The allocation step maps operations on the set of available RUs; their characterization includes latency, area, and the number of pipeline stages. Usually, more operation/RU matchings are feasible; in this case, selecting a proper RU is driven by design constraints. The library of RUs used by Bambu is quite rich and may include several implementations for the same operation. Furthermore, the library includes RUs that are provided as templates in a standard hardware description language, such as Verilog or VHDL. These templates can be customized and retargeted in accordance with the characteristics of the target technology. In this case, the underlying logic synthesis tool will determine the best architecture to implement each operation (for example, multipliers can be mapped on dedicated DSP blocks or implemented with LUTs). To perform aggressive optimizations, each library component is annotated with helpful information during the entire HLS process, such as resource occupation and latency. Bambu adopts a pre-characterization approach through a tool called Eucalyptus to synthesize different configurations of library components and collect their resulting latency and resource consumption metrics as XML files in the Bambu library. The configurations are obtained by specializing a generic template of the resource component, such as a multiplier or an adder, according to the bit widths of its input and output arguments and the number of pipeline stages.

Scheduling. By default, Bambu employs a list-based scheduling algorithm. In its basic formulation, list-based scheduling associates each operation with a priority according to particular metrics. Scheduling proceeds iteratively, associating a set of operations to be executed with each control step. Ready operations (i.e., whose dependencies have been satisfied in previous iterations of the algorithm) can be scheduled in the current control step, considering the availability of the resources. If multiple ready operations compete for a resource, then the one having a higher priority is scheduled. In addition to this old but efficient algorithm, Bambu also features a more aggressive scheduling algorithm, the speculative scheduling algorithm based on System of Difference Constraints [6]. This algorithm builds an integer linear programming formulation of the scheduling problem, allowing code motion and speculation of operations that belong to different basic blocks.

Module Binding. Within the computed schedule, operations that execute concurrently are not allowed to share the same resource instance. In Bambu, binding of operations to resources is performed through a clique covering algorithm on a weighted compatibility graph [11]. The compatibility graph is built by analyzing the schedule: operations scheduled on different control steps are marked as compatible. Weights express how valuable it is for two operations to share the same hardware resource. They are computed considering area/delay trade-offs caused by sharing; for example, RUs that occupy a large area will be more likely shared. Weights computation also considers the cost of interconnections required by the steering logic. Bambu also offers several other algorithms for solving the covering problem on compatibility/conflict graphs.

Register Binding. Register binding associates storage values to registers and requires a preliminary analysis step, the liveness analysis [11]. Liveness analysis starts from the schedule to identify each variable's life intervals, i.e., the sequence of control steps in which a temporary value needs to be stored. Variables with non-overlapping life intervals may share the same register.

Interconnection Binding. Interconnections are bound according to the outcome of the previous steps: if a functional or memory resource is shared, then the algorithm introduces steering logic on its inputs. It also identifies the set of control signals that will be driven by the controller.

Netlist Generation. The final architecture is then generated and represented through a hyper-graph, highlighting the interconnection between modules. The netlist generation step translates such representation in a register transfer-level (RTL) description in Verilog or VHDL. The process accesses the resource library, which embeds the RTL implementation of each resource. This process is target-dependent, and the hardware descriptions may differ for different technologies (e.g., ASIC or FPGA) or target devices.

Generation of Synthesis and Simulation Scripts. Bambu automatically generates synthesis and simulation scripts that can be customized via XML configuration files. The RTL-synthesis tools currently supported are AMD/Xilinx ISE, AMD/Xilinx Vivado, Yosis-Vivado, Intel/Altera Quartus, Lattice Diamond, NanoXplore, and OpenRoad. Supported simulators are Mentor Modelsim, Xilinx XSIM, and Verilator.

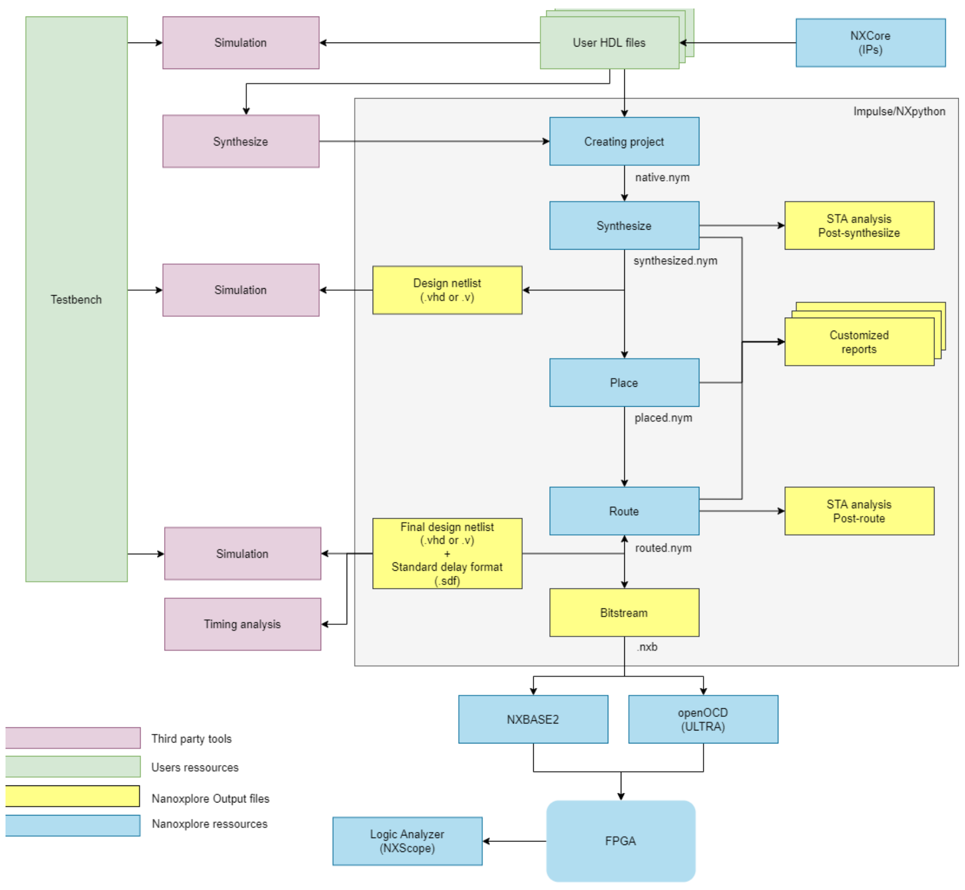


Figure 2 NanoXplore Impulse design flow.

3 New HLS Features

3.1 NanoXplore Logic Synthesis Integration

The first step in the integration of Bambu into the HERMES design flow for space applications was to add support for the NanoXplore synthesis tool Impulse (Figure 2). The Impulse design suite translates HDL code into a device-specific bitstream for NanoXplore radiation-hardened FPGAs through logic synthesis, place-and-route compilation steps, and static timing analysis tools for performance estimation. Seamless integration of Bambu and Impulse is achieved by automatically generating backend synthesis scripts after the generation of the RTL code.

During the HLS process, Bambu applies optimizations that are specific to a target, and therefore, its backend has been customized to support three new types of space-grade FPGAs: NG-MEDIUM, NG-LARGE, and NG-ULTRA. Before integrating the logic synthesis backend based on Impulse and running characterization through Eucalyptus, it was necessary to map Bambu library components correctly to the actual DSPs and True Dual Port RAMs available on the NG-ULTRA fabric. Since the mapping occurs through behavioral HDL templates, the components used by Bambu for arithmetic operations and storage modules have been customized to comply with the Impulse synthesis guidelines.

3.2 AXI Protocol Interfaces

The NG-ULTRA board's ARM processor uses the AXI4 protocol interfaces to communicate with the rest of the system. AXI4 is a standard that includes AXI4, AXI4-stream, and AXI4-Lite protocols. These are used to access memory banks, streaming channels, and memory-mapped registers.

Bambu was therefore extended to offer the possibility to access data outside of the accelerator over an AXI4 bus, which is the standard also for other on-chip communications (e.g., to communicate with HBM on AMD/Xilinx FPGAs). This can be used to connect modules created with Bambu with modules created by other sources, or with external memories. AXI4 is a master/slave protocol that defines a series of channels, i.e., independent groups of signals exchanged between the master and slave devices. As shown in figure 3, AXI4 defines five channels: Address read, Read data, Address write, Write data and Write response, and collects them into bundles associated with different parameters. Each channel contains a set of information signals and two handshake signals, xVALID and xREADY, where x depends on the actual channel. These signals indicate the availability of one of the endpoints to exchange the channel information. In particular, the xVALID signal indicates that the source of the information on the bus has provided valid data, and the xREADY signal indicates that the recipient is ready to accept the information. The handshake can only be completed once both signals are active simultaneously. The address read/write channels contain the signals needed to define the transaction, such as the data address, the burst type that should be used, the length of the burst, and more. The read channel contains the data requested by the transaction and a read response, indicating whether the request was successful. The write channel contains the data that must be passed to the slave device and a write strobe signal, specifying which data bytes are valid. The write response channel contains a signal indicating whether the write transaction was successful. AXI supports unaligned operations: it can exchange data even if the requested value has a size different from the bus or its address is not a multiple of the bus size. The AXI protocol is burst-based, which means the master must only specify the initial address and the burst information. The slave will then compute the correct addresses for each subsequent data transfer, or beat, autonomously based on the information passed when the burst was defined. Using pragmas, Bambu can add an AXI4 controller module inside the hardware design linked to a specific memory parameter or a set. Each master module is responsible for the communications on a bundle of AXI4 channels and can act independently from the others. When a memory operation is issued, the finite state machine activates only the AXI master module related to the requested memory parameter. This allows the execution of parallel memory accesses when there are multiple AXI bundles.

Bambu can also create a testbench that includes the AXI4 slave counterparts of the master interfaces. This enables users to simulate data exchange and verify its correctness. Users can configure memory delay estimates to assess the application's performance, taking into account an estimated latency for data transfers.

3.3 AXI Caches

When requesting an AXI interface, Bambu offers the possibility to add a customizable cache that can intercept or forward memory access requests coming from the memory controller to the AXI slave. These caches can help reduce the average memory access latency by accessing the data present in the cache rather than performing the full transaction over the AXI bus. Caches are requested by the user through pragma annotations, and several different options can be specified to generate caches with e.g., different write policies, replacement policies, and cache line sizes.

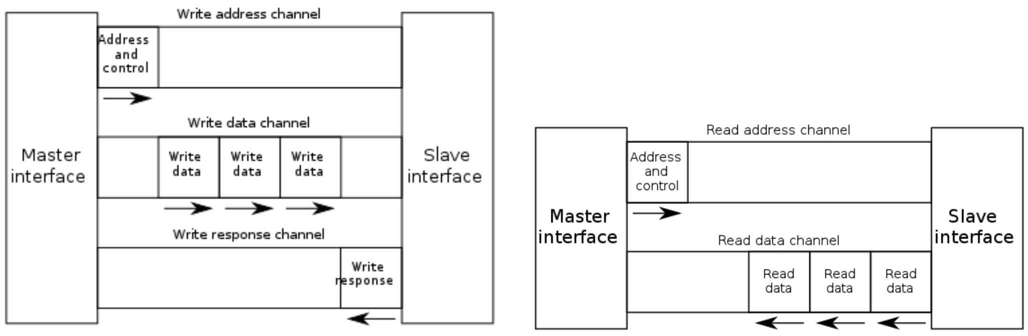


Figure 3 AXI4 channels descriptions: on the left the Write address, Write data and Write response channels, while on the right the Read address and Read data channels.

The caches provided by Bambu are largely based on the work done in [10], with some modifications that allowed it to be integrated into our tool and to improve its performance and customization. The basic element of the Bambu cache is a single datum of the same size as the data type of the kernel argument being stored. These are grouped in cache lines, i.e., a sequence of elements that are contiguous in memory. When there is a cache miss, the cache reads an entire line from memory, so whenever a datum is requested all the content of the line is immediately available for future requests. Cache lines can also be grouped in ways. Unlike elements in a line, lines in a way are not necessarily contiguous. Inside each way, a cache line can only be stored in a single position, depending on the starting address of the line. When more than one way is present, each of them provides a position to hold the line. Multiple memory areas are mapped to the same cache lines, so while populating the cache it is common that new data must be placed in an already populated line. In this case, the line is simply overwritten and requests regarding the old data will need to go through the main memory again.

While caches typically have the same behavior with read operations, different policies are available when performing writes. Bambu caches offer two different policies: write through no allocate and write back allocate. When using the first policy, the cache always immediately forwards the write operation to the main memory. If the address that was written is already present in the cache it is updated in the internal memory too, otherwise, no other action is performed. When the write back policy is selected, the data is not transferred to the main memory: only the internal state of the cache is updated. In case the element that was written was not already present in the cache the line in which it is contained is first read from memory, then the data is updated. The write policy also impacts what happens when a cache line needs to be replaced: in the case of write through policy, the data in the main memory and the cache are always consistent, so no action is needed when replacing lines. However, write back caches need to keep track of modified lines using a dirty bit that is set whenever any element of the line is modified. When a line must be replaced, if the dirty bit is set, the entire line is first written to the main memory, then it can be replaced. Otherwise, no action is needed, and the line can immediately be overwritten. Finally, another difference is that at the end of the computation write back caches need to write all their dirty lines back to main memory, while write though caches do not need this operation. In general, write through policies perform better when the time between two write operations is greater than the latency of the memory because by the time the second write is received, the cache is already done with the previous operation and its state is consistent with the main memory.

On the other hand, write back caches are more useful when performing a lot of writes to the same cache lines in a small period, because the whole line is then transferred in a single memory transaction either when it is replaced or at the end of the computation.

One of the modules used internally by the caches is the write buffer. This is a data structure that holds write transactions to main memory that must be performed but have not yet been completed. It is similar to a circular buffer and is handled by three indexes: a write index, indicating the next slot where data can be written, a read index, indicating the next write transaction that must be initiated, and a backup index, indicating the first transaction that has been started but has not yet completed. The write index is increased whenever a new item is inserted in the buffer, i.e., when a cache line should be written to memory. As long as the buffer is not empty and the AXI bus is available, the controller handling the bus will pick up the element marked by the read index and increase it, while beginning the write transaction on the bus. The controller also monitors the write responses of the AXI slave device and forwards them to the queue. The response indicates whether the transaction was successful or not. In the first case, the write operation has been completed and is no longer needed in the buffer, so the backup index in the write buffer is increased. In case the write response indicated an error, the transaction must be repeated. This is done by reading the element in the buffer marked by the backup index and writing it at the next available position in the buffer. Both the backup index and the write index are then increased. By using this buffer, it is possible to perform pipelined write transactions to the main memory, exploiting the AXI bus more efficiently. While waiting for the response of a transaction (if the buffer is not full), new write operations can be performed without stalling the entire cache. Increasing the size of the buffer is especially useful for write through caches since they typically perform a higher number of smaller data transfers with respect to write back caches, which perform a smaller number of larger transfers. While it can be useful even in this case, it should be noted that while write through caches transfer data one element at a time, write back caches transfer entire lines, so the buffer will use more resources when using the same size parameter.

Caches can be classified as direct-mapped, n-way set-associative, and fully associative. All three types of caches are made available by Bambu by selecting appropriate values for the n_ways and way_size parameters in the pragmas that instruct the tool to generate a cache. Greater associativity can improve the effectiveness of the cache, as it provides more options for storing cache lines. However, associativity has a great cost in terms of resource usage, so it should only be used as needed.

Associative caches can have multiple positions in which a cache line can be located. For this reason, a policy must be enacted that decides which way will store data whenever a new line is read from memory. Bambu offers two different replacement policies: least recently used (LRU) and a tree-structure-based pseudo-LRU.

4 Applications

In the context of the HERMES project, Bambu has been used to automate the design of space-related applications. In particular, the project use cases cover image and vision processing algorithms, software-defined algorithms, and artificial intelligence applications. In this section, as an example of the results achievable with Bambu, we provide some simple benchmarks showcasing the features described in this paper. All the benchmarks have been synthesized using Bambu to create the hardware description and NXmap to map it on the FPGA.

Table 1 Comparison between different Boards for the Sparse Matrix-Vector Multiplication benchmark.

Target	Latency(µs)	Cycles	Frequency	LUTs	Registers	DSPs
nx1h35S	2109	75586	35.85 MHz	1453	1669	10
nx1h140tsp	2083	85960	41.28 MHz	1508	1603	10
nx2h540tsc	1190	81020	68.11 MHz	1664	1642	10

Table 2 Effect of caches on Matrix-Matrix Multiplication benchmark.

Target	Latency(µs)	Cycles	Frequency	MEM	LUTs	Registers	DSPs
nx2h540tsc	5945	302160	50.82 MHz	0	2719	3312	6
nx2h540tsc Cache	3415	129706	37.98 MHz	112	5806	3832	6

4.1 Sparse Matrix-Vector Multiplication

The first benchmark is an implementation of a double precision sparse matrix-vector multiplication (SPMV) from the MachSuite benchmark suite [9]. The results are reported in Table 1 and focus on comparing three boards from Nanoexplore: the first one is the NG-MEDIUM (nx1h35S), the second one the NG-LARGE (nx1h140tsp), and the last one the NG-ULTRA (nx2h540tsc). The considered application implements a sparse matrix-vector multiplication using fixed-size neighbour lists: the matrix has 494 rows and columns, but only 10 elements are assumed to differ from zero for each row vector multiplication.

As shown in the table, there is almost no difference in the number of resources used on the different boards: the number of LUTs, Registers and DSPs is approximately the same on all the proposed designs. This is also true for the number of cycles, as there is only a tiny difference between the fastest and slowest board. However, using larger boards has an advantage: the frequency that NXmap can achieve on nx2h540tsc is two times faster than the one achieved on nx1h35S.

4.2 Matrix-Matrix Multiplication

The second benchmark is an implementation of a single precision dense matrix-matrix multiplication with tiling to increase the locality of the requested data. The results are reported in Table 2 and focus on comparing the effect of the introduction of caches around a kernel synthesized from a regular application. We assumed a memory delay of 20 cycles for both read and write operations to simulate the delay of an external memory. The matrices used in the computations have 32 rows and columns for a total of 1024 elements. In this benchmark, we used AXI4 to exchange data between the external memory and the accelerator; we used Bambu to create three different AXI4 bundles, one for each input matrix and one for the output, to parallelize the memory operations. The difference between the two configurations is that in the one with the caches, we added a 16 elements line cache for each bundle, which allows burst operations and reduces the latency of the external memory.

This benchmark shows the effect of the caches on the performance of kernels synthesized by Bambu. In this application, which is highly regular and with data locality, the configuration with caches can achieve a 2.5 times speed-up while only using 2.1 times LUTs, which is an efficient trade-off. Another critical factor when evaluating the performances of the caches is that the number of physical resources used does not depend on the specific design but is constant. This means that the cost of the caches proportionally decreases with larger and more complex designs as the area used by the kernel increases.

Table 3 High-level synthesis of the quantized digit classifier.

Target	Latency(µs)	Cycles	Frequency	MEM	LUTs	Registers	DSPs
NG-ULTRA Embedded	3712	169649	45.7 MHz	34	4627	5714	54

4.3 Quantized Digit Classifier Synthesis

In the third example, we consider a simple MNIST model for digit classification taken from one of the TensorFlow tutorials [16], demonstrating how to convert a TensorFlow model from 32-bit floating-point to the nearest 8-bit fixed-point model using post-training integer quantization. The aim of this process is to encode the model weights with fewer bits, thereby increasing the inference speed. This is particularly useful for low-power devices such as microcontrollers or edge devices, and the same is true for FPGA-equipped space missions.

The process followed to synthesize the quantized model into an FPGA accelerator is the MLIR-based SODA methodology, which is described in detail in [1]. The Multi-Level Intermediate Representation (MLIR) [5] is a flexible and reusable infrastructure available within the LLVM project for building domain-specific compilers. MLIR allows the creation of specialized intermediate representations (IRs), known as *dialects*, which can implement analysis and transformation passes at different levels of abstraction. It can interface with multiple software programming frameworks, including the ones used for implementing deep learning algorithms.

The MNIST model is first described in Python, trained, and then quantized using the tutorial directives. Instead of running the TensorFlowLite model using the standard TensorFlowLite runtime, we output the MLIR description with the quantized weights and activations. Then, we follow a slightly modified SODA flow that translates the MLIR representation into a low-level IR that is understood by Clang/LLVM and, consequently, by Bambu, which is used to generate a corresponding hardware accelerator. Performance and area consumption metrics are shown in Table 3.

5 Conclusions

The HERMES project has extended an open-source High-Level Synthesis tool to suit the specific needs of space applications, as described in this paper. FPGAs are highly versatile and can be used in many different applications, including the space market, where HLS can help reduce the burden of developing hardware accelerators for radiation-hardened FPGAs by raising the level of abstraction required. The HERMES project has many use cases, including machine learning models, and the paper demonstrates how dedicated design methodologies can be integrated into the hardware acceleration process to improve the quality of results of the resulting FPGA design. Future developments will focus on more optimization techniques and architectural templates that can address the specific needs and requirements of aerospace applications, including both general computational demand and artificial intelligence applications.

References

1 Nicolas Bohm Agostini, Serena Curzel, Ankur Limaye, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph B. Manzano, Antonino Tumeo, and Fabrizio Ferrandi. The SODA approach: leveraging high-level synthesis for hardware/software co-design and hardware specialization: invited. In Rob Oshana, editor, *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 – 14, 2022*, pages 1359–1362. ACM, 2022.

2 Florent de Dinechin et al. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

3 F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo. Bambu: an open-source research framework for the high-level synthesis of complex applications. In *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330, 2021.

4 Michele Fiorito, Serena Curzel, and Fabrizio Ferrandi. Truefloat: A templatized arithmetic library for hls floating-point operators. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 486–493, 2023.

5 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

6 Marco Lattuada and Fabrizio Ferrandi. Code transformations based on speculative SDC scheduling. In *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 71–77, November 2015.

7 M. Minutoli et al. Inter-procedural resource sharing in high level synthesis through function proxies. In *International Conference on Field Programmable Logic and Applications, FPL*, pages 1–8, September 2015.

8 STMicroelectronics (FR) NanoXplore (FR). High Density European Rad-Hard SRAM-Based FPGA – First Validated Prototypes – BRAVE, 2017. URL: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Shaping_the_Future/High_Density_European_Rad-Hard_SRAM-Based_FPGA-_First_Validated_Prototypes_BRAVE.

9 Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 110–119. IEEE, 2014.

10 João V. Roque, João D. Lopes, Mário P. Véstias, and José T. de Sousa. Iob-cache: A high-performance configurable open-source cache. *Algorithms*, 14(8), 2021.

11 Leon Stok. Data path synthesis. *Integration*, 18(1):1–71, 1994.

12 supported by H2020 under grant agreement n. 101004203. Qualification of High-pErformance pRogrammable Microprocessor and dEvelopment of Software ecosystem (HERMES), 2021. URL: https://dahlia-h2020.eu/.

13 supported by H2020 under grant agreement n. 687220. Validation of European high capacity rad-hard FPGA and software tools (VEGAS), 2016. URL: http://vegas.us.es/.

14 supported by H2020 under grant agreement n. 730011. Space Qualification and Validation of High Performance European Rad-Hard FPGA (OPERA), 2017. URL: https://dahlia-h2020.eu/.

15 supported by H2020 under grant agreement n. 821969. Space Qualification and Validation of High Performance European Rad-Hard FPGA (OPERA), 2019. URL: https://operahorizon2020.eu/.

16 TensorFlow tutorial. Post-training integer quantization, 2023. URL: https://www.tensorflow.org/lite/performance/post_training_integer_quant.

Accelerator-Driven Data Arrangement to Minimize Transformers Run-Time on Multi-Core Architectures

Alireza Amirshahi
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Giovanni Ansaloni
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

David Atienza
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

The increasing complexity of transformer models in artificial intelligence expands their computational costs, memory usage, and energy consumption. Hardware acceleration tackles the ensuing challenges by designing processors and accelerators tailored for transformer models, supporting their computation hotspots with high efficiency. However, memory bandwidth can hinder improvements in hardware accelerators. Against this backdrop, in this paper we propose a novel memory arrangement strategy, governed by the hardware accelerator's kernel size, which effectively minimizes off-chip data access. This arrangement is particularly beneficial for end-to-end transformer model inference, where most of the computation is based on general matrix multiplication (GEMM) operations. Additionally, we address the overhead of non-GEMM operations in transformer models within the scope of this memory data arrangement. Our study explores the implementation and effectiveness of the proposed accelerator-driven data arrangement approach in both single- and multi-core systems. Our evaluation demonstrates that our approach can achieve up to a 2.7x speed increase when executing inferences employing state-of-the-art transformers.

2012 ACM Subject Classification Hardware → Hardware-software codesign; Computing methodologies → Neural networks

Keywords and phrases Memory arrangement, Data layout, Hardware accelerators, Transformer models, Multi-core, System simulation

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2024.2

Supplementary Material *Software (Source Code)*: `https://github.com/gem5-X/TiC-SAT`
archived at `swh:1:dir:e2fdf5237af31d31fd7b775cc3c5936419145bf9`

Funding This research was partially supported by EC H2020 FVLLMONTI project (GA No. 101016776) and by the ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR.

1 Introduction

In recent years, the rise of transformer models has significantly impacted the field of artificial intelligence, being used in various domains such as natural language processing, machine translation, speech recognition, and computer vision [2, 3, 7]. As a result, the demand for ever more powerful transformers has emerged, leading to the development of increasingly large and complex architectures.

The substantial size and complexity of transformers have raised new challenges in the ability to cope with their computational costs, memory usage, and energy requirements. In fact, the increasing complexity of these models must be carefully managed to contain the ensuing processing needs, which can lead to unacceptable operating costs [22].

Several works have addressed these challenges [10]. The first notable avenue is that of hardware acceleration, that is, the design of processors and accelerators specifically tailored for transformer models [15, 21, 12]. Such specialized hardware aims to significantly reduce computation time by distributing the calculations across multiple smaller processing units within the accelerators.

The second strategy to optimize the execution of the transformer model is the enhancement of the locality of the data. This approach involves maximizing the reuse of data already fetched from slow and energy-hungry off-chip memory. Given the bandwidth limitations of accessing off-chip memory, reducing reliance on this memory and improving data locality can significantly enhance performance in terms of energy and time efficiency. Methods to improve data locality include employing cache hierarchies as on-chip memory, using loop tiling to divide computational workloads into smaller segments [11], and adopting in-memory or near-memory computation techniques [9, 4].

Although reusing data fetched from memory can considerably improve transformer inference efficiency [1], the sequence in which the data are retrieved remains crucial, especially when using a hardware accelerator. This issue forms the basis of our research focus. In this context, all multi-dimensional data structures, e.g., matrices, must be converted into one-dimensional arrays for storage, a process we term *data arrangement.* The arrangement of data is essential in systems with hardware accelerators and memory hierarchies. In this paper, we propose for such systems a memory-data arrangement that aligns with the processing unit size of hardware accelerators. In this data arrangement, when a data block is loaded into the accelerator, a contiguous block can simultaneously be pre-fetched from off-chip memory because the data arrangement is synchronized with the processing sequence. This approach leads to faster data retrieval for subsequent accelerator loads by handling as many contiguous memory accesses as possible.

Key contributions of this paper include the following:

- The introduction of a memory data arrangement that coordinates with the size of the hardware accelerator, facilitating synchronized data access and storage.
- A comprehensive analysis of implementing this memory data arrangement in an end-to-end transformer model, highlighting its advantages in general matrix multiplication (GEMM) operations with minimal added overhead for non-GEMM processes.
- The demonstration of the feasibility of deploying this memory data arrangement in both single- and multi-core hardware accelerators.
- Extensive full-system evaluations, revealing that our proposed memory data arrangement can greatly speed-up inference in transformer models, e.g. by up to 2.7x in a single-core system.

The remainder of this paper is structured as follows. Section 2 explains the background and related work, providing key information on transformers, hardware accelerators, and existing studies on memory data arrangements. Section 3 describes our methodology, introducing the accelerator-driven data arrangement and its impact on transformer models. In Section 4, we present our experimental setup and results, showcasing the effectiveness of the proposed memory data arrangement in enhancing the performance of the transformer model. Finally, Section 5 concludes the paper, summarizing our findings and contributions.

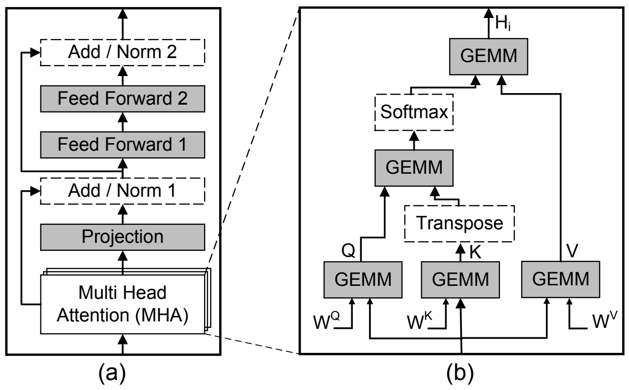


Figure 1 (a) Overall structure of a transformer encoder layer, and (b) detailed representation of a single attention head with its components. The GEMM-based components within the encoder layer and attention head are shaded in grey. Components corresponding to non-GEMM operations have dashed frames.

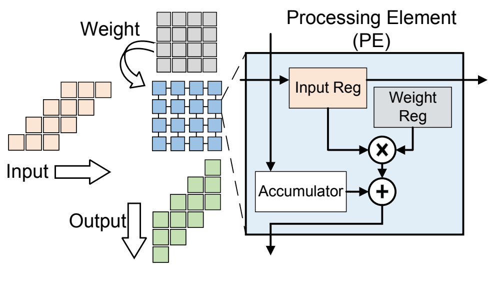
2 Background and Related Works

2.1 Transformers

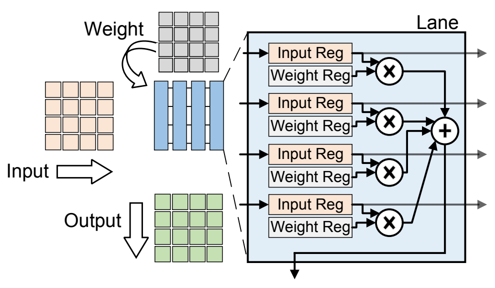
Transformer models consist of multiple interconnected components that generate outputs based on weighted inputs according to a self-attention technique. The self-attention mechanism enables the model to dynamically adjust the weights of different inputs, thus allowing it to capture dependencies between features (e.g., words in case of machine translation), even when distant within the input data. The original version of the transformer model [17] presents an encoder-decoder structure that has proven effective for tasks that require sequence-to-sequence transformations. Several derivatives have been developed, primarily utilizing the encoder portion of the model. The encoder itself has the ability to learn robust high-level representations from input data, which makes these models particularly effective in a wide range of tasks, such as sentiment analysis, question answering, image recognition, and more [2, 3]. In encoder-focused models, the encoder part maintains a structure similar to [17], while the decoder part is often simplified to a single- or two-layer linear component. Consequently, in these applications, optimizations focus on the encoder layers, which comprise the majority of the computational workload. We follow the same approach, aiming to minimize the run-time for encoder-focus transformer models.

The architecture of an encoder layer within a transformer model is depicted in Figure 1a. The Multi-head Attention component takes as input a matrix X with a number of rows equivalent to the sequence length and a number of columns corresponding to the model dimension. This input matrix is applied to h distinct heads, each having a unique set of weight matrices, which generates unique output values from the same input. In more detail, the input matrix X is multiplied by W_i^Q, W_i^K, and W_i^V, where $i \in 0, .., h-1$ represents the head index. The resultant matrices are Q_i, K_i, and V_i, which correspond to Query, Key, and Value matrices, respectively. These multiplications are illustrated in Figure 1b as *GEMM* operations.

Subsequently, the next component involves transposing K_i and multiplying it by Q_i. Following this, a non-linear Softmax operation is applied to the results, and the output is multiplied by V^i. Overall, the output of this single-head attention component is computed as follows:



(a)



(b)

Figure 2 GEMM acceleration using (a) Systolic Arrays and (b) SIMD architectures.

$$H_i = \text{Softmax}(\frac{Q_i \times K_i^T}{\sqrt{d_q}}) \times V_i \quad i \in \{0, 1, \dots, h-1\}. \tag{1}$$

In the above equation, d_q denotes the dimension of the Query, Key, and Value matrices. The softmax operation subsequently scales the matrix values within the range $[0, 1]$. H_i represents the output of the i-th head and, along with the outputs of all other attention heads, serves as input for the projection component.

As illustrated in Figure 1a, the outputs from the heads are concatenated and then subjected to a projection. The projection result is then routed to the Add/Norm component, which performs a non-linear operation. Following this, the output is passed through two feed-forward components. The final stage in the process involves a second Add/Norm operation.

In summary, the core of a transformer encoder layer is predominantly composed of General Matrix Multiply (GEMM) operations. Thus, enhancing the efficiency of GEMM operations is key to accelerating transformer run-time. The proposed method for GEMM acceleration using the data arrangement and the overhead associated with non-linear operations are discussed in detail in Section 3.

2.2 Hardware Accelerators

2.2.1 Accelerators for GEMM Operations

This paper focuses on two predominant categories of accelerators used in GEMM operations: Systolic Arrays (SA) and Single Instruction Multiple Data (SIMD) architectures. Both types are characterized by their integration of data storage registers and processing elements for computation. SA accelerators are widely used in systems such as Tensor Processing Units (TPUs) [8]. Typically, SIMD and TPU systems have competitive performance compared with traditional Graphics Processing Units (GPUs) while enabling tighter integration, requiring less resources [18, 19].

Systolic Arrays consist of a network of Processing Elements (PEs) arranged to process input streams into output streams. Each PE is equipped with arithmetic and storage units, namely an adder, a multiplier, and three registers. Figure 2a shows a 4x4 SA architecture. In this configuration, input and output data propagate through the array in orthogonal directions (for instance, inputs moving left-to-right and outputs top-to-bottom). Weights are preloaded into the PEs before computation begins. During operation, the inputs shift unaltered from one PE to the next with each clock cycle, while the outputs accumulate results from the multiplications of the inputs and weights.

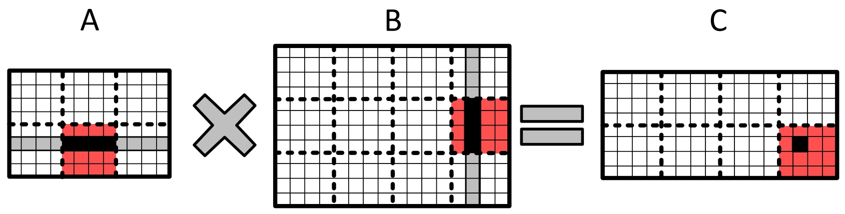


Figure 3 Tiled matrix multiplication $A \times B = C$, considering tiles of size 4x4 elements.

SIMD architectures, as shown in Figure 2b, comprise multiple computing lanes that simultaneously execute the same operation on different data. Each lane performs a dot-product operation. To execute dot-product operations in parallel, the SIMD system loads weights into each lane's registers and processes the input matrix through these lanes to compute the output.

In both SA and SIMD architectures, the configuration of processing elements or lane arrays can be referred to as the accelerator kernel. The term *kernel size* denotes the number of PEs in each row or lane. Our research aims to optimize the memory data arrangement aligning with this kernel size to enhance the efficiency of GEMM operations in these accelerators for a transformer application.

2.2.2 Tiling for GEMM Operations

The size of the matrices involved in transformer models greatly exceeds the accelerator kernel size. Hence, to use the accelerators when computing large GEMMs, matrices must be tiled; that is, partitioned into smaller submatrices. Partial GEMMs are then performed on tiles, increasing locality and data reuse in accelerator kernels. Finally, the results are aggregated with element-wise additions.

Figure 3 illustrates this concept with matrices A and B undergoing multiplication to produce matrix C. Within each of the three matrices, a 4x4 tile is highlighted. Multiplying the highlighted tiles into matrices A and B gives a partial result in the corresponding area of matrix C. For example, the black-marked value in matrix C is partially derived by multiplying the black row in A by the corresponding black column in B. Note that the tiling yields only partial values; however, by sliding the tiles and accumulating these partial results through element-wise addition, the final values in the output matrix C can be derived.

2.3 Related Works

2.3.1 Memory Data Arrangements

Rearranging the layout of the matrix elements can greatly speed up GEMMs, as shown in [6]. This research focused on the execution of a singular General Matrix Multiply (GEMM) operation, where the matrices were arranged in a square block format where the matrices are stored as square submatrices column by column. Building on this concept, the authors in [5] applied the block matrix layout to operations beyond GEMM, such as single convolution and Gaussian blur. Their research demonstrated that with the implementation of an optimized memory layout, they were able to fully utilize the memory capacity.

Our proposed method diverges from the approaches in [6] and [5] by employing block-wise memory arrangement in the context of hardware accelerators. We strategically align this kernel size with the blocks used for data storage in memory. Furthermore, we extend the application of this memory arrangement to a complete transformer application, which comprises heterogeneous layers, as opposed to a single operation among matrices. This approach allows us to explore the potential benefits and efficiencies of aligning the memory arrangement with the hardware accelerator size in realistic computational scenarios.

2.3.2 Hardware Accelerators for Transformer Models

A comprehensive survey of hardware accelerator implementations for transformer models is provided by [10], discussing state-of-the-art hardware accelerator frameworks and potential research directions in transformer hardware acceleration. One notable accelerator design, proposed by [14], introduces an accelerator on FPGAs, which achieves lower latency and higher energy efficiency compared to CPU, GPU, and prior FPGA-based accelerators. [20] presents a hardware accelerator architecture that utilizes a configurable matrix computing array and on-chip buffers on the FPGA. However, these two papers study accelerators in isolation, as opposed to considering them as part of an overall computing system. On the contrary, here we adopt a system-level view, investigating the interplay between memory hierarchy, accelerator design, and system performance and proposing strategies to minimize data access at lower memory levels.

Our approach is related to [1], which also addresses the challenge of minimizing data transfers by introducing tightly coupled small-scale systolic arrays. These are integrated as custom functional units and are governed by dedicated ISA extensions. The ensuing data reuse results in significant application-wide speed-ups. However, the work in [1] does not consider optimization of the memory data arrangement as a hardware-friendly optimization strategy.

3 Methodology

3.1 Aligning memory arrangement with the accelerator kernel size

Our proposed technique, called Block-Wise Memory Arrangement (BWMA), further enhances the benefits of tiling by optimizing the data arrangement in the off-chip memory. Before illustrating BWMA, we contrast it with a conventional approach [1] for storing two-dimensional matrices in (linear) memory, which is named herein Row-Wise Memory Arrangement (RWMA).

3.1.1 Row-Wise Memory Arrangement (RWMA)

Figure 4a illustrates an 8x8 matrix containing various patterns as rows and different colors in the columns. To store this matrix in memory, it must be converted to a one-dimensional array. As demonstrated in Figure 4c, RWMA stores the matrix row by row in memory, treating them as 1-D arrays. When this matrix is incorporated into a GEMM operation via the tiling technique, irregular non-sequential sections of the 1-D memory are accessed to retrieve the tile. Following the GEMM operation, storing the resultant tile back into memory also necessitates access to non-sequential memory regions.

3.1.2 Block-Wise Memory Arrangement (BWMA)

As shown in Figure 4b, our proposed approach partitions weights, input, and output matrices into smaller blocks. The size of the blocks in this partitioning is as large as the size of the kernel in the hardware accelerator. As discussed in Section 2.2 and shown in Figure 2, the size of the kernel is defined as the number of PEs in each row (in SAs) or lane (in SIMDs). The blocks, as opposed to rows, are then stored in sequence as 1-D arrays in memory (see Figure 4d).

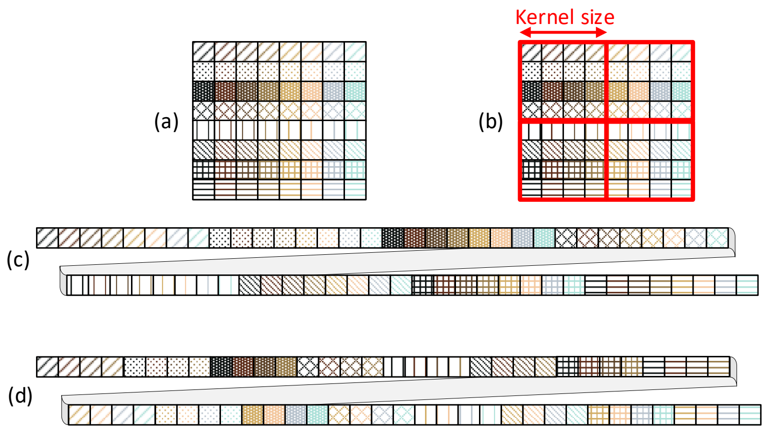


Figure 4 Comparison of conventional row-based memory arrangement (a, c) and the proposed block-wise memory arrangement (b, d) for an 8x8 matrix.

BWMA ensures that GEMM operations fetch data from sequential sections of the 1-D array, as stored in memory. This arrangement is particularly advantageous for systems having a multi-level memory hierarchy, i.e., having a large/slow main memory and multiple layers of progressively faster/smaller caches. In these systems, BWMA allows the expected contiguous data to be pre-fetched correctly into caches and, ultimately, in the hardware accelerator. In fact, BWMA allows for a higher degree of data reuse across the entire memory hierarchy, reducing costly accesses to lower memory levels.

3.2 Impact of BWMA in transformer models

In practical applications, even if the input matrix for a transformer model is structured using RWMA, the transition to-from RWMA and BWMA, is only necessary at the start and at the end of the entire transformer computation process for the input and output matrices. Importantly, there is no necessity to rearrange all the intermediate matrices produced by the GEMM operations, as these can be supplied to the subsequent layer in a block-wise fashion.

Given the multi-layered structure of the transformer and the composition of several components within each transformer layer, as discussed in Section 2, the overhead associated with the RWMA $\leftrightarrow$ BWMA transitions of input and matrices is insignificant in comparison to the overall transformer computation. We observed that they consume only 0.1% of the total execution time for an entire 12-layer transformer model.

Through the transformer model, non-GEMM functions are also influenced by BWMA. The transpose and activation functions are similar, from a computational perspective, in conventional RWMA and block-based BWMA. Softmax and Normalization require proper indexing of blocks in the BWMA case, which has an impact on their run-time. However, this effect is negligible with respect to the run-time of GEMM operation, as overall non-GEMM operations take at most 13.5% of the execution time (even when GEMM is hardware accelerated), as shown in Section 4.2. The impact of BWMA on non-GEMM functions are explained as follows:

Softmax. The softmax layer computes the exponential function for each element in a matrix row and normalizes the result, producing probabilities. The access pattern to elements in a row differs between RWMA and BWMA. Figure 5a illustrates the first eight elements

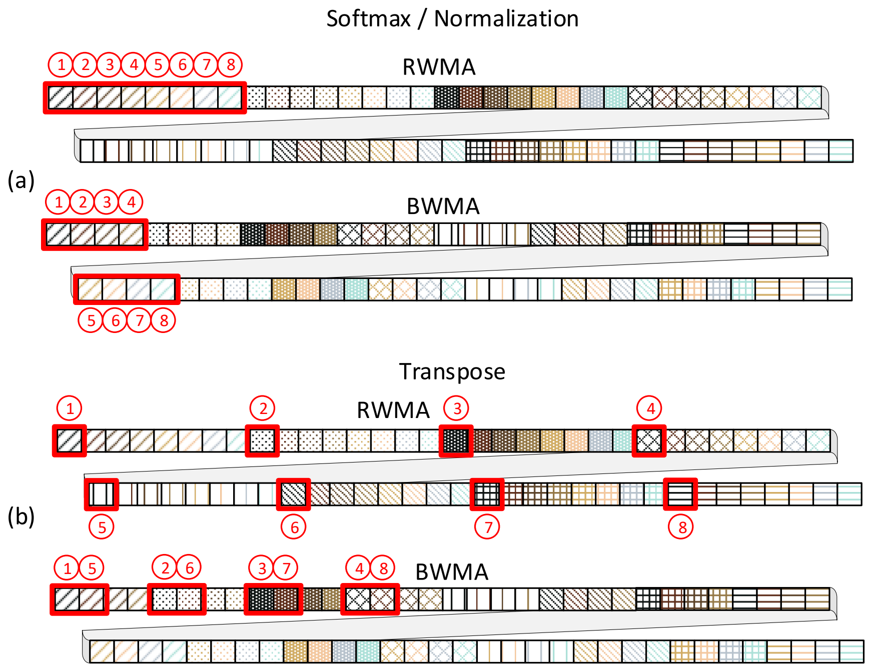


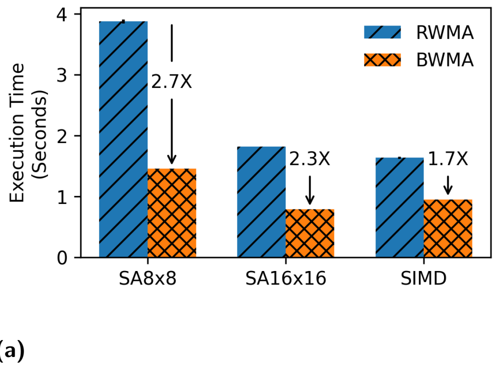
Figure 5 Non-GEMM operation access pattern in RWMA and BWMA for (a) Softmax/Normalization and (b) Transpose layers. The first accessed eight elements are represented in each experiment.

accessed during reading in RWMA and BWMA. As shown, the data is read consequently in RWMA, whereas in BWMA the reading is a non-sequential pattern, introducing overhead in this data arrangement. The processed data is written back to the same matrix position, leading to additional overhead in BWMA compared to RWMA for the write-back operation.

Normalization. This process involves computing a row's element sum, and then calculating the average. Subsequently, the variance is determined using the calculated average and its square root yields the standard deviation. Each row element is normalized by subtracting the average and dividing by the standard deviation. The memory access pattern of the normalization function is identical to the softmax function, as both operate on a row-by-row basis. Therefore, as demonstrated in Figure 5a, this function presents a slight overhead for BWMA compared to RWMA.

Transpose. This operation requires swapping elements in the matrix to change its orientation. The access pattern in this operation differs notably between RWMA and BWMA. Figure 5b shows the initial eight elements accessed in this function for both arrangements. During data reads, neither method accesses the data sequentially; however, BWMA exhibits better data locality. Note that this figure only represents data accessed during reading. Writing back the data in the memory after transpose is sequential for both data arrangements.

Activation. This function is an element-wise operation. Therefore, the change in data arrangement has no effect on the memory access pattern in the activation function. Note that this function is only used in the first feed-forward component of the transformer encoder layer. Due to its element-wise characteristics, it is integrated directly into the feed-forward layer immediately prior to saving the computed values back into the memory. As a result, this particular activation function does not require separate data retrieval from memory.



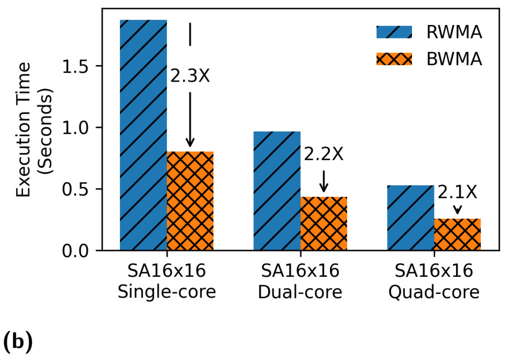


Figure 6 Execution time comparison on BERT model: BWMA demonstrates reduced execution time for transformer models across various (a) hardware accelerators and (b) number of cores.

4 Experimental Setup and Results

4.1 Experimental Setup

We consider BERT-base model [2], as a case study. The input matrix for this model has dimensions of 512x768, where 512 represents the sequence length. The Query, Key, and Value matrices for each of the model's 12 heads have dimensions of 768x64, while the feed-forward matrices have a size of 3072.

We used the gem5-X system simulation framework [16] to explore various performance indicators, including execution time and accesses / failures at different levels of the memory hierarchy. gem5-X is an open-source environment extending gem5 [13] and adding advanced features such as shared guest / host spaces and fine-grained checkpointing. The behavioral models of the accelerators, including the functionality of each defined instruction, are modeled in C++.

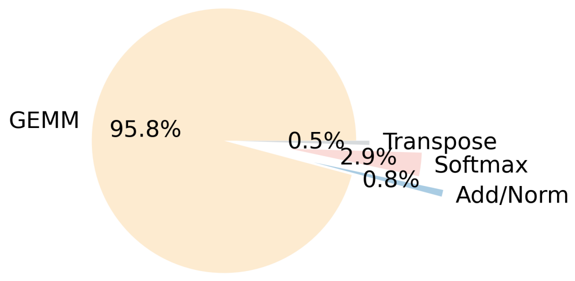
Across experiments, we targeted a hardware system that included a single-, dual-, or quad-core processor. The system features 32 KB of L1 cache for instructions and 32 KB of L1 cache for data for each core, 1 MB of L2 cache shared among the cores, and 4 GB of off-chip memory. Its CPU operates at a frequency of 2.3 GHz. Transformer applications run in full-system mode on the Ubuntu 16.04 operating system.

We considered two classes of accelerators: systolic arrays with a kernel size of 16 and 8 elements and SIMD functional units with a kernel size of 16 elements. In more detail, we employ the gem5-X systolic matrix model from [1], integrated as a custom functional unit as an example of a systolic array. In contrast, we focus on the ARM NEON model, included as part of the gem5 framework, as a SIMD accelerator.

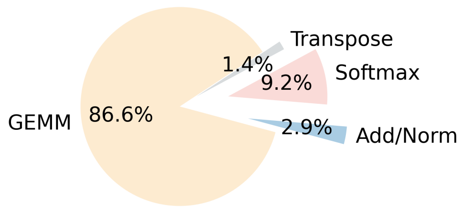
4.2 Impact of BWMA on inference run-time

Figure 6 presents the execution time of a BERT transformer layer when employing various hardware accelerators, such as Systolic Array with a block size of 8x8 (SA8x8), SA16x16, and SIMD on a single-core CPU. As depicted in the figure, BWMA considerably reduces the execution time of the transformer model by up to 2.7x in the SA8x8 case in comparison to RWMA proposed by [1].

In cases where the accelerator has multi-core processing units, BWMA is still very effective in accelerating the transformer model execution. Figure 6b shows the inference run-time for a single-, dual-, quad-core system, where each core embeds dedicated 16x16 SAs. As shown



(a) RWMA.



(b) BWMA.

Figure 7 Execution Time Distribution in SA16x16 in RWMA and BWMA on a single-core system. The area of the pie charts is proportional to the measured inference times (2.3x smaller for BWMA). The run-time of GEMM operation is the majority, even in the presence of hardware acceleration.

in the results, BWMA can decrease the run-time compared to RWMA irrespectively of the number of cores. Furthermore, this figure shows that the inference run-time for a single-core BWMA is even less than that of a dual-core RWMA, highlighting that optimizing the memory arrangement (which has no hardware cost) can be more effective than duplicating the system resources.

A more detailed view of the run-time of RWMA inference for a SA16x16 case with a single-core system is provided in Figure 7a. The figure depicts the proportion of execution time for each individual component. It can be noticed that the non-GEMM components (Transpose, Softmax, and Add/Norm) add up to only 4.2% of the execution time of a whole transformer layer. As discussed in Section 3.2, GEMM operations within these components benefit from memory arrangement, while the non-GEMM components are subject to overhead.

Figure 7b similarly displays the proportion of execution time in a similar way for the BWMA case. In BWMA, the proportion of GEMMs execution time decreases, while the non-GEMM components' run-time increases. Despite this last effect, non-GEMMs only account for 13.5% of the execution time, with the GEMM portion still comprising the majority. Hence, as the decrease in GEMM execution time largely overcompensates for the increased overhead in the non-GEMM components, ultimately resulting in large performance improvements at the application level.

4.3 Impact on memory accesses

The speed-up outlined above is caused by the better use of the memory hierarchy of BWMA with respect to RWMA when performing tiled GEMMs. To further illustrate this aspect, we show the memory accesses in detail for a single- and quad-core system.

Figure 8 displays memory accesses and memory misses for both RWMA and BWMA for a single-core SA16x16. The data are on a logarithmic scale and encompass cache levels as well as the main memory. For both methods, the number of data accesses requested by the processor is almost the same. Thus, the number of L1 data cache (D-cache) accesses remains nearly constant. In contrast, L1 instruction cache (I-cache) accesses are higher in the case of RWMA, because the data in each tile have to be explicitly indexed. This is not the case for BWMA, where the values belonging to each tile are stored in contiguous memory locations. However, although the I-cache accesses for the RWMA are higher, these are well served by the L1 I-cache, with comparatively few misses. In contrast, the number of L2 cache accesses

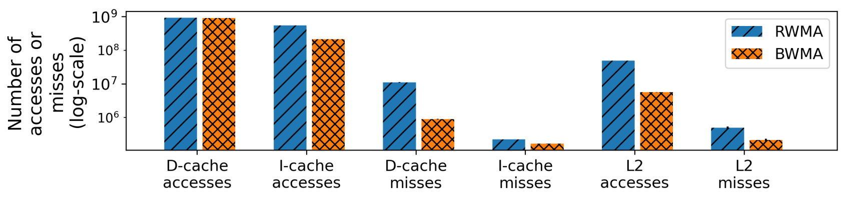


Figure 8 Memory access comparison in a single-core system equipped with SA16x16: BWMA results in significantly reduced L2 cache accesses and cache misses compared to RWMA, showcasing improved data reutilization and performance.

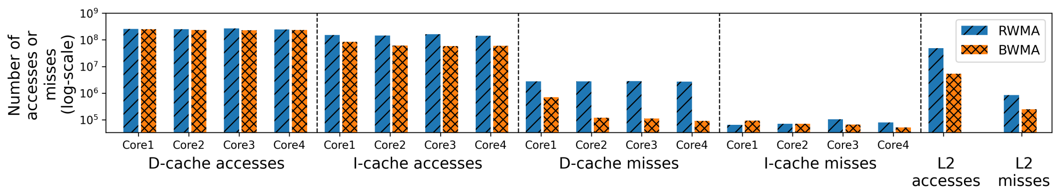


Figure 9 Memory access comparison in a quad-core system equipped with SA16x16. The number of accesses and misses are shown for each core of the target architecture.

is significantly different in RWMA compared to BWMA. Indeed, BWMA leads to a higher data reuse in L1 with 12.3 times fewer L1 D-cache misses and, therefore, much fewer L2 accesses. Given the difference in access latency between L1 (2 cycles) and L2 (20 cycles) caches, the BWMA approach results in much better overall performance.

In Figure 9, we observe a similar memory access and miss pattern during transformer execution on a quad-core system utilizing SA16x16. In this figure, the logarithmic-scale plot illustrates the count of accesses and misses at different levels of the memory hierarchy. As shown in this figure, similar to Figure 8, BWMA exhibits fewer memory accesses and misses compared to RWMA across each core. Additionally, the first core experiences a higher number of D-cache and I-cache accesses and misses compared to the other cores. This discrepancy arises from the fact that, in our study, where GEMM operations are parallelized, only the first core handles non-GEMM operations in this quad-core setup. Nevertheless, the overhead associated with non-GEMM operations is negligible in relation to the overall speedup achieved in the complete execution time of the transformer.

5 Conclusion

This paper has demonstrated the effectiveness of aligning memory data arrangements with hardware accelerators to enhance the performance of transformer models. Our proposed arrangement technique, called BWMA, significantly optimizes memory access patterns, which leads to substantial improvements in processing speed and efficiency. In particular, implementing BWMA in transformer models has shown up to a 2.7x increase in speed for single-core accelerator systems. Also, we have shown that while there is a slight increase in runtime for non-GEMM components due to BWMA, this is negligible compared to the overall computational benefits. The BWMA method is scalable and applicable to various hardware architectures and can be effectively utilized in multi-core systems, hence providing a versatile solution for memory data arrangement in diverse computational environments.

References

1 Alireza Amirshahi, Joshua Alexander Harrison Klein, Giovanni Ansaloni, and David Atienza. Tic-sat: Tightly-coupled systolic accelerator for transformers. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, pages 657–663, 2023.

2 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint*, 2018. `arXiv:1810.04805`.

3 Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint*, 2020. `arXiv:2010.11929`.

4 Grégoire Axel Eggermann, Marco Antonio Rios, Giovanni Ansaloni, David Atienza Alonso, and Sani Nassif. A 16-bit floating-point near-sram architecture for low-power sparse matrix-vector multiplication. In *VLSI SoC*, 2023.

5 Corentin Ferry, Tomofumi Yuki, Steven Derrien, and Sanjay Rajopadhye. Increasing fpga accelerators memory bandwidth with a burst-friendly memory layout. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

6 José R Herrero and Juan J Navarro. Using non-canonical array layouts in dense matrix operations. In *International Workshop on Applied Parallel Computing*, pages 580–588. Springer, 2006.

7 Wei-Ning Hsu, Benjamin Bolte, Yao-Hung Hubert Tsai, Kushal Lakhotia, Ruslan Salakhutdinov, and Abdelrahman Mohamed. Hubert: Self-supervised speech representation learning by masked prediction of hidden units. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 29:3451–3460, 2021.

8 Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

9 Soroosh Khoram, Yue Zha, Jialiang Zhang, and Jing Li. Challenges and opportunities: From near-memory computing to in-memory computing. In *Proceedings of the 2017 ACM on International Symposium on Physical Design*, pages 43–46, 2017.

10 Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W Mahoney, et al. Full stack optimization of transformer inference: a survey. *arXiv preprint*, 2023. `arXiv:2302.14017`.

11 Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.

12 Bingbing Li, Santosh Pandey, Haowen Fang, Yanjun Lyv, Ji Li, Jieyang Chen, Mimi Xie, Lipeng Wan, Hang Liu, and Caiwen Ding. Ftrans: energy-efficient acceleration of transformers using fpga. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 175–180, 2020.

13 Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. The gem5 simulator: Version 20.0+. *arXiv preprint*, 2020. `arXiv:2007.03152`.

14 Hongwu Peng, Shaoyi Huang, Tong Geng, Ang Li, Weiwen Jiang, Hang Liu, Shusen Wang, and Caiwen Ding. Accelerating transformer-based deep learning models on fpgas using column balanced block pruning. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, pages 142–148. IEEE, 2021.

15 Panjie Qi, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Hongwu Peng, Shaoyi Huang, Zhenglun Kong, Yuhong Song, and Bingbing Li. Accelerating framework of transformer by hardware design and model compression co-optimization. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

16 Yasir Mahmood Qureshi, William Andrew Simon, Marina Zapater, David Atienza, and Katzalin Olcoz. Gem5-x: A gem5-based system level simulation framework to optimize many-core platforms. In *2019 Spring Simulation Conference (SpringSim)*, pages 1–12. IEEE, 2019.

17 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

18 Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint*, 2019. `arXiv:1907.10701`.

19 Yuxin Wang, Qiang Wang, Shaohuai Shi, Xin He, Zhenheng Tang, Kaiyong Zhao, and Xiaowen Chu. Benchmarking the performance and energy efficiency of ai accelerators for ai training. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 744–751. IEEE, 2020.

20 Xin Yang and Tao Su. Efa-trans: An efficient and flexible acceleration architecture for transformers. *Electronics*, 11(21):3550, 2022.

21 Haoran You, Zhanyi Sun, Huihong Shi, Zhongzhi Yu, Yang Zhao, Yongan Zhang, Chaojian Li, Baopu Li, and Yingyan Lin. Vitcod: Vision transformer acceleration via dedicated algorithm and accelerator co-design. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 273–286. IEEE, 2023.

22 Juan Zhong, Zheng Liu, and Xi Chen. Transformer-based models and hardware acceleration analysis in autonomous driving: A survey. *arXiv preprint*, 2023. `arXiv:2304.10891`.

Zero-Copy, Minimal-Blackout Virtual Machine Migrations Using Disaggregated Shared Memory

Andreas Grapentin
Operating Systems and Middleware Group, Hasso Plattner Institute, University of Potsdam, Germany

Felix Eberhardt
Operating Systems and Middleware Group, Hasso Plattner Institute, University of Potsdam, Germany

Tobias Zagorni
Operating Systems and Middleware Group, Hasso Plattner Institute, University of Potsdam, Germany

Andreas Polze
Operating Systems and Middleware Group, Hasso Plattner Institute, University of Potsdam, Germany

Michele Gazzetti
IBM Research Europe, Dublin, Ireland

Christian Pinto
IBM Research Europe, Dublin, Ireland

Abstract

We propose a new live-migration paradigm for virtual machines called *zero-copy migration*. By making the working set of the virtual machine available on the destination host through transparently byte-addressable disaggregated memory, we remove the need for a pre-copy phase while simultaneously reducing the performance impact of the post-copy phase. We describe an open-source implementation of the proposed paradigm based on QEMU-KVM and libvirt, and we evaluate the efficiency of the approach with a deployment on a functional hardware prototype of a memory disaggregation system realized using ThymesisFlow. Using a series of configurable benchmarks, we show that the lead time and blackout time of the migration are equal to best-case scenarios of traditional pre-copy, post-copy and hybrid approaches. Key performance metrics from the perspective of applications running in the virtual machine, such as memory latency and throughput, are improved by up to three orders of magnitude, increasing both flexibility and responsiveness of live-migrations in the datacenter.

2012 ACM Subject Classification Hardware → Memory and dense storage; Computer systems organization → Cloud computing; Software and its engineering → Virtual machines; Software and its engineering → Distributed memory; Software and its engineering → Cloud computing; Information systems → Data centers; Information systems → Computing platforms

Keywords and phrases disaggregation, disaggregated memory, vm live migration, thymesisflow, power9, opencapi, performance evaluation, zero copy

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2024.3

Supplementary Material *Software*: `https://github.com/disaggr/qemu/tree/thymesisflow`
archived at `swh:1:dir:b1c1f4d8767857b66e0d2b1c6f2c76177ec9e976`

1 Introduction & Motivation

Flexibility of deployment and serviceability of the infrastructure are key features of the cloud datacenter. Workloads are contained in Virtual Machine (VM) instances as a unit of deployment with diverse resource footprints. This results in an unbalanced allocation of server resources that creates fragmentation over time. VM migration is a useful tool to rearrange workloads, and both mitigate resource fragmentation as well as respond dynamically to deployment challenges without losing availability. However, live-migrating a virtual machine between physical machines incurs a severe penalty on the service quality of the contained workload. Both in literature and in practice it has been established that the performance implications of migrating VM instances with large memory footprints and write-intensive guest workloads can be so substantantial as to render the instance un-migratable [17].

However, modern applications such as graph analytics, in-memory databases and artificial intelligence oriented applications require increasingly large amounts of memory, in the order of hundreds of Gigabytes[21]. Migrating VMs hosting such workloads and transferring such large amounts of data over the network saturates the available network bandwidth, interfering with applications running on the infrastructure, and further exacerbating the performance issues of traditional live-migration approaches.

Techniques exist that are aimed at mitigating these issues, such as using Remote DMAs (RDMA) [9]. While these solutions are indeed effective, they don't completely mitigate the latency degradation during post-copy transfers, and consequently don't solve the problem of migrating very large VM instances.

The advent of Composable Disaggregated Infrastructures (CDI) [3], and in particular the ability of accessing remote memory over a standard PCIe 5.0 fabric thanks to Computer Express Link (CXL) [5] fabric interconnect standard, opens new avenues for improving virtual machine migration techniques. In a CDI, the physical boundaries of a server become indistinct. Memory and compute resources are broken into disaggregated components over a dedicated network fabric and can be composed via software. Existing prototypes show that in the near future, systems may become commercially available that have the ability of transparently accessing memory from a shared pool or from neighboring nodes with a load/store semantic. CXL also shows promise of breaking the bandwidth bottleneck of modern x86 based server processors [2] in a similar manner that the introduction of the Open Memory Interface (OMI) allowed on POWER10™ systems. CXL over ethernet is also being explored [20], both with the possibility of forming memory pools, but also for allowing disaggregated access to a nodes private memory.

In this paper we present the first implementation of virtual machine live-migration based on disaggregated memory. To evaluate the approach on real hardware, we utilize the open-source disaggregated memory prototype, ThymesisFlow [18] that enables borrowing memory from a neighboring machine. Although ThymesisFlow relies on the OpenCAPI [14] coherent interconnect and is targeting IBM POWER9™ based server systems, the approaches outlined in this paper are independent of the choice of disaggregated memory interconnect and are applicable to similar architectures created on the basis of CXL or other technologies. We investigate how traditional live-migrations of virtual machines can utilize disaggregated memory as a medium for memory access and transfer, in order to improve the performance metrics from the perspective of both the hypervisor and the guest workload during the migration.

2 State of the Art

VM live migration was first presented as a response to the necessity for more flexible workload distribution in the datacenter by Clark et al. and Nelson et al. on both the Xen and VMWare VSphere virtualization platforms [1][4][12]. Virtual machine instances present the opportunity of migrating self-contained units of work, mutually independent

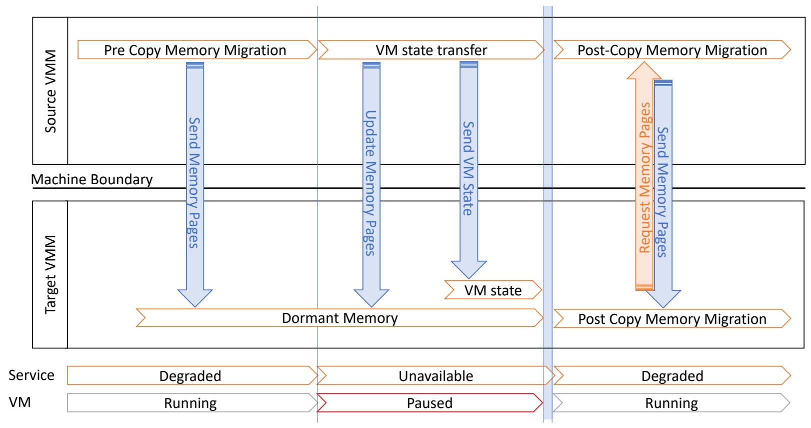


Figure 1 The phases of virtual machine migration, highlighting the duration in which the machine is paused and thus the service unavailable (*blackout* time) as well as the duration in which the service is degraded (*brownout* time).

and decoupled from the operating system, as opposed to previous work which focused on the migration of individual processes. Later approaches extended the scope of live virtual machine migration to new transfer media such as wide area networks [7] and RDMA [19]. To further mitigate the performance impact of live-migrations, the *pre-copy* paradigm heavily utilized in traditional VM migrations has been steadily improved [8, 11] and the *post-copy* paradigm was introduced [6], enabling hybrid migrations that combine the benefits of pre- and post-copy. In 2018, Ruprecht et al. have described the application of VM migration in Google datacenters and outlined the necessity for an improved deployment flexibility with minimal application downtime. They have also shown that using state-of-the-art migration technologies, the performance impact on guest systems is still a relevant concern [16].

2.1 Sequence of a Traditional Hybrid VM Live-Migration

Figure 1 outlines the sequence of a standard virtual machine live-migration utilizing both a *pre-* and *post-copy* phase and summarizes their associated impact on the guest. First is a *pre-copy* phase, in which memory pages are transferred concurrently from the *source* to the *destination* host. The degradation of the guest system due to the introduced contention on the memory subsystem is described as a *pre-copy brownout*.

After concluding the pre-copy phase, the virtual machine is paused on the source and the state of the runtime is transferred to the destination to be resumed there. In the *blackout* time between pausing and resuming the virtual machine, the services provided by the guest are unavailable.

Lastly, when the guest has been successfully resumed on the destination, the remaining memory pages are transferred in a *post-copy* phase. During this phase the guest performance can be degraded significantly if memory pages are accessed that are not yet locally available on the destination, leading to a *post-copy brownout* where the application must wait for the memory to be transferred from the source before execution can continue.

2.2 Beyond State of the Art

While both post-copy and pre-copy phases of the virtual machine migration have been shown to be useful techniques to reduce the duration of the blackout time and increase service availability, both techniques have disadvantages. During the pre-copy phase, pages that are overwritten (*dirtied*) by the guest after successful transfer to the destination will need to be invalidated, and eventually re-transferred. As a consequence, workloads that produce frequent write accesses drastically reduce the effectiveness of a pre-copy phase[10]. Conceptually, the pre-copy phase also introduces a *lead-time* delay between initiating the migration and the release of the compute resources on the source, which may be undesirable when the migration needs to be performed quickly, for example to reduce thermal load.

Similarly, the post-copy phase can cause severe performance issues if the guest frequently needs to wait for the remote memory to become available. Access to pages not yet migrated would require the generation of traffic on the network, a physical link characterized by much higher latency compared to the memory bus. As a consequence, applications that exhibit unfavorable *memory access patterns* can become unresponsive during traditional migrations, leading to service downtime much longer than the machines apparent blackout time.

The introduction of disaggregated memory with the paradigm of zero-copy live-migrations allows the migration of virtual machines unimpeded by these issues. Zero-copy live-migration means that the memory of the guest can be made immediately and transparently byte-addressable on the destination without the need to pre-copy any memory, and without incurring expensive page-faults when accessing remote memory. The virtual machine can be paused on the source and immediately resumed on the destination, eliminating the lead-time delay and minimizing the blackout time. Coupled with a modified post-copy phase also utilizing the disaggregated memory to efficiently copy and re-map the memory from the source to the destination, the virtual machine can be migrated completely with greatly reduced performance degradation during the post-copy brownout phase when compared to traditional live-migrations.

Very recently, first steps have been made to design a virtual machine migration over CXL by the *nil-migration* project [13], utilizing CXL 3.0 to create a shared disaggregated memory pool device. To perform the migration, the working set of the virtual machine on the source will be transferred to pooled memory, and then transferred back to local memory on the destination. However, no concrete implementation and evaluation of the technique is available at the time of writing this article. In our prototype, we don't use a memory pool and instead directly promote the private memory of the source machine that contains the working set of the virtual machine instance to shared memory. However, the same approach could be used in case of a pooled memory system.

3 Architecture of the Memory Disaggregation Prototype

This work is based on a real hardware-based disaggregated shared memory system, implemented thanks to the ThymesisFlow [15] [18] open-source project. ThymesisFlow is a HW/SW framework that uses OpenCAPI [14] attached FPGAs to enable memory disaggregation across IBM POWER9™ servers via the memory borrowing template (Figure 2). The minimum working system is composed of two nodes: a *lender* and a *borrower*. The *lender* node is capable of reserving a portion of its local main memory for access from a remote machine. The borrower accesses memory from a remote lender using load/store semantics, as if the memory was physically attached to it. The remote memory is attached to the lender machine using either the abstraction of an additional, CPU-less, NUMA node or as a memory

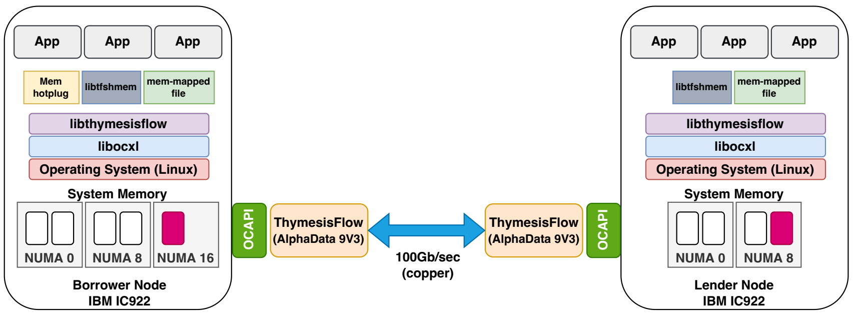


Figure 2 Abstract representation of the ThymesisFlow HW/SW prototype.

mapped file. On the hardware side, each node part of a ThymesisFlow deployment must support OpenCAPI and be equipped with a supported FPGA card. In the case of this paper, we used IBM IC922 POWER9™ servers and AlphaData 9V3 FGPAs.

On the software side, ThymesisFlow offers *libthymesisflow* which consists of a set of user-space applications that are used for controlling the configuration of the FPGAs and the actual attachment of memory to the borrower OS memory subsystem. Memory in ThymesisFlow can be assigned in exclusive mode to the borrower or shared between borrower and lender machine. In the exclusive mode, the borrower can decide to hotplug the memory to the OS or to access it via a custom `mmap` and Linux character device (useful for assigning memory to a specific application/process). In the shared case, instead, both parties can either use the shared memory library (*libtfshmem*) provided as part of the ThymesisFlow suite or have the memory exposed as a memory mapped file. In this paper we leverage the shared memory via memory mapped file.

Logically this disaggregated memory architecture is what we call a *peer-to-peer* approach, where every node in the system is able to map every other nodes memory. In contrast, there is the *pool* architecture with a separate memory pool, which nodes with private memory can map and share. While we require the former architecture, it is possible to adapt our prototype to the latter as well. Additionally, even though the underlying technology is OpenCAPI, which has been merged with the CXL Consortium in 2022, our techniques can be applied to CXL as well. That said, the integration of CXL in Thymesisflow is beyond the scope of this work.

4 Design of the VM Migration

We implemented the disaggregated zero-copy virtual machine migration by extending the open-source virtualization and system emulation software *QEMU-KVM*[1]. *QEMU-KVM* is a *type 1 hypervisor* in Linux, capable of supporting both hardware accelerated virtualization backends, as well as advanced platform and orchestration abstractions such as *libvirt*. For the live-migration of virtual machines, QEMU already natively supports the pre- and post-copy paradigms. The existing advanced migration capabilities of *QEMU*, coupled with the well-documented and active open-source codebase and community made it an ideal testbed for implementing the zero-copy live migration. To implement our prototype, we extended QEMU to enable a migration from the disaggregated memory *lender* to the *borrower*.

[1] Our modifications to QEMU are also open source and available via `https://github.com/disaggr/qemu/tree/thymesisflow`

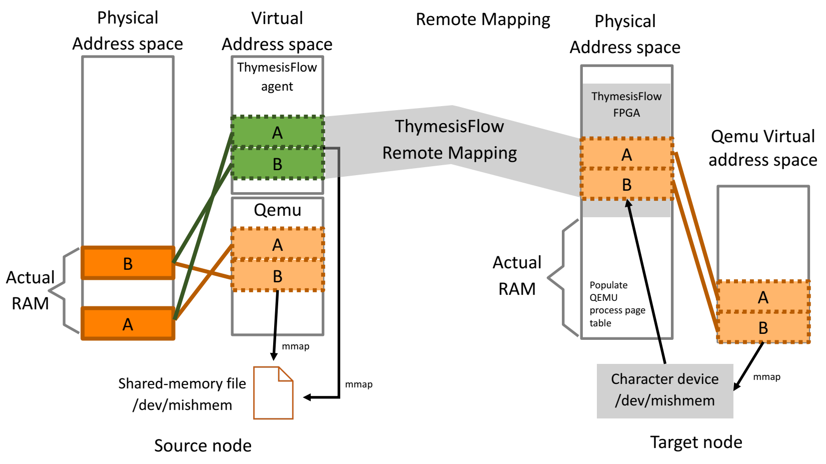


Figure 3 A conceptual visualization of the page table mappings provided to QEMU during the zero-copy migration.

To enable the disaggregated live-migration, we replace the live-migration memory moving behavior of QEMU both on the source and the destination by supplying the relevant function tables with custom implementations. We disable the pre-migration handling of all RAM blocks, so no pre-copy phase can occur on the source, and install a custom handler for the post-copy phase.

4.1 Organizing the Memory Backing Store

In order to make the disaggregated memory available to the virtual machine on both the source and destination, we utilize the file-mapped memory backends implemented in QEMU to provide the VM with the same view of its memory on both the borrower and lender side. To achieve this, we create a shared memory segment of sufficient size on the lender side and supply its path to QEMU. This memory segment is shared with the *ThymesisFlow Agent* that communicates with the FPGA and that owns the memory that is lent to the borrower. By using a shared memory segment for this lent memory, we can make sure that the memory presented on the lender side is consistent with the memory on the borrower side after migration.

On the borrower, the system is configured with a custom kernel module to interface with the FPGA and present the remote memory as a special character device that supports the *mmap* system call. We configure QEMU to use this character device as mapping file for the VM memory. This ensures that the order of the memory pages is preserved from lender to borrower side, making the view on the physical address space from the perspective of the guest identical on both source and destination. The layout of the addresses and translations is visualized in Figure 3. This way, with one exception outlined below, all memory remains transparently accessible to the guest machine at all times, even while the memory is being copied during the post-copy phase.

4.2 Disaggregated Post-Copy Transfers

To implement a disaggregated post-copy transfer, we start a concurrent thread that incrementally moves the VM memory from the source to the destination by copying and immediately remapping pages of memory as outlined in Figure 4.

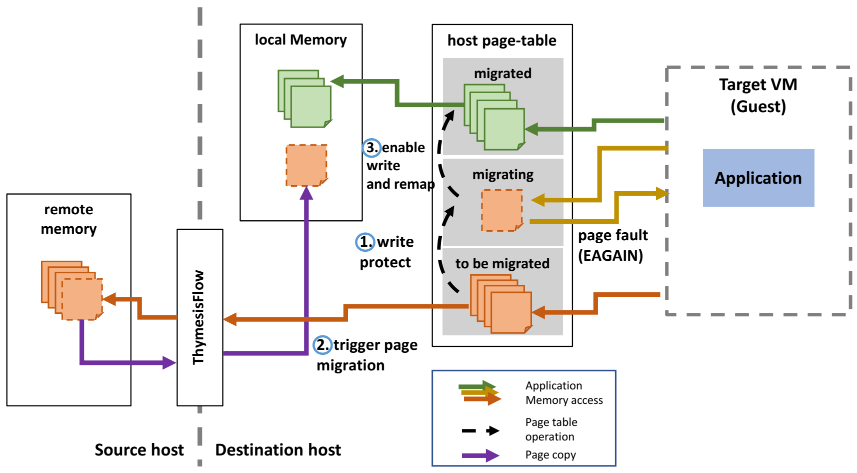


Figure 4 A conceptual visualization of the RAM transfer in QEMU during the disaggregated post-copy phase.

Each page is transferred in three steps. Firstly, to avoid data loss during the small time window of transfer, the page is briefly locked for writing, such that no VM thread may be modifying a page that is currently being moved from source to destination. This lock causes an access violation to occur in case the guest does write to the locked memory, upon which the failed memory access is deferred and retried. Secondly, the page is transferred from the lender to the borrower. Finally, the page is used to replace the mapping of the corresponding page of remote memory in the VMs working set, and the write lock is lifted, completing the page transfer.

4.3 Finishing the Migration

The remainder of the virtual machine state, such as the CPU registers and the device configurations, are transferred via the network using the existing implementation in QEMU. If the disaggregated post-copy phase is enabled, all memory of the VM will be local on the destination after the migration completes. Otherwise, in *cpu-only* mode, the live-migration completes with all memory remaining remotely but still entirely transparently byte-addressable on the borrower machine.

5 Performance Evaluation

In the evaluation of the performance characteristics of the different types of VM migration, we distinguish between two types of performance metrics that we can observe in our testbed.

- *Macro-properties* of the migration are performance characteristics observable by the host system and the virtual machine process, including the *total time of migration*, *total ram transferred* and the *blackout* and *pre- and post-brownout times* of the guest.
- *Micro-properties* of the migration are performance characteristics that are internal to the VM and are observable by the guest workload processes. These metrics include the *memory latency* and the *application throughput* during the migration.

To evaluate the performance characteristics of the zero-copy virtual machine migration, we compare our approach against state-of-the-art hybrid live-migration and analyze individual migration runs to gain insight on the behavior of the guest in order to determine the impact of a migration.

5.1 Infrastructure setup and orchestration

The testbed used for this evaluation is described in Section 3. A third machine on the same network was used to host the evaluation orchestrator and remotely collect the performance data of the migration.

We use *ansible* playbooks and *libvirt*[2] to automate repeatable runs of the benchmarks, as well as orchestrate the virtual machines and make the required hypervisor calls. The virtual machine subjected to the live-migration is a *Debian bullseye GNU/Linux* installation with 2 virtual CPUs, 20GiB of RAM, and configured with an unmodified kernel and firmware[3].

5.2 Description of the Benchmarks

One of the most influential factors on the migration of a virtual machine is the memory behavior of the guest. In particular, the rate at which the guest machine produces *dirtied* pages of memory can influence both the macro and micro performance characteristics of the migration. A second important factor to consider is the memory access pattern of the guest workload. For this reason, we implemented the *SMOG* microbenchmark suite[4] capable of producing dirtied pages in the guest at a configurable rate. We also extended *SMOG* to be capable of accessing memory at configurable access patterns, such as a randomized pointer-chaser workload. These configurable workloads are a useful tool to examine the micro performance characteristics of the guest during migration, in particular memory latency and throughput.

5.3 Memory Latency Degradation during the VM Migration

In this work, we try to investigate the detailed performance characteristics of the guest under the stress of the migration, where certain areas of the memory may still be unavailable, or only available at high latency costs.

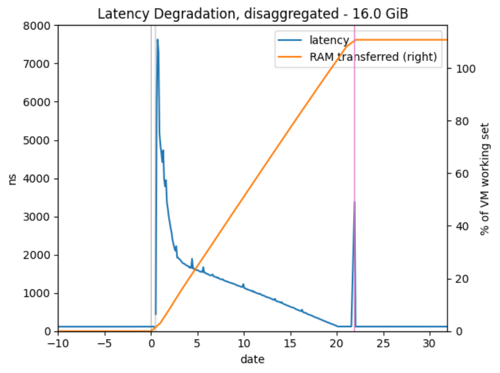
Figure 5 visualizes the average latency in nanoseconds of the randomized pointer-chasing workload of *SMOG* over a 16GiB memory region over the course of the migration. The graphs show that before the migration starts and after it concludes, the memory latency rests at a consistent low average of 80 nanoseconds, which is in alignment with the local memory latency of the machines in our testbed.

During the migration, the impact of the remote memory on the workloads average latency can be separated into two phases. With the start of the migration at a time of 0.0 seconds, the measurements show a sharp increase in latency. This is due to the high latency of the memory, which at this point resides entirely in the remote memory of the lending machine. Since the caches on the borrowing machine are cold and the page tables of the guest machine, which are required by the guest to make memory accesses are also not prefetched, the first accesses to the remote memory are expensive. Through the caching of the page table structures in the L3 cache as well as the beginning transfer of the physical memory region of the VM, we can observe a sharp decrease of the memory latency during the first few seconds of the migration. Once the majority of the page tables used by the benchmark have been transferred or cached,

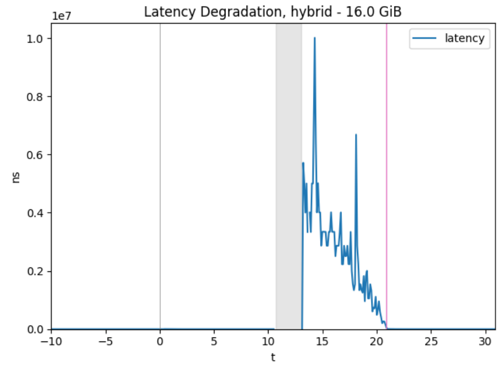
[2] The playbooks and scripts used to create the performance measurements used in this evaluation are available on github: `https://github.com/disaggr/vm-migration-public`

[3] The scripts used to bootstrap and configure the guest systems is available on github via `https://github.com/disaggr/qemu-image-factory`

[4] the SMOG benchmark suite is open-source and available on github via `https://github.com/disaggr/smog`



(a) Memory latency degradation during a disaggregated post-copy migration.



(b) Memory latency degradation during a traditional hybrid migration.

Figure 5 The memory latency as measured by a randomized pointer-chasing workload during both state-of-the-art hybrid and disaggregated zero-copy migrations.

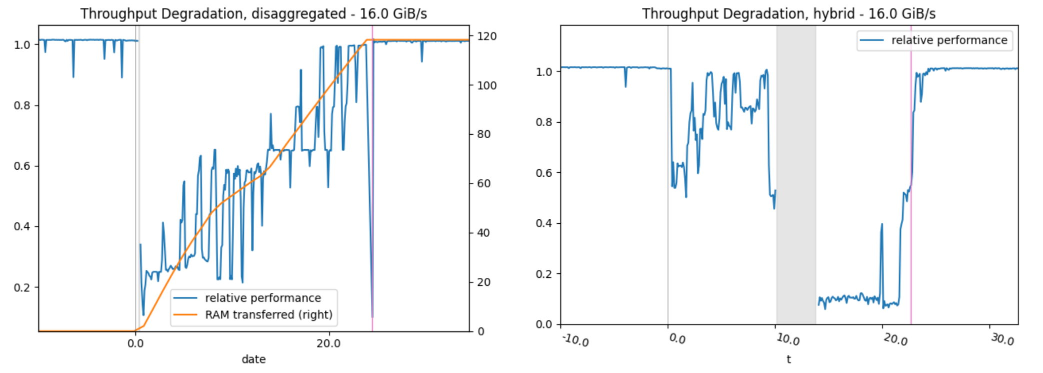
the memory latency decrement levels out into the second phase, in which the amount of transferred working set pages of the pointer chasing workload determines the average latency, which decreases linearly with the remaining portion of remote pages.

In total, approximately 10 percent of the memory has to be transferred more than once during the migration. This corresponds to cache lines that are requested by the guest MMU and provided by ThymesisFlow before the pages are transferred by the RAM mover thread in qemu. When compared to the hybrid migration, it becomes evident that the average memory latency of the guest workload during the zero-copy migration is up to 3 orders of magnitude lower than for the state-of-the-art approach. There are two reasons for this dramatic difference. Firstly, the post-copy migration relies on expensive page-fault mechanisms to make the remote memory locally available. Secondly, the disaggregated memory benefits from a finer granularity of the transferred memory, since only cache-lines of 128 Bytes are transferred for each remote memory request, whereas traditional post-copy transfers entire pages of 4 KiB of memory, which further impacts the average memory latency.

5.4 Memory Throughput Degradation during the VM Migration

The second metric we investigated is the throughput of SMOG processing a sequential combined read/write workload with strides equal to the memory page size. For this benchmark, SMOG was configured to a nominal throughput of 16 GiB/s of memory pages every second, and we measured the actual throughput during migration and visualized the results in Figure 6.

Similarly to the compared memory latency metrics, the throughput of the benchmark running in the disaggregated post-copy phase consistently outperforms the benchmark running in the traditional post-copy phase. It is noteworthy that the pre-copy phase of the hybrid migration provides little benefit to both latency and throughput due to the write-heavy workload configuration.



(a) Memory throughput degradation during a disaggregated post-copy migration. **(b)** Memory throughput degradation during a traditional hybrid migration.

Figure 6 The memory throughput as measured sequential memory read/write workload during both state-of-the-art hybrid and disaggregated zero-copy migrations.

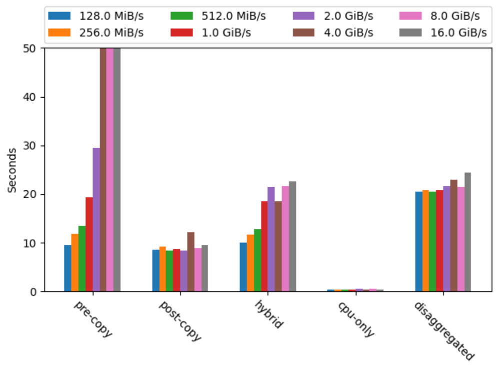
5.5 Analysis of Migration Times and Transferred Ram

We additionally compared two types of performance metrics of the virtual machine migration from the perspective of the hypervisor. Firstly, the total amount of memory transferred as an indicator of increased contention on the network, which may be shared between the implementation of the migration and services running on the same network. In this context, the following times are particularly of interest.

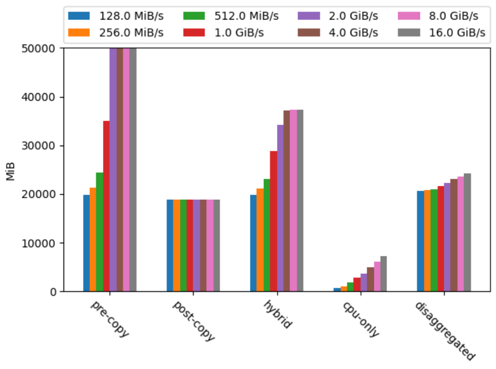
- *Total Time of Migration* – The time between initiating the migration until it is completed.
- *Blackout Time* – The time in which the guest is paused, and no CPU cycles are allocated to applications and services running in the guest.
- *Pre- and Post-Copy Brownout Time* – The time in which the performance of the applications and services running in the guest are degraded because of an active pre-copy or post-copy transfer.
- *Migration Delay* – The time between initiating the migration until the execution of the guest is paused and the last CPU cycle is allocated to the guest and its applications and services running on the source machine.

Some of these metrics are independent of the differences between a zero-copy and a traditional live-migration. For instance, the blackout time of zero-copy live-migrations is guaranteed to be equal to the best case of traditional migrations. Also, the total time of migration as well as the duration of the brownouts will only depend on the size of the VM working set and the bandwidth of the transfer medium. However, by using a dedicated medium for the disaggregated memory transfer, we reduce resource contention on the network, which may be relevant to some applications. A comparison of the total time of migration, and the related metric of total amount of ram transferred are outlined in Figure 7 for different amounts of memory dirtying rate by the guest workload, as well as different types of VM migration.

Our measurements reaffirmed the known issue that pure pre-copy migrations can become unable to meet the threshold of valid pages on the destination for higher rates of dirtied memory in the guest. Because these are academic edge-cases, we are not showing the full extent of these effects on the plots. In real world scenarios, typically a hybrid live-migration approach is used, which limits the number of passes during the pre-copy phase, but still uses



(a) The total time of migration.



(b) The total amount of ram transferred.

Figure 7 A comparison of the total time of migration and total transferred memory during both traditional and disaggregated virtual machine migrations.

a pre-copy phase to help mitigate the performance impact of the post-copy phase. In our experiments, the hybrid migration introduced a maximum overhead of the size of the memory region used by the benchmark. The disaggregated migrations also incur a slight overhead in transferred ram due to memory that is being accessed remotely by the application before being transferred to the borrower. This overhead is also visible in the total time of migration, but this comparison is dominated by a difference in the throughput between our hardware prototype and the network. We expect that future hardware for disaggregated memory will show even more impressive results and easily bridge that gap in throughput.

In consequence, the zero-copy virtual machine migration paradigm is capable of migrating virtual machine instances regardless of working set size or workload behavior with both a greatly reduced performance impact on the guest, as well as a more predictable runtime behavior than traditional virtual machine migration techniques, pushing the boundaries for datacenter workload placement flexibility and transparency of deployment.

6 Conclusions and Future Direction

In this paper, we have introduced the concept of a disaggregated zero-copy virtual machine migration utilizing byte-addressable, disaggregated shared memory, for which we have used the ThymesisFlow prototype technology. We have shown that using disaggregated memory for the resource sharing and transfer during a virtual machine migration is beneficial in two ways. Firstly, from the perspective of the hypervisor it shows potential to improve the flexibility of migrating resources through reduced delays, and to mitigate problems with write-intensive guests. Secondly, from the perspective of the application the degradation of memory latency and throughput are greatly reduced when compared to traditional post-copy migrations. In conclusion, we believe that resource migration, and in particular virtual machine migrations in cloud datacenters are a good use-case for newly emerging memory disaggregation technologies. In future work, it needs to be evaluated whether memory pools could also be utilized in a similar manner, and how CXL shared resources as defined in the recently released CXL 3.0 standard could be utilized to that regard, to make the concept outlined in this work more widely available. In the future, fully disaggregated virtual machine working sets could become possible, where the migration could be reduced to merely reallocating the compute resources of the virtual machine instances inside the datacenter.

References

1 Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

2 Albert Cho, Anish Saxena, Moinuddin Qureshi, and Alexandros Daglis. A case for cxl-centric server processors. *arXiv preprint*, 2023. `arXiv:2305.05033`.

3 I-Hsin Chung, Bulent Abali, and Paul Crumley. Towards a composable computer system. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 137–147, 2018.

4 Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286, 2005.

5 Computer Express Link. `https://www.computeexpresslink.org/`.

6 Michael R Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review*, 43(3):14–26, 2009.

7 Wei Huang, Qi Gao, Jiuxing Liu, and Dhabaleswar K Panda. High performance virtual machine migration with rdma over modern interconnects. In *2007 IEEE International Conference on Cluster Computing*, pages 11–20. IEEE, 2007.

8 Khaled Z Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.

9 C. Isci, J. Liu, B. Abali, J. O. Kephart, and J. Kouloheris. Improving server utilization using fast virtual machine migration. *IBM Journal of Research and Development*, 55(6):4:1–4:12, 2011. `doi:10.1147/JRD.2011.2167775`.

10 Canturk Isci, Jiuxing Liu, Bülent Abali, Jeffrey O Kephart, and Jack Kouloheris. Improving server utilization using fast virtual machine migration. *IBM Journal of Research and Development*, 55(6):4–1, 2011.

11 Yuji Muraoka and Kenichi Kourai. Efficient migration of large-memory vms using private virtual memory. In *Advances in Intelligent Networking and Collaborative Systems: The 11th International Conference on Intelligent Networking and Collaborative Systems (INCoS-2019)*, pages 380–389. Springer, 2020.

12 Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. Fast transparent migration for virtual machines. In *USENIX Annual technical conference, general track*, pages 391–394, 2005.

13 nil-migration. `https://nil-migration.org`.

14 OpenCAPI Consortium. OpenCAPI Specification. Online: `http://opencapi.org`, 2017. Accessed: January 2019.

15 C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. Hofstee. Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 868–880, Los Alamitos, CA, USA, October 2020. IEEE Computer Society. `doi:10.1109/MICRO50266.2020.00075`.

16 Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. Vm live migration at scale. *ACM SIGPLAN Notices*, 53(3):45–56, 2018.

17 Petter Svard, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 111–120, 2011.

18 ThymesisFlow. `https://github.com/OpenCAPI/ThymesisFlow`.

19 Franco Travostino, Paul Daspit, Leon Gommans, Chetan Jog, Cees De Laat, Joe Mambretti, Inder Monga, Bas Van Oudenaarde, Satish Raghunath, and Phil Yonghui Wang. Seamless live migration of virtual machines over the man/wan. *Future Generation Computer Systems*, 22(8):901–907, 2006.

20 Chenjiu Wang, Ke He, Ruiqi Fan, Xiaonan Wang, Yang Kong, Wei Wang, and Qinfen Hao. Cxl over ethernet: A novel fpga-based memory disaggregation design in data centers. *arXiv preprint*, 2023. `arXiv:2302.08055`.

21 Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015. `doi:10.1109/TKDE.2015.2427795`.

Precision Tuning the Rust Memory-Safe Programming Language

Gabriele Magnani
DEIB – Politecnico di Milano, Italy

Lev Denisov
DEIB – Politecnico di Milano, Italy

Daniele Cattaneo
DEIB – Politecnico di Milano, Italy

Giovanni Agosta
DEIB – Politecnico di Milano, Italy

Stefano Cherubin
NTNU – Norwegian University of Science and Technology, Trondheim, Norway

Abstract

Precision tuning is an increasingly common approach for exploiting the tradeoff between energy efficiency or speedup, and accuracy. Its effectiveness is particularly strong whenever the maximum performance must be extracted from a computing system, such as embedded platforms. In these contexts, current engineering practice sees a dominance of memory-unsafe programming languages such as C and C++. However, the unsafe nature of these languages has come under great scrutiny as it leads to significant software vulnerabilities. Hence, safer programming languages which prevent memory-related bugs by design have been proposed as a replacement. Amongst these safer programming languages, one of the most popular has been Rust. In this work we adapt a state-of-the-art precision tuning tool, TAFFO, to operate on Rust code. By porting the PolyBench/C benchmark suite to Rust, we show that the effectiveness of the precision tuning is not affected by the use of a safer programming language, and moreover the safety properties of the language can be successfully preserved. Specifically, using TAFFO and Rust we achieved up to a 15× speedup over the base Rust code, thanks to the use of precision tuning.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Software safety; Software and its engineering → Software performance; Computer systems organization → Embedded software

Keywords and phrases Approximate Computing, Memory Safety, Precision Tuning

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2024.4

Funding The authors gratefully acknowledge funding from European Union's Horizon 2020 Research and Innovation programme under the Marie Skłodowska Curie grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation, `http://www.apropos.eu/`). Also the Italian MISE project F/340043/01-03/X59 "Piattaforma ed ecosistema cooperativo, C- ITS ETSI standard per la mobilità digitale integrata" within the framework FCS - Accordi Innovazione Settore Automotive (DM 31/12/2021 e DD 10/10/2022).

1 Introduction

Proper memory management is crucial to ensure program stability and prevent software vulnerabilities which can be exploited by cyber-criminals to cause unauthorised actions for their own benefit. In fact, memory-related weaknesses remain a major concern in software development, as highlighted in the *Top 25 Most Dangerous Software Weaknesses* report by Mitre. Out-of-bounds writes held the top spot for three consecutive years, while several other memory-related issues, such as *use-after-free* and *NULL* pointer dereference, appear on the list every year [6]. This is also stated by the United States National Security Agency (NSA), which in a recent report (at the time we are writing) advocates the use of memory-safe programming languages [10]. However, the implementation of preemptive measures to ensure memory safety in programming languages can have an impact on performance. Specifically, the use of garbage collection and array bounds checking, which are common in memory-safe languages, can result in significant performance overhead. As a result, while memory-safe languages have been available for a long time, the embedded world is still dominated by the use of C and C++, which give no guarantee on how the programmer handles the memory. Some memory-safety features have been added to these languages over the years, specifically in C++ [7], however they are opt-in and not universally adopted. The reason is that the ability of C and C++ to operate at lower (unsafe) abstraction levels is *desired* by programmers in order to avoid the inefficiencies generated by memory-safety features. Indeed, some of the first memory-safe languages such as Java and C# completely disallow the use of unsafe concepts such as pointers. As a result, more recent language designs provide *safe* and *unsafe* subsets that also allows their use in low-level programming tasks.

One language of this sort that has gained considerable traction in the industry [8] is Rust[1]. Its goal is to allow the adoption of these languages in security-sensitive applications such as operating system development and embedded systems. According to the 2023 Annual Stack Overflow Survey [11], Rust is positively ranked among people who want to learn a new programming language – 6^{th} in the *desired* programming language category – but what makes it really attractive is the experience of using it. Rust placed first in the *admired* category with a score of 84% – twice as much higher than Java and C, and more than 20% better than C#. The *admired* category represents the proportion of users who have used the same technology in the past year and want to continue using it. As such, it is reasonable to expect that the popularity of this language will continue to grow in the coming years.

In general, software development for embedded systems often requires the use of specific optimization techniques to cope with reduced computational capabilities and memory sizes. Among these techniques is Precision Tuning [3], which trades off result accuracy for time and energy efficiency by reducing the bit width of an operation, or by switching from floating point to fixed point arithmetic. Precision Tuning is part of the larger family of Approximate Computing (AxC) [12]. Precision Tuning impacts the data width, and therefore all memory accesses. As such, it can help to offset some of the penalties introduced by the memory management integrated within modern programming environments like Rust. However, at the time of writing and to the best of the authors' knowledge, no Precision Tuning tool supports Rust. Among the most notable recent efforts in the field of precision tuning, we notice that TAFFO [1] is an automated Precision Tuning set of plugins that leverages the LLVM compiler framework [9]. As such, it provides a useful baseline that could be integrated with other compiler components based on LLVM, which is currently the industry standard for Rust compiler development. Indeed, *rustc* – the reference Rust compiler implementation – is based on this framework.

In this work, we aim at filling the gap with automated Precision Tuning support in memory-safe programming languages. To this end, we use Rust as a test-bed for a proof-of-concept implementation that employs TAFFO as the engine for performing the analyses and transformations required for this kind of approximate computing task. More specifically, we integrate the TAFFO framework with *rustc* by providing the appropriate components that

[1] `https://www.rust-lang.org`

allow Rust code to have the programmer-inserted metadata and annotations required by TAFFO to work. As a side-effect, this demonstrates the language-independence of TAFFO, as well as its ability to optimise the code without negatively affecting memory safety.

We evaluate the effectiveness of TAFFO applied to Rust benchmarks, when compared to its effectiveness on C versions. The results show that TAFFO provides similar performance improvements in both C and Rust, when the *rustc* compiler is able to optimize away the out-of-bound access guards. When this is not possible, the benefits of TAFFO increase.

1.1 Key contributions

The main contribution of this work is the introduction of a precision tuning framework in the context of a memory-safe programming language. This task is non-trivial due to Rust's inability to classify as memory-safe some of the features required to perform precision tuning of selected variables. We therefore show that those features, when used to convey the information to TAFFO, do not lead to actual unsafe actions, and that the TAFFO precision tuning process – after minor adjustments – does not harm the memory safety guarantees enforced by Rust.

As additional contribution, we provide a new Rust port of the PolyBench suite, that is closer in design to the original PolyBench/C, to allow a better comparison between the *rustc* and Clang compilers when employing a particular optimisation, in this case precision tuning.

1.2 Structure of the paper

The rest of this paper is organised as follows. In Section 2, we cover the background on TAFFO and Rust. In Section 3, we describe the proposed approach for precision tuning a Rust program with TAFFO. In Section 4 we discuss the effectiveness of the integration, first by describing the experimental setup, and then by providing an assessment of the integrated system. Finally, in Section 5, we draw some conclusions and outline future directions.

2 Background

TAFFO and Rust are two different but complementary technologies. While Rust is a modern programming language, TAFFO is a set of compiler passes for precision tuning. In this section we discuss the peculiarities of these two tools in order to provide a background for the later discussion of their integration.

2.1 Memory-Safety Properties of Rust

To achieve its memory-safety properties, the Rust language enforces strict rules to prevent common programming errors such as *NULL* pointer dereferences and buffer overflows. In particular, the *rustc* compiler employs a dedicated component, the *borrow checker*, to enforce these rules. Among others, the borrow checker ensures that variables are initialised before they are used. It ensures that values, whether they are held in variables or temporaries, are not moved twice, or while borrowed. Moreover, it ensures no variable is accessed while mutably borrowed (except through the reference), and no variable is mutated while immutably borrowed. In general, Rust's ownership system [5, 13] is based on the idea that if a certain object (of type `T`) is owned by multiple aliases (`&T`), then none of them can be used to modify it.

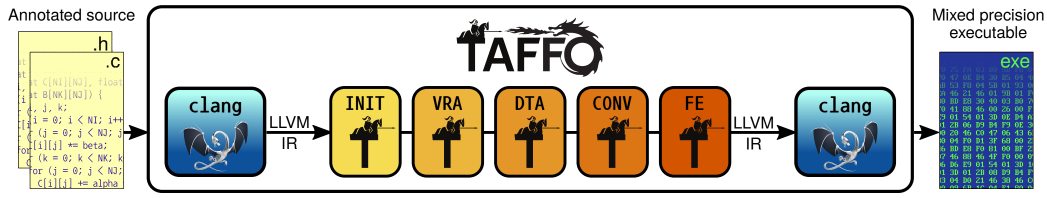


Figure 1 Architecture of TAFFO.

The borrow checker can be a severe limitation when implementing data structures and synchronisation mechanisms, which require the ability to mutate an aliased state. To overcome these limitation, Rust provides the `unsafe` keyword which allows developers to switch to unsafe Rust. Unsafe Rust is a strict superset of Rust that enables developers to perform actions that are normally prohibited, among which the invocation of `unsafe` function will be relevant to this work. It is important to understand that the `unsafe` keyword does not disable any of the other Rust safety checks, and it does not interact with the borrow checker in any way.

The combination of strict memory safety guarantees and flexibility provided by the `unsafe` code regions makes Rust a desirable target for our efforts. However, it is critical to investigate the best approach to integrate the precision tuning process within the Rust environment. In particular, it is in our interest to reuse the memory checks provided by *rustc* and do not invalidate them. The *rustc* compiler internally employs three Intermediate Representations (IRs), namely *High-Level Intermediate Representation* (HIR), *Mid-level Intermediate Representation* (MIR), and *LLVM-IR*. The borrow checker works at the MIR level. As a consequence, analyses and transformations performed at the LLVM-IR level have no visibility on the constraints imposed by it, and may in principle thwart the safety guarantees provided by the language if they do not strictly adhere to the semantics imposed by the others intermediate presentation.

2.2 TAFFO: The Compiler-based Precision Tuner

TAFFO [1] is a precision tuning that operates at the level of the intermediate representation provided by LLVM (LLVM-IR). The archiecture of TAFFO consists of five LLVM analysis and transformation passes which cooperate in building the final precision-tuned program. These passes are *Initializer*, *Value Range Analysis* (VRA), *Data Type Allocation* (DTA), *Conversion*, and *Feedback Estimator* (FE), as depicted in Figure 1. All of these passes are independent from each other, as they share data entirely through the *metadata* facilities provided in LLVM-IR.

A programmer who wishes to use this framework should first add specific annotations to the source code in order to tell TAFFO which variables must be involved in the tuning process. This task is achieved through Clang *annotations* that can be added to any variable declaration – including function arguments – as shown in the following example.

```
float x __attribute__((annotate("target('init') scalar(range(-16384, 16384) final)")));
```

Through annotations the user of TAFFO may also specify estimates for the range of values that a variable can have at runtime. This hint is required for variables whose initial definition does not depend on any other annotated variable, i.e. for *input variables*. As an example, in the annotation shown above, the “`target`” declaration gives a name to the variable and informs TAFFO about where to start collecting value range information. The “`scalar`” declaration contains information about the values the variable will assume at runtime: in particular, "`range`" specifies their range, and "`final`" also adds the information that this range is valid for the entire execution of the program. After annotating the source code, the precision tuning operation is performed simply by invoking the "`taffo`" command line tool, which is a drop-in replacement for GCC or Clang. The output will be an executable tuned for the current build machine.

Internally, the passes of TAFFO operate by first reading the annotations inserted by the user (the *Initializer* pass). Then, the *VRA* pass calculates the numerical intervals for annotated variables and other variables that depend on them. At this point, the tuned data types are selected in the *DTA* pass. At the time of writing, three approaches are available within this pass. The first one is a simple greedy algorithm that always assigns the fixed-point data type with the highest valid point position to each variable. Alternatively, the user can manually specify a floating point format which will be used for all the annotated variables. Finally, the third and most sophisticated option allows for a fully automated tuning based on precision estimation through the construction and optimization of an integer programming model of the code [2]. At the end of the pipeline, the *Conversion* pass modifies the LLVM-IR using the data types picked by the previous passes. Optionally, the *FE* pass can provide the user with an estimation of the error in the tuned program [4].

3 A Precision Tuner for Rust

The key challenge when it comes to perform precision tuning on Rust is the positioning of the precision tuning action within the compilation flow of Rust code. As anticipated in Section 2.1, the *rustc* compiler implements its memory checks on the code when it reaches the MIR stage of the compilation pipeline. It follows that precision tuning – to avoid conflicting with the *rustc* checks – should be performed either entirely before this stage, or entirely after it. This situation can be compared to the dilemma of performing precision tuning on high-level description of the code, such as on the source code, or onto a lower-level representation of the code. As mentioned in a survey on this topic [3], fine-grained precision tuning requires a lower-level representation of the code, and therefore we opt for introducing the precision tuning process entirely after the *rustc* memory checks.

On the positive side, this challenge can easily be solved by employing precision tuning components that are already built for this level of code representation, such as TAFFO. However, there are also integration issue between TAFFO and the *rustc* compiler: one in particular is the mechanism underlying the creation of the programmer annotations, as required by the TAFFO Initializer pass. For C and C++, Clang annotations are employed for this purpose, as exemplified in Section 2.2. During the generation of the LLVM-IR code, Clang simply converts each annotation to a call to the `llvm.var.annotation` intrinsic exposed by LLVM. This intrinsic function takes four parameters, representing the annotated variable, the annotation text (a string), the source file name and line number of the annotation. Notice that these annotations are a built-in extension provided by Clang, and their existence is independent from TAFFO.

Ideally, the same approach could be replicated with *rustc*. Unfortunately, this is not possible since Rust does not have an equivalent annotation syntax, hence we must design a dedicated solution. To solve this problem, we propose a Rust-native variable annotation mechanism. This annotation mechanism leverages the metaprogramming features offered by Rust *procedural macros*. The peculiarity of this kind of macros is that they can manipulate the program's token stream. We introduce the `annotate!` procedural macro, which takes two parameters: the annotated code and the annotation string. This last string supports the same syntax as TAFFO annotations for C and C++. As an example, consider the following fragment of Rust code:

```
annotate!(let mut tmp = [0_f32 ; I * J],
        "target('init') scalar(range(-16384, 16384) final)");
```

In this snippet, the `tmp` variable is associated with a value range of $[-16384, 16384]$. The procedural macro `annotate!` translates the variable declaration into the following Rust code:

```
static mut range: &'static str = "target('init') scalar(range(-16384, 16384) final)";
static mut name: &'static str = "2mm.rs";
let mut tmp = [0_f32; I * J];
unsafe {
    var_annotation(
        &mut tmp as *mut _ as *mut i8,
        &mut range as *mut _ as *mut i8,
        &mut name as *mut _ as *mut i8,
        2,
    );
};
```

The original variable declaration is preserved, while two new variables – `range` and `name` – are declared to hold the annotation string and the name of the source file respectively. Finally, the expanded body of the macro calls the `var_annotation` function with the same four parameters as the `llvm.var.annotation` LLVM intrinsic in the original Clang TAFFO interface. `var_annotation` is indeed a call to the LLVM intrinsic, which is exposed through a feature of the Rust compiler that exposes all the LLVM intrinsics as a set of foreign functions, employing the foreign function interface for C calls (FFI).

It is worth noting that the FFI call to `var_annotation` is wrapped in an `unsafe` construct. Indeed, FFI calls in Rust are considered unsafe actions in general, and must always be wrapped in an `unsafe` block guard.

3.1 Soundness of `unsafe`

Since the `annotate!` procedural macro introduces an `unsafe` region in the code for each variable involved in the precision tuning, we need to assess the impact of these regions in terms of memory safety. As mentioned before, the `unsafe` region is added because of the presence of a compiler intrinsic – that is, a function defined and handled directly by the compiler. Intrinsics are typically provided by compiler implementations to allow the use of non-standard functionalities or extensions that heavily depend on specific machine instructions to be implemented in the most optimised way. The most common application of intrinsics is for exploiting vectorisation primitives when the language does not provide a machine-independent way to access them.

In terms of memory safety, it is useful to partition intrinsics between code-generating and non-code-generating ones. Code-generating intrinsics in LLVM include, for instance, `llvm.log10.*` and `llvm.sin.*`, which generate machine-optimised code. Non-code-generating intrinsics, instead, are used only for internal purposes by the compiler, and are ignored by the code generator. Code-generating intrinsics may be memory-unsafe, especially if the code they generate involves memory operations, whereas non-code-generating intrinsics are safe, unless the compiler uses the information they carry to otherwise affect which memory accesses are made.

In our specific case, `llvm.var.annotation` belongs to the *non-code-generating* class. It is only handled in the middle-end of LLVM by keeping the annotation content tied to its variable declaration, while it is ignored by the back-end code generation. These annotations do not prevent neither enable additional transformation steps in LLVM. Thus, we can conclude that our `unsafe` block is safe for the final binary since it does not generate any code, as long as TAFFO itself is safe – which we will informally prove in the following section. Furthermore, to prevent compiler crashes, TAFFO checks each argument to confirm that it is of the correct type before it is used. If an incorrect annotation is used, the compilation emits a warning, discarding the wrong annotation. Finally, TAFFO also deletes all annotation intrinsics it is able to read at the end of its transformation, adding another layer of safety.

3.2 Soundness of TAFFO

To ensure TAFFO's transformations do not affect Rust's runtime guarantees on memory accesses, we need to demonstrate that it does not interfere with the methods used by Rust to ensure these guarantees. Memory-safety depends on the fact that memory accesses occur at addresses that fall within the bounds of allocated memory. For scalar variables, this requirement only implies forcing all accesses to happen through references to the variable – i.e., avoiding explicit pointer arithmetics. However, for array accesses, the situation is more complex, since pointer arithmetics is induced by the use of indices. The *rustc* compiler first attempts to prove at compile time that all memory accesses occur within bounds. This attempt may fail, because the values of indices may not be entirely predictable at compile-time. In this case, *rustc* introduces a guarding conditional statement before the access operation, which checks whether the index is actually within the array bounds. To ensure the safety guarantees of Rust are always mantained, TAFFO's handling of memory allocations was modified in order to disable the following operations:

1. Increasing the size of the elements of an array.
2. Changing the size of a dynamic memory allocation.

With the introduction of these two additional constraints, we can ensure that TAFFO never modifies Rust's code in a way that violates memory access bound checks.

In our informal proof we are going to consider both the compile-time and the runtime scenarios. In the first scenario, *rustc* generates an LLVM-IR equivalent to the one generated by Clang for the equivalent C code. More in detail, the generated code consists of a sequence of two instructions: a `getelementptr` instruction which computes the correct address of an element given a base pointer and an index, and a `store` or `load` instruction that actually performs the access at that address. When TAFFO needs to generate a new `getelementptr` instruction to index a fixed-point array, it copies the original `getelementptr` used to index the corresponding float arrays, and modifies only the returned datatype. The returned datatype, however, must be consistent with the one employed at the time of buffer allocation. As a result TAFFO must also modify the buffer allocation code, whose location is unfortunately not always detectable nor modifiable at LLVM-IR level. In case the buffer allocation cannot be modified, TAFFO mantains safety by only allowing datatypes that are smaller than the original one, therefore employing only part of the allocated buffer and preventing any buffer overflow. In the second scenario, Rust introduces a guarding conditional by means of a branch in LLVM-IR. The branch employs a condition obtained by comparing integer data, which are not modified by TAFFO and therefore reflect the array element size employed in the Rust code before precision tuning. In order to maintain the validity of the guard condition at all times, TAFFO *must not* modify the size of the memory allocation being guarded. With this additional constraint alone, the Rust code will keep its memory-safety properties. However when TAFFO increases the element size the program will stop working, as array accesses to correct indices in the non-tuned program become locations outside of the buffer bounds. Therefore the first condition (not changing the size of the array elements) needs to be applied for runtime-checked arrays as well. Clearly, the goal of reducing the precision of a program is better achieved with *smaller* data types rather than with *larger* ones. As a result we expect a minimal impact to Precision Tuning from these restrictions.

4 Experimental Evaluation

A key feature of Rust is its high performance, which enables its use as a C or C++ replacement in embedded systems. The ability to combine the guarantees provided by this language with the even greater speedups obtainable through precision tuning makes the combination of TAFFO and Rust the ideal platform for development of high-performance applications. In this section, we perform an experimental assessment of the impact of TAFFO on the Rust compiled code, employing a new port of the PolyBench benchmark suite to Rust.

4.1 Experimental Setup

All the experiments were performed on an STM32F207ZG microcontroller chip that features an ARM® Cortex®-M3 32-bit RISC core operating at 120 MHz. This chip includes a region of Flash memory for code of 1 Mbyte, and 128 KBytes of working RAM. The ARM Cortex-M3 core does not have a hardware floating-point unit, hence all floating-point operations are executed in software emulation. We generated the board setup code using STMCubeMX, compiled it with Clang, and linked it with each benchmark as a separate compile unit for both C and Rust.

For C benchmarks, the Precision Tuning process was done through the unmodified "`taffo`" command line tool. Instead, for Rust benchmarks, we modified the "`taffo`" tool to accept Rust code. In order to introduce Rust within the TAFFO compilation pipeline, invocations to Clang were replaced with invocations to *rustc* used as a front-end to generate LLVM-IR code. The rest of the pipeline is identical to the one employed for C code. This method ensures that both C and Rust codes are processed in the same way, applying the same optimization levels in the middle- and back-end. Additionally, we take into account that *rustc* is able to perform some optimizations at the internal IR level (MIR, discussed previously in Section 2.1), by passing the option "`-Z mir-opt-level`" to the *rustc* frontend. This is not the case and is not required for Clang. The baseline non-tuned benchmarks were compiled using the *rustc* and Clang tools. The version of LLVM and Clang employed was 15.0.7, while the version of *rustc* was 1.64.0. The optimization level used in `opt` for Rust and C was `-O3`.

4.2 Polybench/Rust

For our experimental work we selected the PolyBench suite Version 4.2.1, as its intended purpose is indeed to evaluate novel compiler optimizations, and as such it has been widely adopted in the literature. It consists of several numerically-intensive kernels which natively rely on floating point arithmetics, making it a good target for Precision Tuning. Moreover, its C implementation is already supported by TAFFO [1], which we will employ as-is.

For our comparison, we must employ a port of PolyBench to Rust which is as similar as possible to the baseline C code, in order to reduce any confounding factor determined by implementation differences between the benchmarks. To this end we evaluated the existing rewrite of PolyBench in Rust, PolyBench-rs[2]. However, after careful consideration

[2] `https://github.com/JRF63/polybench-rs`

we deemed this port not suitable for a direct comparison with PolyBench/C, as it heavily modifies the original benchmarks by adopting a functional programming style. While the choice is legitimate per-se, analysis of the resulting LLVM-IR code evidenced several non-functional differences with PolyBench/C, which result in very different execution time and memory consumption patterns between the two. For this reason, we developed a new port of PolyBench/C, named PolyBench/Rust, which aims to be as close as possible to PolyBench/C while still avoiding any *unsafe* block. In all the following experimentations, we have used the SMALL_DATASET as the array size. A comparison between the baseline PolyBench/C and PolyBench/Rust is provided in Figure 2.

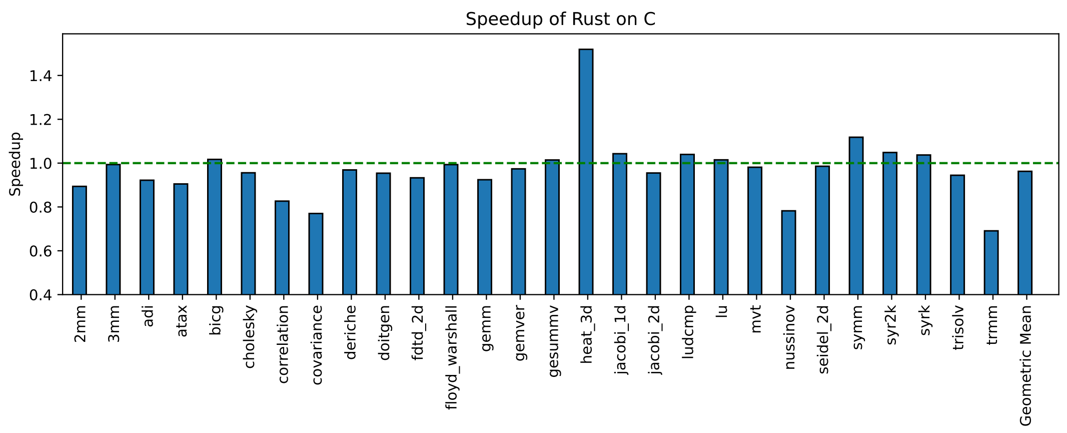


Figure 2 The ratio of the execution time between PolyBench/Rust and PolyBench/C as a speedup. The behavior of the two benchmarks is very similar, with exceptions for *heat_3d*, *correlation*, *covariance*, *nussinov* and *trmm*. The geometric means of the speedup of all benchmarks is represented in the last entry under "Geometric Mean".

Indeed, we get very similar execution times to PolyBench/C. In 15 benchmarks out of 28 the speedup is between 0.9 and 1, and in 8 benchmarks the speedup is between 1 and 1.1. One benchmark has a speedup greater than 1.1, *heat_3d*, while 4 benchmarks have a speedup lesser than 0.9 (*covariance*, *correlation*, *nussinov* and *trmm*). Overall, we notice PolyBench/Rust is slightly slower than PolyBench/C, due to the additional runtime checks introduced by the language. For the specific case of *heat_3d*, the *rustc* compiler applies – at the LLVM-IR level – an aggressive loop unrolling strategy enabled by better alias disambiguation offered by Rust. Speedups lower than 0.9 are instead caused by a combination of a greater amount of runtime checks and less efficient code generation from *rustc*. While for *trisolv* and *trmm* the main cause of the slowdown are the bound checks, for *correlation* and *covariance* the main cause is a greater number of memory accesses around floating point emulation calls that could be optimized away by Clang but not by *rustc*. We expect most of these differences to be smoothed over and disappear as *rustc* matures as a compiler.

4.3 Experimental Results

Figure 3 displays the speedups of each benchmark in PolyBench/C and PolyBench/Rust achieved through the Precision Tuning process of TAFFO (blue and orange bars). The baseline is always the same benchmark, for the same language (Rust or C), but compiled without TAFFO. After Precision Tuning, PolyBench/Rust is faster than PolyBench/C in 9

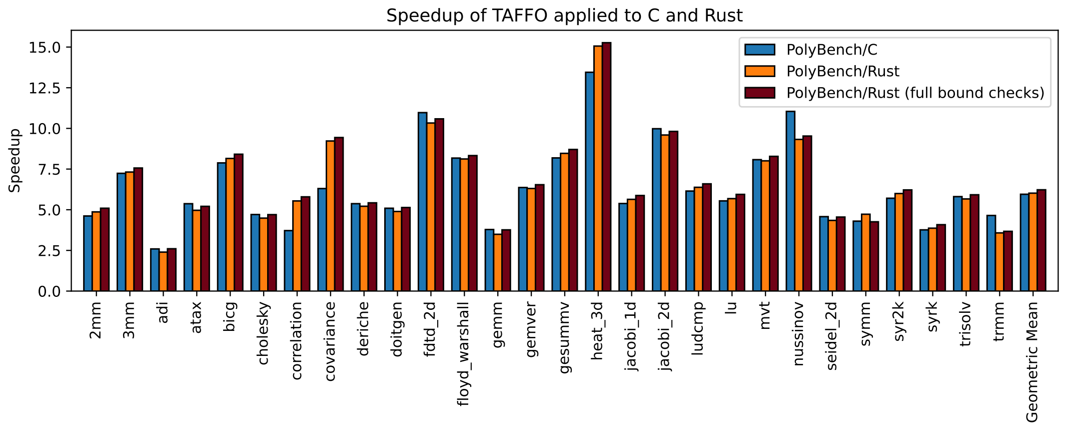


Figure 3 Comparison of the speedup achieved by TAFFO on PolyBench/C and PolyBench/Rust. Rust results are also shown with additional bound checking. The geometric means of the speedup of all benchmarks grouped by their configuration is represented in the last entry under "Geometric Mean".

out of 28 benchmarks, but aside from a few outliers, TAFFO performs in a similar way in both languages. The speedup for Rust ranges between 2.3 and 15.0, while the speedup for C ranges between 2.5 and 13.4.

Looking at individual benchmarks, *covariance* and *correlation* get a much higher speedup in Rust than in C. This is due to the aforementioned fact that *rustc* sometimes generates suboptimal code around calls to floating point emulation functions. After the Precision Tuning passes performed by TAFFO, the floating point emulation code gets replaced with integer fixed-point primitives, lessening the register allocation constraints for the backend, which then manages to eliminate redundant memory accesses. Benchmark *heat_3d* is another example where Rust code achieves a higher speedup with respect to the C one. In this case, the speedup is due to constant propagation optimisations enabled by the reordering of instructions performed by *rustc*, which further improves with the aggressive loop unrolling mentioned in Section 4.2 and is amplified by the TAFFO precision tuning transformations.

Overall, the difference in the speedup introduced by TAFFO on Rust and C is mainly due to the fact that *rustc* generates code that more closely represents the actual data flow within the program. While the PolyBench/C code relies on re-using existing arrays for storing intermediate computation results, Rust is able to remove these superfluous array accesses by using temporaries instead. Indeed, Rust guarantees that accessing memory through a mutable reference cannot be aliased, hence the compiler can prove that the re-use of an array in such a way does not have side-effects. This transformation makes the overall data flow more visible to the analyses and transformations of TAFFO (especially the Value Range Analysis), allowing for more performant code in the end.

Since most Polybench kernels are written to work with fixed-size arrays, only a few require extensive bound checks in the code for ensuring safety. Real-world applications, though, may exhibit more arrays with size dependent on input data, thus leading *rustc* to be less effective when optimizing bound checks than it would appear from benchmarking. In order to show how TAFFO is affected when the code contains more extensive bound checks, in Figure 3 we also show the speedups obtained when disabling bound check optimization (orange bars). The speedups are extremely similar to the ones obtained with the bound check optimization enabled, so we can state that TAFFO is not significantly affected by it.

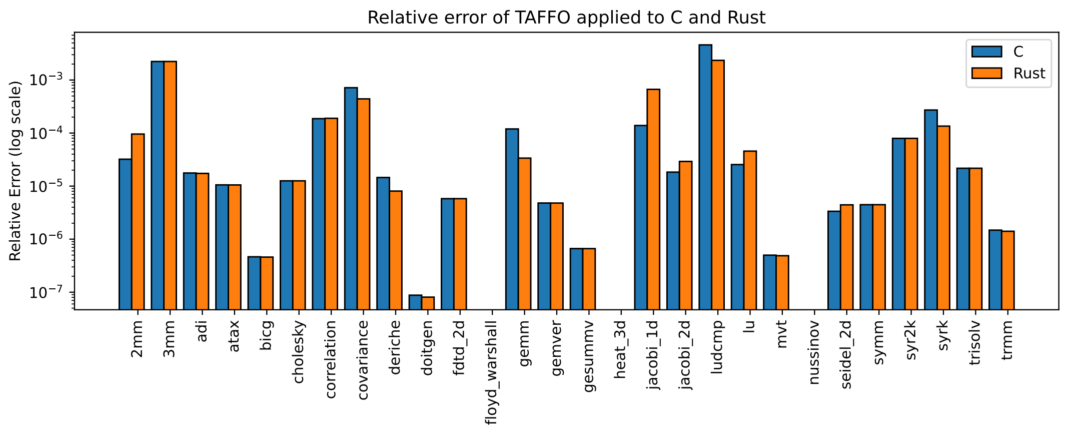


Figure 4 The graph displays the relative error of TAFFO C and TAFFO Rust as compared to the C version. The plain C version and the plain Rust version generate identical values.

Finally, we compute the relative errors resulting from the Precision Tuning approximation for both programming languages, which are shown in Figure 4. These figures do not depend on the amount of bound checks performed by the code. First of all, we observe that in most cases the relative error introduced is the same both for Rust benchmarks and C ones. The largest relative error was found to be approximately 0.45%, which occurred in the case of *ludcmp* of PolyBench/C. We can conclude that TAFFO introduces errors of similar magnitude regardless of the source language.

5 Conclusions

We provide the first automated Precision Tuning framework supporting the Rust programming language. We employ TAFFO to provide the analyses and transformation required for Precision Tuning, and we integrate it with the Rust compiler *rustc* by introducing the appropriate macros that allow for passing the programmer annotations required by TAFFO from Rust. As these macros use the unsafe features of Rust, we demonstrate that they do not actually affect the actual memory-safety of the code. Then, we evaluate our approach by developing and annotating a Rust version of the PolyBench suite for use with TAFFO. We see that Precision Tuning in Rust is just as effective as Precision Tuning in C, within a margin of error that can be fully attributed to differences in the compiler front-end, and further, Precision Tuning can help in offsetting the penalties imposed by the introduction of runtime checks. Rust is just one of the several memory-safe programming languages available for use in embedded systems, therefore a natural extension of this work includes support for languages such as Go and Swift. A further research direction is the design and implementation of Rust language extensions that allow the integration of precision tuning without the need for potentially-unsafe macros for generating compiler intrinsics.

References

1 Daniele Cattaneo, Michele Chiari, Giovanni Agosta, and Stefano Cherubin. TAFFO: The compiler-based precision tuner. *SoftwareX*, 20:101238, 2022. doi:10.1016/j.softx.2022.101238.

2 Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. Architecture-aware precision tuning with multiple number representation systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 673–678, 2021. doi:10.1109/DAC18074.2021.9586303.

3 Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation: a survey. *ACM Computing Surveys*, 53(2), April 2020. `doi:10.1145/3381039`.

4 Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Agosta Giovanni. Dynamic precision autotuning with TAFFO. *ACM Transaction on Architecture and Code Optimization*, 17(2), May 2020. `doi:10.1145/3388785`.

5 David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, October 1998. `doi:10.1145/286942.286947`.

6 MITRE Corporation. The common weakness enumeration (CWE) initiative, 2021. URL: `http://cwe.mitre.org/`.

7 Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo M.K. Martin, and Steve Zdancewic. Ironclad C++: A library-augmented type-safe subset of C++. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &amp; Applications*, OOPSLA '13, pages 287–304, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2509136.2509550`.

8 Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 597–616. USENIX Association, August 2021. URL: `https://www.usenix.org/conference/soups2021/presentation/fulton`.

9 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, page 75. IEEE Computer Society, 2004.

10 National Security Agency (NSA). The national security agency (nsa) cybersecurity information sheet, 2022. URL: `https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF`.

11 StackOverflow. 2023 developer survey. `https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages`. Accessed: 2023-12-01.

12 Phillip Stanley-Marbell, Armin Alaghi, Michael Carbin, Eva Darulova, Lara Dolecek, Andreas Gerstlauer, Ghayoor Gillani, Djordje Jevdjic, Thierry Moreau, Mattia Cacciotti, Alexandros Daglis, Natalie Enright Jerger, Babak Falsafi, Sasa Misailovic, Adrian Sampson, and Damien Zufferey. Exploiting errors for efficiency. *ACM Computing Surveys*, 53:1–39, July 2020. `doi:10.1145/3394898`.

13 Jesse A. Tov and Riccardo Pucella. Practical affine types. *SIGPLAN Not.*, 46(1):447–458, January 2011. `doi:10.1145/1925844.1926436`.

Embedded Multi-Core Code Generation with Cross-Layer Parallelization

Oliver Oey
Karlsruhe Institute of Technology, Germany
emmtrix Technologies GmbH, Karlsruhe, Germany

Michael Huebner
BTU Cottbus - Senftenberg, Germany

Timo Stripf
emmtrix Technologies GmbH, Karlsruhe, Germany

Juergen Becker
Karlsruhe Institute of Technology, Germany

Abstract

In this paper, we present a method for optimizing C code for embedded multi-core systems using cross-layer parallelization. The method has two phases. The first is to develop the algorithm without any optimization for the target platform. Then, the second step is to optimize and parallelize the code across four defined layers which are the algorithm, code, task, and data layers, for efficient execution on the target hardware. Each layer is focused on selected hardware characteristics. By using an iterative approach, individual kernels and composite algorithms can be very well adapted to execution on the hardware without further adaptation of the algorithm itself. The realization of this cross-layer parallelization consists of algorithm recognition, code transformations, task distribution, and insertion of synchronization and communication statements. The method is evaluated first on a common kernel and then on a sample image processing algorithm to showcase the benefits of the approach. Compared to other methods that only rely on two or three of these layers, 20 to 30 % of additional performance gain can be achieved.

2012 ACM Subject Classification Software and its engineering → Source code generation; Software and its engineering → Imperative languages; Software and its engineering → Very high level languages; Computer systems organization → Embedded software

Keywords and phrases Parallelization, multi-core Processors, model-based Development, Code Generation

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2024.5

1 Introduction

State-of-the-art embedded multi-core processors offer high performance with low power consumption, but programming them efficiently presents new challenges due to a high complexity e.g. in the partitioning of tasks on the specific multi-cores. Some of the main challenges when developing applications for embedded multi-core are:

- Developers tend to think sequentially. To take full advantage of multi-core systems, parallel applications need to be partitioned up front. This extra work does not occur in sequential development and distracts from the actual programming of the algorithm.
- To distribute tasks across processing units, data and control dependencies have to be considered to avoid errors like race conditions or deadlocks which cannot occur in sequential programs.
- Debugging is much more complex on multi-core systems because parallel processing with multiple threads does not ensure determinism. If execution is interrupted at any time, the current state of each processing unit cannot be predicted.

Programmers encounter challenges that divert their focus from implementing actual functionality. This paper proposes a solution that segregates application development from target platform optimization. The proposed solution involves commencing with a model-based design that remains entirely platform-agnostic, followed by iterative optimization grounded in four defined abstraction layers. Each layer progresses towards the hardware in granular stages. This approach ensures that algorithm development is completely decoupled from hardware platform optimization, as no code needs to be written after model implementation. The approach is intended to be used with applications that can be analyzed statically which means problem/data sizes are known or at least bounded so that the best performance of the application on the target platform can be achieved. There are various possibilities to implement the approach, including using multiple software tools within a tool flow for layer optimizations or employing manual optimization steps that concentrate on the defined layers only. For the remainder of this paper, we adopted a middle approach: using currently available tools for optimization at each layer, but without a fully integrated tool flow. We used tools developed by emmtrix Technologies[1] that were originally based on results from the ALMA research project [2]. While these existing tools provide the framework to apply the optimizations on different abstraction layers, the concept of this proposed cross-layer parallelization is not integrated in the usual tool flow. This novel combination of layers allows the use of specifically optimized implementations for identified algorithms, in addition to general optimizations such as code transformations and task distribution with optimized communication placement. The following four abstraction layers will be used in this paper:

1.1 Definition of Algorithm Layer

The algorithm layer in this approach refers to the abstract representation of an algorithm that has been developed independently of the target hardware platform. The choice and implementation of the algorithm has a high impact on performance. Let's take sorting algorithms as an example: different algorithms vary in terms of runtime, memory requirements, and stability. The potential for parallel execution is different with each implementation but usually comes with a slower sequential execution or increased memory requirements. The best selection can therefore only be made in combination with the information about how many processing elements and how much memory is available and how big the data set to process is. The goal of this layer is to provide a library of common algorithms or kernels with a selection of implementations to choose from, to identify these known algorithms in the source code and together with the decisions on later layers select the best performing realization for the actual hardware.

1.2 Definition of Code Layer

The code layer refers to the actual representation of the algorithm as source code. How the individual computations are represented determines how many (independent) tasks can be generated and thus how well parallelization can work. Code transformations can be used to change the code of the application without affecting the result of its calculations. They can be used to take advantage of the intrinsic parallelism that is already part of the code, for example, when processing large amounts of data. Two example transformation that are used in the evaluation:

[1] `https://www.emmtrix.com`

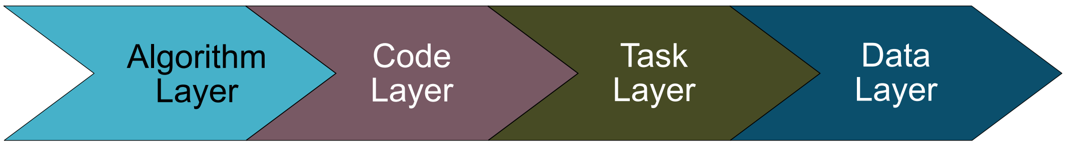


Figure 1 Abstraction layers used in this approach.

- Loop fission: If there are multiple statements in the body of a loop that can be executed independently, it is possible to split the statements into two or more loops over the same index range. Since the resulting loops have no dependencies on each other, they can be executed in parallel.
- Variable splitting: Instead of having a single array variable, splitting it into multiple ones allows parallel calculations on different regions of the original data without affecting each other.

The most beneficial transformations are applied to loops as that has the greatest potential for improving parallelism. But besides the loops and their number of iterations, other properties play a role at this layer as well, e.g. data access or locality, the actual number of individual code blocks and how well the code can be statically analyzed.

1.3 Definition of Task Layer

The task layer refers to the part of this approach at which tasks are assigned to individual execution units of the target platform. During parallelization, both mapping and scheduling are performed to select which core or processor a task is executed on and the order in which the tasks are executed there. All dependencies between tasks must be taken into account, mainly from the data flow, but also from the control flow. Depending on the target architecture, these dependencies lead to synchronization or communication overhead. In addition to pure data dependencies, anti and output dependencies must be considered to ensure the correct order of execution. Also important for parallelization decisions is the execution time of each task relative to each other and relative to the synchronization overhead. If the granularity of the tasks is not taken into account, the overhead of communication and synchronization may outweigh the benefit of parallel execution, resulting in a slowdown compared to the sequential program.

1.4 Definition of Data Layer

The data layer in this paper refers to synchronization and data exchange between cores. Without loss of generality, this work assumes hardware models with distributed memory. Any shared memory system can also be considered and treated as a distributed memory system by hard allocating the available memory to the cores. The goal of optimization on this layer is to ensure data availability on each core while achieving the best runtime on the hardware and ensuring that no errors are introduced due to false synchronization. Important aspects are the placement of synchronization instructions and keeping the overhead of transfers to a minimum.

Figure 1 shows all used layers and their typical usage from the input source code to the code for the platform. The remainder of this paper is organized as follows: Section 2 discusses the current state of the art and how the four layers are usually employed, Section 3 elaborates on the actual cross-layer optimization, Section 4 evaluates the approach with an experimental case study, and Section 5 concludes this paper with an outlook.

2 State of the Art

Foster[7] formulated already in 1995 a workflow to design and build parallel programs. The main steps were partitioning, communication, agglomeration and finally mapping of tasks and are taught at universities as a standard approach for parallelization. The approach is comparable with the last three layers of our approach but lacks the potential of special optimizations of known algorithms which allow more specialized implementations of known kernels that usually make up large parts of typical embedded applications.

In [5], methods for programming parallel platforms based on algorithmic skeletons and parallel design patterns are evaluated. While they prove useful for parallel programming, they don't ease the required hardware knowledge from the programmer. This kind of library-based approach mostly focuses on the algorithm layer to provide special parallel implementations with known functionality. Without any considerations from the task and data layers, this kind of parallelization is limited to these functions and resources respectively an efficient load balancing throughout the execution of the whole program cannot be achieved.

Automatic parallelization by the compiler is still a research topic in works like [9] and [8]. They focus on pattern recognition of common programming patterns with more recent work also on the usage of machine learning for the detection. This pattern recognition covers the optimizations on the algorithm layer and should achieve good results for programs that make great use of these patterns. However, programs usually also have parts that don't fit into patterns and might need some other optimization steps which could e.g. be applied on the code layer. [1] uses source-to-source compilation to optimize the source code for automatic parallelization using OpenMP. Together, this covers the code, task and data layers as defined in this work but does not use the benefits that optimizations on the algorithm layer could bring to known algorithms.

The Daedalus framework [14] is a more recent approach for the design of multi-processor system-on-chips. By approaching the issue together with system level synthesis, a synchronous development of software and hardware is performed. To optimize the software for parallel execution, the tool PNgen is used to apply polyhedral optimization techniques which result in loop transformations according to mathematical equations. This approach together with the design space exploration covers the two middle layers of our approach: the code and the task layer. By adding the algorithm layer, specific optimizations can improve the results while the data layer allows for more efficient execution of the parallel program on the target platforms.

A summary of popular parallelization techniques can be found in [13]. Comparing it to this approach it shows that they mostly focus on the code and data layers by optimizing loops and communication.

[12] shows how concepts from high performance computing (HPC) can be applied to embedded systems. HPC is originally more focused on achieving the best performance with the available hardware without taking the potential worst case into account. While this is not the case for typical embedded applications, the need to get the best performance out of multi or many-core processors is getting higher. The work described relies on OpenMP with task scheduling and does not take optimizations on algorithm or data layer into account.

Looking at heterogeneous systems, [11] compares different programming frameworks like OpenMP, OpenCL and CUDA regarding programming productivity, performance and energy. One major result here is that the human factor significantly impacts the fraction of lines of code used to parallelize the code. Reducing this human impact should therefore be the goal. The methodology proposed in this paper covers this aspect by only letting the programmer develop the model and no further changes to source code.

[16] introduces a compiler-based optimization specialized for machine learning. The compilation applies various parallelization techniques controlled by the user through source code annotations. This approach is sophisticated in the sense that the user does not necessarily need in-depth hardware knowledge. All four layers are addressed, with reference to the algorithm layer done through the limited applications that mainly relies on library calls. The code and task layers, along with the data layer, are automatically processed, utilizing user annotations as guidance for specific components.

Examining techniques that utilize multiple abstraction layers, studies such as [10] expand the LLVM intermediate representation using domain-specific languages (DSL) that can reuse the same compiler passes across numerous levels of abstraction. Although the initial aim was to use transformations to lower abstraction, research such as [4] can be utilized to raise the abstraction level and back-propagate information to higher levels. While this approach optimizes all abstraction layers used in this work iteratively, it lacks flexibility compared to the model-based approach used here due to its focus on DSLs.

Functional development is decoupled from optimization for the target platform in studies such as [15]. Functional development is conducted in the high-level general-purpose programming language Python 3 and the mapping process is carried out in Artisan meta-programs. This decoupling allows different experts to handle these developments, eliminating the need for understanding both the algorithm and the optimal implementation for the heterogeneous hardware. Compared to our approach, the presented work relies on downstream processing by OpenMP or high-level synthesis for optimal performance. Our work, on the other hand, optimizes program scheduling and data transfer directly on the task and data layers, eliminating the need for downstream tools or operating systems.

3 Cross-layer Optimization

While independent execution of the layers is possible, the combined approach provides distinct advantages. Progressing through the layers reduces the level of abstraction from the hardware at each stage, necessitating more information with each subsequent layer. When switching to a different hardware platform that has at least some similarity with the previous one, e.g. the same numer of processing elements, some optimizations can be re-used. To demonstrate the optimizations achieved at each layer and their potential interactions, we will employ the Fast Fourier Transform (FFT) as an exemplar kernel. It was selected as it is a well-known algorithm and shows good optimization potential on each defined layer.

3.1 Realization of Algorithm Layer

Recognition of known algorithms cannot simply be achieved by extracting them from source code, as illustrated by the halting problem, where determining a specific behavior for generic programs is impossible. Our solution is to use MATLAB® as the programming language for a model-based approach. It offers several built-in versions of common algorithms, with clear descriptions of the results. So instead of attempting to identify established algorithms, the specific algorithm can be identified through a straightforward function call and precise behavior specification within the MATLAB® description. To transfer the algorithm to the embedded device, it was converted from MATLAB® to C code, as this enables more targeted optimizations customized for the designated hardware. The code generation tool developed includes support for basic arithmetic operations. For example, when adding two arrays, the tool will convert them to arrays in C, which are then summed up with for-loops. Library functions are utilized to customize function conversions, implementing the requisite functionality in MATLAB® scripts, subsequently generating the pertinent C code. Data type and range analysis ensure the employment of only the most efficient data types in the C code. Furthermore, this approach can offer unique realizations of established functions that can be chosen with custom pragmas within the scripts. This permits additional enhancements of the produced C code by returning to the code generation phase from later stages. To achieve efficient implementation on the embedded target platform, a conversion of code must be undertaken due to the inefficiency of executing MATLAB® scripts. Utilizing C code on embedded platforms provides greater convenience and enables precise control of hardware resources. The conversion step also facilitates the selection of an algorithm realization and the generation of C code optimized for later layer simplifications.

The algorithm layer only uses the number of available cores as information about the target platform. This information can be used to select an implementation that is optimized for that number of cores.

Now, looking at the FFT, we can see that it is an in-built function of MATLAB® that has a clear specification about its behavior and accuracy. Knowledge about the algorithm allows us to provide an option for later layers of the flow: a N-point FFT may be substituted with two $N/2$-point FFTs, which can be computed independently, as illustrated in Figure 2. In other words, a simple option for the code generation can be used to specify the number of times the FFT should be split up into. With the additional information about the number of processing cores available, this can also be used as an upper boundary as splitting the FFT up into more parts is usually not useful.

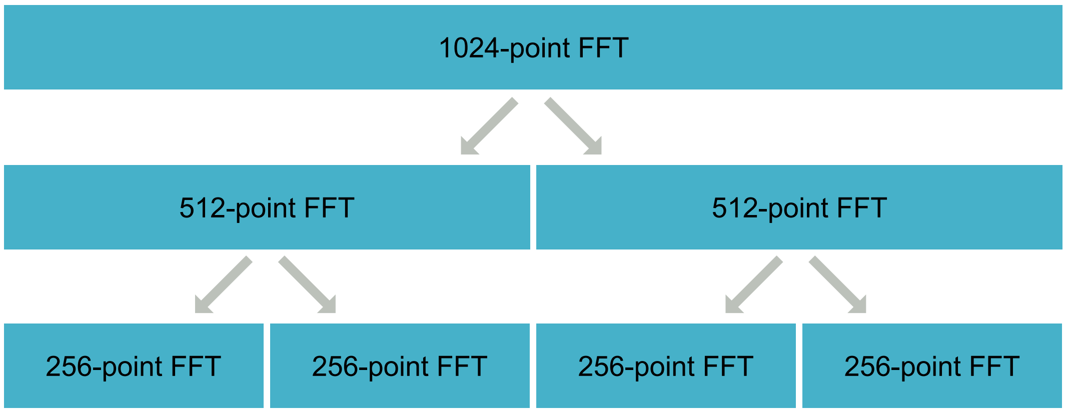


Figure 2 Splitting FFTs.

3.2 Realization of Code Layer

The code layer focuses on transforming the C code for task level parallelization. It is implemented as a source-to-source compiler that reads the input C code, applies selectable code or loop transformations to the desired parts of the code, and then outputs the transformed C code. While these steps are not new in themselves, the importance lies in their interaction with the other layers: Transformations in the code layer rely heavily on input from the algorithm layer. E.g. an increased number of loops increases the potential for beneficial loop transformations. The granularity of the transformations ensures that independent tasks can be generated very well at this point for parallelization at the task layer. However, the number of tasks should be determined together, since too many interdependent tasks only increase the complexity of the scheduling algorithm without offering more optimization potential. All transformations that can change the access to the data also have the potential to reduce the overhead at the data layer. At this layer, more information about the hardware can be used for all decisions. Besides the actual number of processing elements, the type is relevant to address available accelerators. Optimizations that improve data locality require information about memory layout and cache availability to be useful.

For the FFT example, code transformations may be applied to the main loops of the kernel by using variable splitting and loop fission to generate independent loops. Considering the splitting option at the algorithm layer, this provides several options for dividing the code execution. However, the best options will be selected with the assistance of the next layers.

3.3 Realization of Task Layer

After the enabling optimizations on the previous layers, the main part of the coarse-grain parallelization takes place on this layer. As a pure optimization algorithm on a graph representation, it takes into account all dependencies that are crucial for the allocation to the individual cores. This is important for maintaining correctness, but sometimes overestimates the actual performance. Since the placement and optimization of the communication takes place later at the data layer and is based on a different representation, the synchronization and duplication of the data cannot yet be fully considered and can only be passed as hints to the next layer.

The most important function of this layer is to identify parts of the code that can be executed independently by analyzing the data and control dependencies, and then to assign these extracted tasks to the available cores so that the overall runtime is minimized. This requires information about the actual runtime on each core to model the optimization problem and the cost of the overhead of parallelization. The actual parallelization at this layer of abstraction can be done with many different optimization algorithms. In this work, the well-known Heterogeneous Earliest Finish Time (HEFT)[8] is used. As a greedy heuristic algorithm, it usually does not reach the optimal runtime, but it has been extended to respect user constraints like fixing tasks on certain cores in order to guide the optimization process in the right direction. The parallelization on the task layer distributes all tasks onto the available cores taking into account the communication overhead required to synchronize the data. To minimize this overhead and ensure the correct execution, the actual placement and optimization of the communication is handled on the next layer.

For the FFT example, partitioning of the code on both the algorithm and code layers is decided at the task layer. It is only through the utilization of a more precise task runtime cost model on the actual hardware and overhead costs for parallel execution that the overhead of the two approaches can be accurately calculated. A performed test on an ARM Cortex A-76 processor with four cores demonstrated that dividing the FFT by four in the algorithm layer gets the best use of the available cores. Code transformation are useful on the parts where the data is partitioned, but as more involved also increases the required data transfers, an optimization for two cores on the code layer achieves the best runtime.

3.4 Realization of Data Layer

The main function of this layer is to resolve all data dependencies that cross core boundaries by inserting send and receive instructions for data transfers. Done correctly, this ensures that the two most common errors in parallelization cannot occur: race conditions and deadlocks. Race conditions occur when the result of one operation depends on the timing of another operation. This can often happen when shared resources are accessed by multiple cores at the same time. By modeling these accesses through data dependencies, synchronization instructions can be inserted whenever the resource is accessed, enforcing a predetermined order across all cores that need to access the resource. This ensures the absence of race conditions with the added overhead of synchronization whenever resources are shared. A deadlock is an error state that occurs when the program waits indefinitely for data or signals that never arrive. The program is stuck and cannot continue its execution. One way to avoid this is to duplicate the control flow for all cores involved. That is, if a signal is sent within an if-block only when a certain condition is met, the receiving core executes a copy of the if-block that evaluates the same condition. This guarantees that send and receive are always executed under the same conditions and that a receive is never called without an associated send. The data layer requires the most details about the target platform. It needs information about the communication infrastructure with all the data buffers, access to the interface code, and information about what the C code should ideally look like for the target platform's compiler.

For the FFT example, the data layer is crucial for achieving the expected performance as calculated at the task layer. To minimize core wait times, it is necessary to have a better understanding of the actual interconnections between the cores and to manage data transfers efficiently. This depends heavily on the volume of data being processed. As FFT is typically utilized for larger amounts of data and requires minimal synchronization between the cores, achieving high throughput is critical for optimizing the runtime on the hardware.

3.5 Interactions Between the Layers

With each layer, the optimization takes into account additional information about the hardware specification. Figure 3 depicts which main features are utilized by each layer. The arrows at the bottom of the figure show the typical process of cross-layer optimization: the order of the layers is the algorithm, code, task, and data layer. From the code layer to the algorithm layer, transitions occur when there is a need to implement a modified algorithm or prepare for a specific number of cores. Likewise, the task and data layers transition to the code layer when the code's representation makes it difficult to efficiently allocate tasks among the cores, either due to the volume of tasks or the resulting communication overhead. From the data layer back to the task, reevaluation is necessary when scheduling proposes parallelization that would create excessive communication overhead in the data layer. Considering a change made on a previous layer would be advantageous, going back more than one step is also an option.

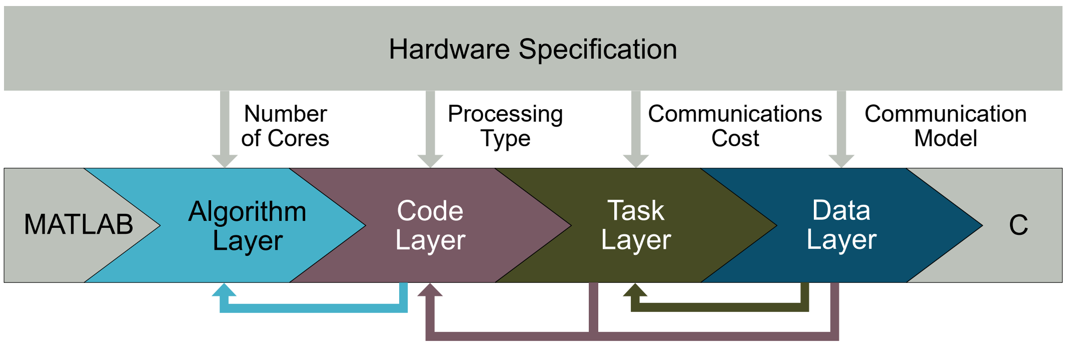


Figure 3 Interactions between the different layers.

4 Evaluation of the Method

For the evaluation, a streak detection algorithm as utilized for space debris detection is used. It was selected as it best shows the potential of the approach on all the layers. The algorithm consists of two main steps:

1. The input image is converted to grayscale and a Canny edge detector is used to extract all edges in the image. The Canny edge detector is a well known algorithm [3] that is still quite popular due to its low error rate and high accuracy. It is applied on an image that is usually denoised with a Gaussian blur filter to find the edges identified with the highest gradient change.
2. A Hough transform is applied to the resulting image, which is used to detect the peaks with the highest contrast and to mark the found lines in the image. The Hough transform is a formerly patented method [6] that is used to find geometric shapes such as straight lines in binary black and white representations.

The source code of the algorithm is shown in Listing 1. The most relevant parts of the program are directly available in MATLAB® so that calling the inbuilt functions is sufficient. Only for the function to extract a line, a new custom function needed to be developed.

Listing 1 MATLAB® code of streak detection.

```
function [point1Arrays,point2Arrays] = streak_detection(img)
  %convert to grayscale, available in MATLAB
  gray = rgb2gray(img);
  %extract edges from image, available in MATLAB
  [E,thresh] = edge(double(gray),'Canny');
  %apply Hough transform, available in MATLAB
  [H,T,R] = hough(E);
  %extract peaks in Hough representaiton, available in MATLAB
  P = houghpeaks(H, 50);
  %draw a line to connect points, implemented manually
  [point1Arrays,point2Arrays] = custom_houghlines(E,T,R,P);
end
```

The target system is an NXP P4080 DS with 8 PowerPC e500mc cores.

4.1 Evaluation of Algorithm Layer

Since the Canny edge detector algorithm is directly available in MATLAB® a call to the function can be used as an entry point for the optimization at the algorithm layer. This offers the potential for a few different optimizations: fixing the threshold for the edge detection maintains the same results while reducing the runtime by 6%. Switching to single precision data types with 32-bit floats still has enough accuracy, but reduces the runtime by another 28%. Finally, optimizing the memory allocation allows a further optimization of 19%, for a total gain of 45% on the algorithm layer alone. The optimizations were applied without any specific hardware knowledge, and the reduced complexity provides more options in later layers of the flow. The performance gains were determined on the development PC using an Intel Core i5-4570 with 16 GB DDR3 RAM with 1600 MHz. Measured was the execution time of the C program generated from the MATLAB® code and the initial 259 ms were reduced to 140 ms.

Most functions in the source code 1 are part of the MATLAB® language scope, so that specially optimized versions can be used during code generation. By using parameters to guide the code generation, the generated C code can be prepared for later transformations. In order to be able to apply the beneficial variable splitting and loop fission transformations on the functions `rgb2gray`, `edge` and `hough`, the number of generated loops and variables can be changed according to the input from the code layer.

4.2 Evaluation of Code Layer

The main task of the code layer is to provide the task layer with as many independently computable tasks as possible. This is best achieved from the combination of variable splitting and loop fission. The C code prepared at the algorithm layer now allows easy exploration of the various options. The best results are achieved when the load between cores is balanced over time with as little data exchange as possible. For the `hypot`, `hough`, and `rgb2gray` functions, which each perform only a few calculations on a single matrix, the best solution is to split them among eight, i.e., all available cores of the target system. For the Canny Edge algorithm, which is executed independently in the X and Y directions, splitting for four cores at a time has been shown to give the best results. By executing the two directions in parallel, all eight cores can be utilized and limiting them to four cores each reduces the communication overhead. While the actual splitting of the variables and loops is done on this code layer, the decision about the splitting factors is decided by the later task and data layers.

4.3 Evaluation of Task Layer

The previously generated independent tasks ensure that parallelization at the task layer can be solved largely automatically. Only for tasks that split or collect data is there room for optimization in the allocation by selecting cores in a way that minimizes communication. This is done in conjunction with the data layer, since it is there that the final placement of the communication instructions is made.

The parallelization on task layer uses a simple cost model for the actual timing: each time data is accessed by multiple cores, a cost is added to reflect the overhead. The algorithm then attempts to reduce the overall runtime while minimizing the communication overhead. This model can be used to determine the best split factors for the code layer. The results of the scheduling on the task layer can be seen in Fig. 4. It shows the load on the eight cores of the target platform over time. During the first half of the execution, the edge detection in both directions can very well utilize all available cores. While there is still some parallel processing left afterwards at the 350 ms mark, the end is dominated by functions that don't benefit much from the parallelization on either layer. For the actual Canny edge algorithm, a speedup of 6.28 is estimated while for the complete application, a speedup of 3.02 is estimated. A more accurate estimate of performance, including core wait times for data, is handled on the data layer.

4.4 Evaluation of Data Layer

Due to the nature of the algorithm, placement at the data layer is very straightforward: data is always sent after it has been split or computed in the previous step, and it is received immediately before further processing. It is important to transfer data between the cores as quickly as possible to minimize waiting times. To determine the best positions for the

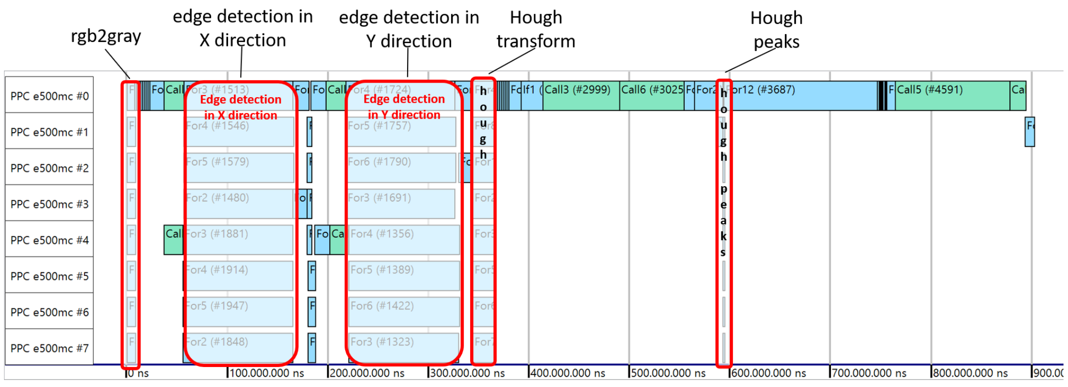


Figure 4 Scheduling result of the task layer.

synchronization between the cores, the control flow of the application is used to find the blocks that have the lowest number of executions while also minimizing any latency on the receiving cores. The communication overhead is relatively large so that some of the gains from the task layer cannot be implemented in the actual code for the hardware. Without any further changes on the data layer, a speedup of 2.4 was achieved. Further optimizations to reduce the communication overhead by implementing functions that directly access the shared memory regions gained an additional 19 % for a final speedup of 2.86.

5 Conclusion and Outlook

The runtime of the streak detection algorithm was reduced by 65 %, from an initial runtime of 2720 ms to 952 ms. Most of the optimization was done at the algorithm and code layer, but parallelization at the task layer was critical to map the prepared tasks to the available cores. Finally, the optimizations at the data layer were necessary to get most of the performance gains from the task layer to the actual hardware. This short evaluation shows the potential of this cross-layer approach and how the interaction between optimizations on different hierarchies can achieve better results compared to focusing only on two or three layers. However, more evaluations will be necessary to further analyze the benefits of this approach.

While the evaluation presented here was performed with a data-driven algorithm, the approach is not limited to it: further tests with more control-flow-driven algorithms also showed promising performance gains. Differences could be observed in the actual optimizations to be applied on the different layers. The examples used were of low complexity on the algorithmic side, so the task layer had little impact on the selection of the most efficient versions. The code layer still proved to be very valuable, but instead of ensuring that enough independent tasks were available for the task layer, smart clustering of the many tasks to reduce the actual number for the scheduling was the way to go. The task layer then had to find the most efficient distribution over the available cores, while the focus of the data layer is to minimize the amount of communication, since the cost per transfer is much higher when only few bytes need to be transferred.

It is also possible to extend the approach to support heterogeneous systems: the most relevant layers are then the algorithm layer and the code layer. The algorithm layer allows the use of optimized functions from existing libraries, while the code layer and its code transformations can be used to generate source code for accelerators such as CUDA or OpenCL.

References

1 Hamid Arabnejad, João Bispo, João M. P. Cardoso, and Jorge G. Barbosa. Source-to-source compilation targeting OpenMP-based automatic parallelization of C applications. *The Journal of Supercomputing*, 76(9):6753–6785, December 2019. doi:10.1007/s11227-019-03109-9.

2 Jürgen Becker, Thomas Bruckschloegl, Oliver Oey, Timo Stripf, George Goulas, Nick Raptis, Christos Valouxis, Panayiotis Alefragis, Nikolaos Voros, and Christos Gogos. Profile-Guided Compilation of Scilab Algorithms for Multiprocessor Systems. In *Reconfigurable Computing: Architectures, Tools, and Applications: 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14-16, 2014. Proceedings 10*, pages 330–336. Springer, 2014.

3 John Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, November 1986. doi:10.1109/tpami.1986.4767851.

4 Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. Progressive Raising in Multi-level IR. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, February 2021. doi:10.1109/cgo51591.2021.9370332.

5 Marco Danelutto, Gabriele Mencagli, Massimo Torquati, Horacio González–Vélez, and Peter Kilpatrick. Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming. *International Journal of Parallel Programming*, 49(2):177–198, November 2020. doi:10.1007/s10766-020-00684-w.

6 Richard O. Duda and Peter E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

7 Ian Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

8 Saiyedul Islam, Sundar Balasubramaniam, Shruti Gupta, Shikhar Brajesh, Rohan Badlani, Nitin Labhishetty, Abhinav Baid, Poonam Goyal, and Navneet Goyal. Pattern-Based Automatic Parallelization of Representative-Based Clustering Algorithms. In *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, October 2018. doi:10.1109/dsaa.2018.00020.

9 Nikita Kataev. Interactive Parallelization of C Programs in SAPFOR. In *SSI*, pages 139–148, 2020.

10 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, February 2021. doi:10.1109/cgo51591.2021.9370308.

11 Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. ACM, July 2017. doi:10.1145/3110355.3110356.

12 Luís Miguel Pinho, Eduardo Quinones, and Andrea Marongiu. *High-performance and time-predictable embedded computing*. River Publishers, 2018.

13 Sabri Pllana and Fatos Xhafa, editors. *Programming multi-core and many-core computing systems*. John Wiley & Sons, Inc., January 2017. doi:10.1002/9781119332015.

14 Todor Stefanov, Hristo Nikolov, Lubomir Bogdanov, and Angel Popov. DAEDALUS framework for high-level synthesis: Past, present and future. In *2021 25th International Conference Electronics*. IEEE, June 2021. doi:10.1109/ieeeconf52705.2021.9467445.

15 Jessica Vandebon, Jose G. F. Coutinho, Wayne Luk, Eriko Nurvitadhi, and Tim Todman. Artisan: a Meta-Programming Approach For Codifying Optimisation Strategies. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 177–185, 2020. doi:10.1109/FCCM48280.2020.00032.

16 Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. GSPMD: General and Scalable Parallelization for ML Computation Graphs, 2021. doi:10.48550/arXiv.2105.04663.

Accelerating Large-Scale Graph Processing with FPGAs: Lesson Learned and Future Directions

Marco Procaccini
University of Siena, Italy

Amin Sahebi
University of Siena, Italy

Marco Barbone
Imperial College London, UK

Wayne Luk
Imperial College London, UK

Georgi Gaydadjiev
Delft University of Technology, The Netherlands

Roberto Giorgi
University of Siena, Italy

Abstract

Processing graphs on a large scale presents a range of difficulties, including irregular memory access patterns, device memory limitations, and the need for effective partitioning in distributed systems, all of which can lead to performance problems on traditional architectures such as CPUs and GPUs. To address these challenges, recent research emphasizes the use of Field-Programmable Gate Arrays (FPGAs) within distributed frameworks, harnessing the power of FPGAs in a distributed environment for accelerated graph processing. This paper examines the effectiveness of a multi-FPGA distributed architecture in combination with a partitioning system to improve data locality and reduce inter-partition communication. Utilizing Hadoop at a higher level, the framework maps the graph to the hardware, efficiently distributing pre-processed data to FPGAs. The FPGA processing engine, integrated into a cluster framework, optimizes data transfers, using offline partitioning for large-scale graph distribution. A first evaluation of the framework is based on the popular PageRank algorithm, which assigns a value to each node in a graph based on its importance. In the realm of large-scale graphs, the single FPGA solution outperformed the GPU solution that were restricted by memory capacity and surpassing CPU speedup by 26x compared to 12x. Moreover, when a single FPGA device was limited due to the size of the graph, our performance model showed that a distributed system with multiple FPGAs could increase performance by around 12x. This highlights the effectiveness of our solution for handling large datasets that surpass on-chip memory restrictions.

2012 ACM Subject Classification Hardware → Hardware accelerators

Keywords and phrases Graph processing, Distributed computing, Grid partitioning, FPGA, Accelerators

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2024.6

Related Version
Full Version: `https://link.springer.com/article/10.1186/s40537-023-00756-x`

Supplementary Material
The source code related to this paper is available from the authors upon request.

1 Introduction

Graph computing is a specialized field within computer science dedicated to the analysis and manipulation of data organized in a graph structure. This approach aims to reveal the patterns, relationships, and insights inherent in interconnected data, particularly in domains such as social networks, biology, and transportation [22, 21]. The use of graph structures improves the extraction of valuable information, contributing to better decision-making and system improvement.

The emergence of big data has caused an increase in the size of models, datasets, and graphs, which are known as large-scale graphs. The volume of data organized in graph structures is expanding rapidly, necessitating efficient and scalable techniques for processing these large-scale graphs [18]. The difficulties arise from dealing with the large amount of data and the complex structure of the graphs. When dealing with large-scale graphs that contain billions of nodes and edges, the demand extends beyond the raw computing power. A single node is not capable of handling such large graphs, so distributed large-scale graph computing is a beneficial solution. Furthermore, specialized methods, such as graph partitioning or optimizing random memory access, are crucial for improving efficiency and scalability in managing these complex datasets.

Field Programmable Gate Arrays (FPGAs) are becoming increasingly popular for graph processing, offering an attractive alternative to conventional CPUs and GPUs [6, 9]. Indeed, GPUs are optimized for massively parallel workloads, such as those seen in Deep Neural Networks. However, their efficiency decreases when faced with applications that involve highly memory-sparse operations and issues related to data races[17]. On the other hand, CPUs, while more general-purpose, face limitations in large-scale graph processing due to their intrinsic architecture, which is not inherently optimized for the intricate and parallel nature of graph-related computations. Unlike conventional architectures, FPGAs are remarkable for their high level of customization, providing optimized performance for specialized tasks [5, 12, 10, 11].

This work outlines the challenges and insights associated with large-scale graph processing encountered during the evaluation of our framework presented in [19]. The main objective is to evaluate and suggest improvements based on the initial examination of a structure that uses large-scale graph processing with FPGAs in a distributed system based on Hadoop.

The rest of this paper is structured as follows: In Section 2 we present the motivation and challenges in dealing with large-scale graph computing. In Section 2.1, we introduce our framework for accelerating large-scale graph computing with FPGAs. In Sections 3 and 4 we discuss the methodology of design implementation and its evaluation. Finally, Section 5 presents our conclusion and provides a brief overview of potential future directions.

2 Motivation and Challenges

Recent research on graph processing using FPGAs [5] has been conducted mainly on medium-sized data sets, rather than large-scale ones. Sakr et al. [20] have shown that these medium-sized graphs can be managed by desktop CPUs. This could reduce the attractiveness of using hardware accelerators such as FPGAs or GPUs, as they are more expensive and less user-friendly than general-purpose CPUs.

Motivations

The emergence of big data technologies has changed the landscape, making it easier to collect, store, and process large amounts of data. This has led to an abundance of data, often in the form of graphs, which have grown to the point of reaching PetaBytes [20], surpassing the memory capacity of current CPUs or GPUs. For instance, when the graph size exceeds GPU memory limits, unified memory becomes essential. This requires frequent data transfers between host memory and GPU on-chip memory, incurring additional overhead and leading to performance degradation [16].

Our second motivation is to integrate a high-level interface for the deployment of a distributed platform on the underlying hardware. Hadoop is a valuable solution for large-scale graph processing, providing powerful tools for storing, processing, and analyzing extensive graph datasets in a distributed manner. The Hadoop Distributed File System (HDFS) allows for the storage of large datasets across a cluster of machines, making it possible to process graphs that are too large for a single machine. Hadoop's scalability allows for the flexible adaptation of the cluster by adding or removing machines, making it cost-effective to process large graphs. Using the map-reduce programming model inherent in Hadoop enables distributed computing, significantly improving the efficiency of graph processing algorithms.

Challenges

When designing large-scale graph processing on FPGAs, it is essential to make critical design decisions. To begin with, it is necessary to divide the graph into small, equal-sized parts due to the limited on-chip memory in modern FPGAs. This partitioning technique is essential since FPGAs do not have the capability to do dynamic memory allocations. It is of utmost importance to reduce external memory accesses, as data transfers can cause considerable overhead and have a negative effect on performance. The design of a processing kernel is critical as it should have a memory access pattern that is compatible with the partitioning scheme mentioned above. This decision is made to reduce communication overhead, particularly when reading host computer memory from the accelerator or between different accelerators. The processing kernel should be as efficient as possible, taking advantage of the parallelism that can be achieved on FPGAs. Multiple instances of computational units can be created to increase parallelism and improve performance.

2.1 Proposed Solution

In this section, we will delve into the structure of the framework, exploring the partitioning techniques employed, the single-FPGA architecture and its adaptation for the multi-FPGA distributed system.

Graph Partitioning

Graph partitioning is a well-discussed challenge in the graph computing literature, with numerous works proposing innovative techniques and algorithms [23].

This framework utilizes the GridGraph partitioning technique to divide the edges of a graph into smaller, self-contained chunks, each of which is assigned to a particular vertex. These chunks, along with the associated vertex data, are stored on the host computer's file system. This design ensures that the chunks are independent and compatible with the Block RAM (BRAM) size of the target FPGA. During the processing, the kernel reads the chunks in sequence from the host memory, updating the values, which are then written back. GridGraph is a partitioning technique that is selected for FPGA acceleration due to its advantageous tradeoffs. It is especially beneficial for large-scale graph processing scenarios, as it divides the graph into smaller grids for independent processing, resulting in high data locality and avoiding data conflicts [23]. Furthermore, the ability to map grids onto on-chip FPGA resources increases performance scalability. An illustration of graph partitioning with GridGraph is shown in Fig. 1, which divides the vertex set into grid subsets. Each partition represents outgoing edges for a specific range of vertices, facilitating an efficient iterative processing sequence. The process loads and processes edges from each partition in sequence, computing vertex values until a specific termination condition for the algorithm is met.

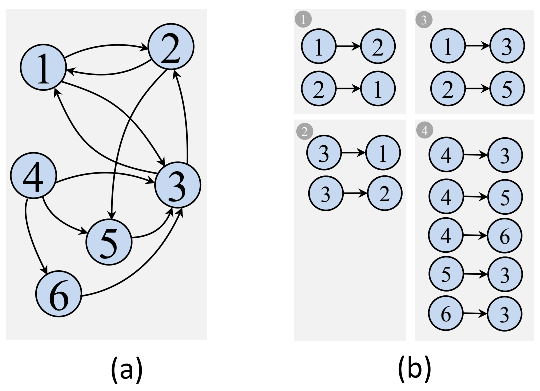


Figure 1 (a) A given sample graph. (b) Produced edge blocks using GridGraph partitioning. The number of partitions is P=2, producing P^2 edge blocks.

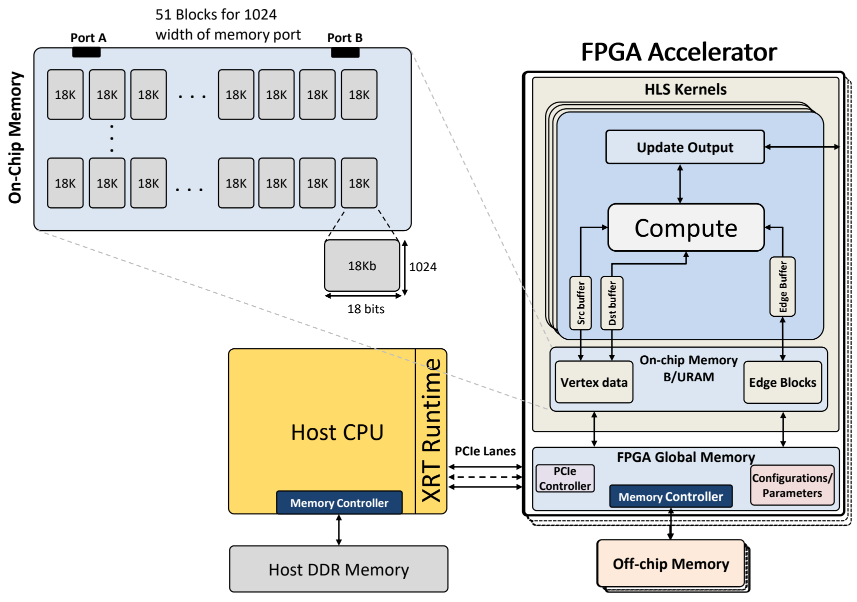


Figure 2 An overview block diagram of the proposed hardware design [19].

FPGA Architecture

In a single FPGA process, the graph data is pre-processed using the partitioning technique described in Section 2.1. The resulting edge blocks are then stored on the host file system. Given that the overall size of these graph blocks may reach terabytes, the host memory must be large enough to accommodate the reading of all edge blocks from the file system. Subsequently, the data is loaded into the FPGA on-chip memory before processing. Fig. 2 provides an overview of the single FPGA graph processing unit. In this representation, the on-chip memory is configured to optimize the memory bandwidth while maintaining the system frequency at its peak. Dedicated FPGA kernels are responsible for reading the data from the stream input provided by the host and directing them to the computational units within the FPGA. The units execute the graph algorithm, such as PageRank, afterwards. Upon completion of the computation, the aggregated results are written back to the host memory and stored in its file system.

Distributed Architecture

Graph datasets that are too large to be processed on a single machine can be handled in distributed computing. This involves distributing the data across multiple machines, allowing for the processing of graphs that would be too large for a single machine to handle. A distributed system is made up of multiple machines that work together as one virtual system. Each machine or node is responsible for processing a portion of the data. These systems can be managed manually with custom software, such as popular message passing interfaces (MPI). This manual management allows for precise application optimization, leading to high performance. However, this approach requires a lot of engineering effort and knowledge, especially in the field of distributed computing. Various distributed computing frameworks are available that automate and tackle many of the issues discussed earlier. Hadoop, a popular open source framework created by the Apache Software Foundation in 2005, is a widely used technology for large-scale data processing in machine clusters [2]. It is based on the map-reduce programming model and enables parallel processing of large amounts of data across a distributed platform, due to the integration of the Hadoop Distributed File System (HDFS). Data are usually stored in a distributed file system, such as HDFS or ZFS, and processed through a distributed computing framework, such as Apache Hadoop. Graph algorithms, including PageRank, can be implemented in these frameworks for graph processing and analysis [8]. Recently, the integration of FPGAs with Hadoop for large graph processing has gained attention [3]. This combination allows one to take advantage of Hadoop's scalability and fault tolerance while taking advantage of the high performance of FPGAs for graph processing tasks. Although the use of FPGAs with Hadoop is still in its early stages, further research is needed to evaluate the feasibility of using FPGAs in combination with Hadoop to speed up graph processing and optimize system performance.

Data processing within the map-reduce architecture involves two main phases. In the initial "Map" phase, the data are divided into smaller chunks known as input splits. Each split is processed by a separate node in the cluster, utilizing user-defined functions called Mappers to transform the input data into intermediate key-value pairs. The subsequent "Reduce" phase processes these intermediate key-value pairs using user-defined functions called Reducers, which merge the input data into the final output. In a map-reduce system, a user application initiates a root controller and a set of mappers and reducers distributed across various compute nodes. The root node coordinates the generation of mappers and reducers and monitors their progress. The overall system overview of the Hadoop map-reduce design is depicted in Fig. 3.

In our scenario, nodes with multiple FPGA accelerators are configured to appear as multiple nodes with a single FPGA to Hadoop. For instance, a node with four FPGAs executes four distinct instances of Hadoop, each FPGA being mapped one-to-one to an instance. This design simplifies the distribution of workload across multiple FPGAs, eliminating the need for manual splitting. Moreover, it allows for the use of Hadoop scheduling for load balancing and fault tolerance. The Hadoop scheduler can handle single FPGA failures without taking the entire node offline. The framework for large-scale graph processing involves three primary phases (see Fig. 3). During the initial stage, the graph is divided into smaller sections, named edge-blocks, using the GridGraph partitioning technique. This initial step serves as pre-processing, enabling the parallel processing of sub-graphs through a single scan. The second phase constitutes the core processing stage, where a customized Map function executes the graph processing algorithm on the previously computed subgraphs (e.g., PageRank). The third and final phase involves the integration of partial results generated by different workers using a custom Reduce function. This step is optional and can be utilized to merge results

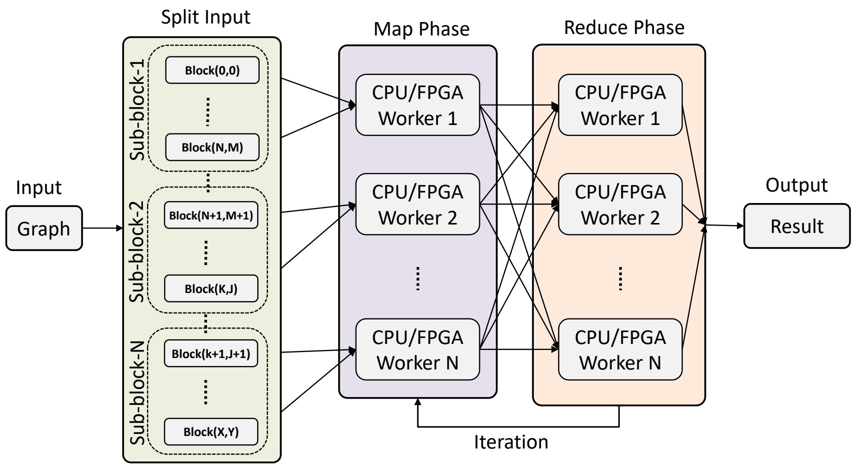


Figure 3 The Hadoop framework for distributed graph processing [19].

saved in different files by Mappers. However, the necessity of consolidating results into a single monolithic file may vary depending on the specific use case. For iterative algorithms, such as PageRank, it is necessary to repeat phases 2 and 3 until the algorithm converges.

3 Methodology

In this section, our aim is to provide insight into the methodology employed in developing the graph processing framework. This framework is implemented using the Vivado HLS toolchain (version 2022.2) on the Xilinx Alveo U250 board, featuring the Xilinx VU13P FPGA. Xilinx Alveo boards serve as Data Center accelerator cards tailored for diverse tasks such as machine learning inference, video transcoding, and database search and analytics. The design utilizes the Vivado design suite (Vitis version 2022.2) and is divided into two parts: implementation of the kernel and host program. The host program, executed on the CPU, comprises two main stages. The initial stage involves receiving and preparing pre-processed graph data (see Section 2.1) and parameters, while the subsequent step creates buffers for optimal memory alignment to enhance performance. The bridge between the host and the kernels is established through runtime buffers and commands to program the FPGA device with the bitstream. Data exchange between host memory and kernel local memory, like BRAM, is facilitated by using OpenCL functions and specific FPGA kernels.

On the contrary, the kernel implementation, executed on the FPGA, is written in HLS and structured to optimally utilize accelerator resources. Efficient FPGA implementation is influenced by structures such as local arrays, which require careful resource allocation in terms of Lookup Tables (LUTs), Block RAM (BRAM), and registers. Strategies such as partitioning and streaming data through small and fast FIFOs are employed to minimize resource utilization and accommodate large local arrays on FPGAs. To enhance performance, the proposed design incorporates tuning at the kernel link stage, allowing a single kernel to instantiate multiple hardware compute units (CUs). The host program initiates overlapping kernel calls, executing kernels concurrently by running independent CUs.

In this paper, we present the performance evaluation of our framework in the single FPGA configuration, extracting the execution time of the FPGA kernel. For distributed execution, since there is no access to a Hadoop cluster, we employed a mathematical model to estimate performance. The map-reduce programming model has been extensively explored in the

Table 1 The XACC server is used to evaluate the real implementation.

Instance Name	CPU	CPU Freq	Cores	Memory	FPGA board
alveo1.ethz.ch	2× Intel Xeon Gold 6234	3.30 GHz	16	376 GiB	Alveo U250

literature, allowing for predictable performance modeling [7, 13]. Consequently, this study relies on performance forecasts drawn from previous analyses of algorithm efficiency in map-reduce implementations. Before delving into the analysis, it is crucial to underline specific assumptions. First, the graph is stored in the Hadoop Distributed File System (HDFS), and second, graph partitioning occurs in parallel through a MapReduce job or custom partitioner. The first assumption remains valid, given the focus on large-scale graphs, which are too extensive for a single node, making it likely that they are stored in a distributed file system. The second assumption is valid as long as the first holds. Once the graph is situated in the file system, any necessary pre-processing can be directly executed in MapReduce. Given these assumptions, the graph scale necessitates analysis in a distributed system, as attempting such an analysis on a single parallel node would require an impractical amount of time. In the distributed framework, FPGA accelerators are exclusive to mappers for computing a partial state. The Eq. (1) provided is essential for assessing how a faster mapper would impact the overall computation time.

$$T = T_{split-input} + T_{overhead} + N(T_{scatter} + max(T_{map}) + T_{transfer} + T_{reduce}) \tag{1}$$

where:

- $T_{split-input}$ time needed to partition the graph in sub-graphs;
- N is the number of iterations of the iterative algorithm;
- $T_{overhead}$ is the overhead introduced by map-reduce;
- $T_{scatter}$ is the time needed to distribute the state vector to all the workers;
- $max(T_{map})$ is the time needed by the slowest mapper;
- $T_{transfer}$ is the time needed to transfer the state vector to the reducers (account for shuffle and sort);
- T_{reduce} is the time needed to aggregate the partial state vectors.

4 Evaluation

This section presents the first evaluation of the framework using an optimized version of the PageRank algorithm. PageRank is often used as a standard for evaluating the performance of large graph processing. It is one of the most computationally intensive graph algorithms and requires the processing of a large number of vertices and edges. This algorithm is used to assess the importance of each node in a graph, which was initially used by search engines to rank webpages based on their relevance. PageRank assigns a score, referred to as the PageRank score, to each vertex, based on the number and importance of the vertices pointing to it. Vertices with higher PageRank scores are considered more significant than those with lower scores. Using PageRank as a measure allows for the assessment of the proposed model's effectiveness in dealing with large graph data efficiently.

Table 2 The datasets for evaluating our proposed study. We choose them based on the size and the structure of the datasets to be comparable with other works.

Graph dataset	Vertices	Edges	Size (GB)	Type
LiveJournal [15]	4.8M	0.069 B	1.1	Social Web
Web-UK-2005 [4]	39M	0.994 B	16	Web Graph
Twitter [14]	41.6 M	1.47 B	23	Web Graph
Friendster [15]	68.3M	2.58 B	43	Social Web

The implementation of the framework described in section 2.1 has been evaluated against CPU, GPU, and FPGA architectures. The hardware implementation was carried out on a Xilinx Alveo U250, which was integrated into Xilinx Adaptive Compute Clusters (XACC) [1]. The XACC servers distributed resources evenly across multiple Virtual Machines (VMs), guaranteeing that each VM had access to its own FPGA card. The software environment within the VM is based on Ubuntu 20.04 and incorporates FPGA accelerator deployment frameworks such as Vitis and Vivado HLS. Table 1 provides detailed specifications for the server and the hardware accelerator.

In the context of PageRank comparisons with the CPU, we used both a sequential version, an OpenMP multicore version, and the GridGraph library, recognized as one of the most efficient graph processing frameworks for CPUs. The OpenMP version used in this assessment is version 4.0.3, and the software is compiled using GCC version 9.4.0. In the case of GPU implementation, we used the cuGraph library for comparison, using the CUDA toolkit version 11.7. "cuGraph" is an open source GPU graph analysis library integrated into the RAPIDS ecosystem, offering a high-performance, user-friendly, and extensible framework for GPU-based graph analysis. Our experiments used the NVIDIA V100 GPU, renowned for its high performance based on the Volta architecture. The cugraph library was chosen not only for its capabilities in single-GPU execution, but also for its potential in running multi-GPU executions to facilitate a comprehensive comparison with our distributed execution. Nevertheless, it is important to mention that our attempts to execute the multi-GPU function with our extensive dataset were unsuccessful, indicating the need for additional effort to resolve the issue. To evaluate our optimized PageRank algorithm, we selected graphs of various sizes from the datasets listed in Table 2, ranging from a small graph (LiveJournal) to a large graph (Friendster). These datasets were intentionally chosen to assess performance as the graph size exceeds the memory capacities of the CPU or GPU devices.

Fig. 4 shows that the FPGA-based PageRank implementation outperforms sequential and OpenMP execution on a CPU for all datasets. The speedup achieved ranges from approximately 9.7x for the smallest dataset to about 26x for the Twitter dataset. Compared to the GridGraph library, which employs a grid partition schema similar to ours and uses OpenMP for parallel execution, the framework shows a performance improvement with a speedup up to 1.5x (Twitter). Our FPGA implementation not only overcomes CPU solutions but also achieved a speedup of up to 4.5x compared to the cuGraph library (Web-UK-2005). In particular, the Web-UK-2005 data set, exceeding GPU on-chip memory, requires the use of Unified Memory, incurring additional overhead and degrading performance due to data transfers between host memory and GPU on-chip memory. Larger datasets (Twitter and Friendster) cause GPU experiments to fail due to insufficient memory on the GPU board.

Table 3 Alveo U250 platform experimental results in comparison to a software baseline and state-of-the-art FPGA works. Reported numbers are all in Seconds.

Dataset	Sequential CPU 1)	OpenMP CPU 1)	GridGraph CPU 1)	cuGraph GPU 2)	Vitis Library FPGA	Our work FPGA
LiveJournal	27.01	5.49	3.54	5.28	79.79	2.78
Web-UK-2005	275.44	185.4	34.9	90.73	N/A 3)	20.6
Twitter	1443	658.5	88.5	Failed 4)	N/A 3)	55.6
Friendster	2258	950	141	Failed 4)	N/A 3)	95.4

1) CPU details are described in Table 1.
2) The GPU used for the experiments is a NVIDIA Volta V100.
3) N/A indicates that the aforementioned study did not report this dataset evaluation.
4) The experiment hit the GPU memory limit.

These results underscore the advantages of employing FPGA for graph processing tasks, especially with large datasets. Furthermore, the framework surpasses the PageRank algorithm available in the Vitis Library, achieving a speedup of about 28x. This emphasizes the benefits of custom FPGA implementations for graph processing tasks, particularly with large datasets that surpass on-chip memory capacity. The findings position our FPGA implementation as a suitable solution for accelerating graph processing tasks.

It is noteworthy that the framework's speedup nearly doubles from Web-UK-2005 to Twitter, despite Twitter's size being approximately 1.5 times larger than Web-UK-2005 (see Table 2). However, the performance boost diminishes with the larger Friendster dataset, prompting an exploration of whether a multi-FPGA solution in a distributed system, like Hadoop, could enhance FPGA usage for large-scale graph processing.

Graphs larger than Friendster exceed the computing capabilities of contemporary machines. To address this, a potential solution involves distributing these large graphs across multiple machines through distributed computing. Fig. 5 presents forecasts for the integration of FPGA acceleration in a Hadoop-distributed cluster for large-scale graph processing. Profiling the code and using the results to evaluate Eq. (1) indicates that FPGA utilization is most effective when a significant amount of time is spent in the mapping phase. In the forecasts, it is assumed that the majority of time (51% of the total time) is spent in the mapping phase. Under these conditions and considering a worst-case scenario in which FPGA-accelerated nodes achieve only a 20x speedup compared to CPU-only nodes, the forecasts show a potential 54% reduction in total time. In a more realistic scenario where 80% of the time is spent in the mapping phase (a common assumption with partition methods such as GridGraph to minimize data transfer), the time reduction achieved by a hybrid CPU-FPGA Hadoop cluster can increase to 84% compared to a CPU-only cluster. Fig. 5 summarizes these forecasts and illustrates the best-case scenario (although unlikely) where 90% of the total execution time is spent in the mapping phase, showing a potential over 90% time reduction under optimal conditions.

This initial evaluation has revealed the current performance of the framework, as well as providing useful knowledge that can be used to improve it. Examining the results has exposed areas where the framework can be improved and optimized. We investigated the graph partitioning technique and discovered that the first possible enhancements can be achieved by refining the graph partitioning approach described in Section 2.1. As observed

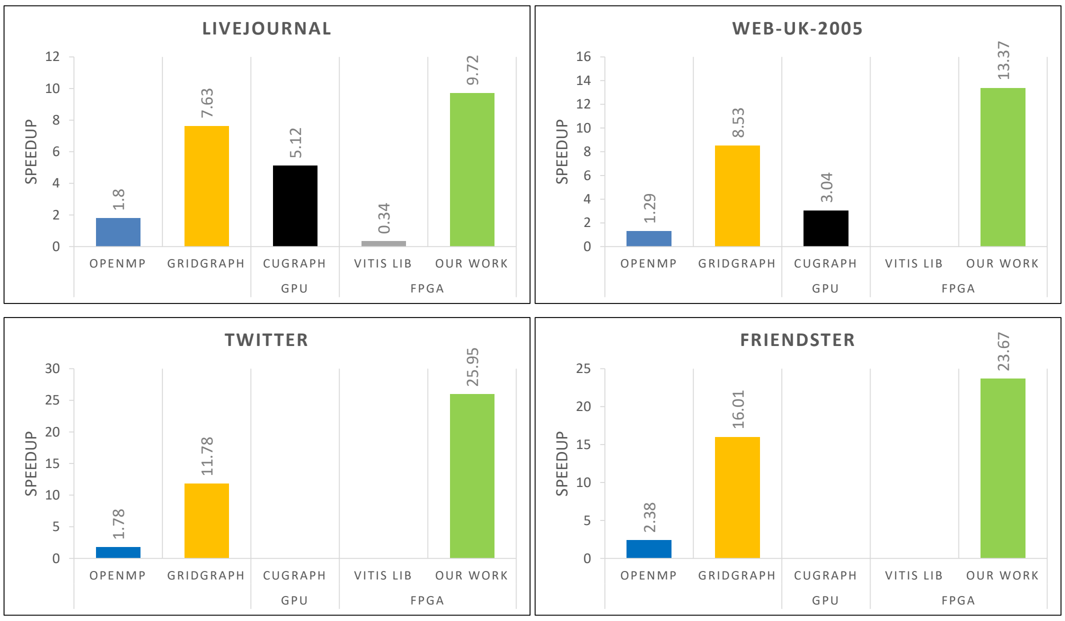


Figure 4 Speedup evaluation on the sequential execution of the proposed FPGA PageRank algorithm (our work) with the CPU, GPU, and FPGA solutions for the LiveJournal, Web-UK-2005, Twitter, and Friendstser datasets [19].

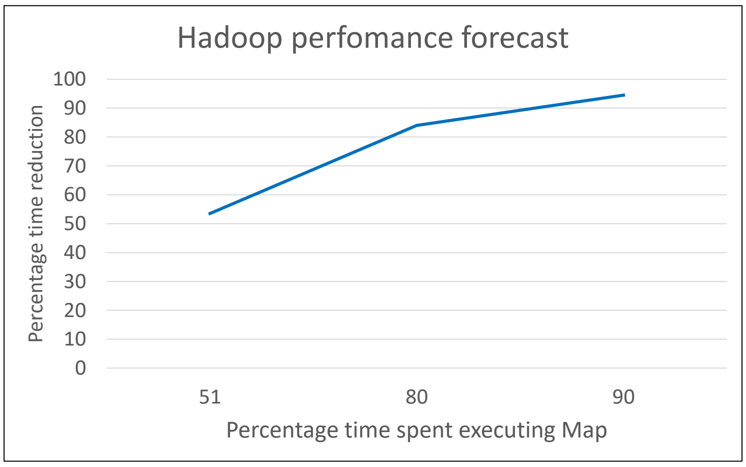


Figure 5 Reduced processing time when employing FPGAs for accelerated graph processing in contrast to utilizing a Hadoop cluster without FPGA integration [19].

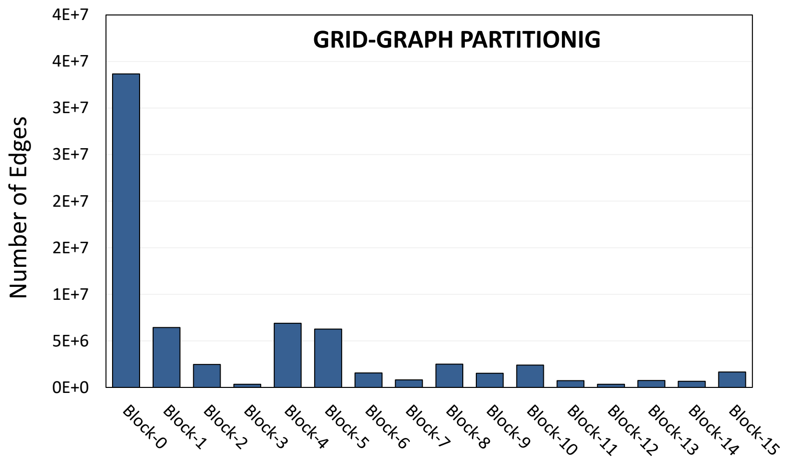


Figure 6 Edge block partitioning of the LiveJournal dataset with P=4 partitions using the GridGraph method.

in Fig. 1, the partitions are unevenly balanced in the grid, with partition 4 being larger than the others. Our investigation extended to the LiveJournal dataset, where the execution of the grid graph partitioning exhibited a similar imbalance, as illustrated in Fig. 6. The output of the grid graph resulted in one substantial partition (block 0) and several smaller partitions. This lack of balance in workload could lead to notable performance declines. Addressing this issue requires the development of an improved version of the graph partitioning method to ensure a more balanced workload, which can significantly improve overall performance.

5 Conclusions

The field of large graph processing is growing in importance as the amount of data generated by applications such as social networks, web graphs, and biological networks continues to increase. To handle this increase in graph sizes, efficient and scalable processing methods must be developed. The integration of Field Programmable Gate Arrays (FPGAs) with the open source Hadoop framework is becoming increasingly popular for the purpose of managing large graph datasets. FPGAs, which are integrated circuits designed for specific tasks, are highly efficient when it comes to executing graph processing algorithms. Hadoop, on the other hand, is a widely used platform for distributed processing of large datasets. The first evaluation of our framework has been shown to be more effective than existing implementations for the PageRank algorithm. This highlights the usefulness of FPGA-based solutions for large datasets. Additionally, the combination of FPGAs and Hadoop can potentially improve performance when dealing with large datasets. This initial evaluation highlights the potential improvements of the framework, stressing the importance of taking into account factors such as graph partitioning. Further research is needed to broaden the benchmark set for a more comprehensive evaluation and to look into other essential aspects such as the host-FPGA communication and the FPGA memory usage.

References

1 AMD Xilinx. Heterogeneous Accelerated Compute Clusters. `https://www.amd-haccs.io/`, 2023 (accessed 15 November 2023).

2 Apache Hadoop. Hadoop. Accessed 30 Jan 2023. URL: `https://hadoop.apache.org/`.

3 Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, et al. The future of FPGA acceleration in datacenters and the cloud. *ACM TRETS*, 15(3):1–42, 2022.

4 Paolo Boldi and Sebastiano Vigna. The web graph framework I: Compression techniques. In *Proc. of the 13th ACM International WWW Conference*, pages 595–601, NY, USA, 2004.

5 Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. ThunderGP: HLS-based graph processing framework on FPGAs. In *The ACM/SIGDA International Symposium on FPGAs*, FPGA '21, pages 69–80, New York, NY, USA, 2021.

6 Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture. In *Proc. of the 2017 ACM/SIGDA International Symposium on FPGAs*, FPGA '17, page 217226, NY, USA, 2017.

7 Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

8 Benedikt Elser and Alberto Montresor. An evaluation study of BigData frameworks for graph processing. In *2013 IEEE International Conference on Big Data*, pages 60–67, 2013.

9 Nina Engelhardt and Hayden K.-H. So. GraVF-M: Graph processing system generation for Multi-FPGA platforms. *ACM Trans. Reconfigurable Technol. Syst.*, 12(4), November 2019.

10 Antonio Filgueras, Miquel Vidal, Marc Mateu, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Dimitrios Theodoropoulos, Dionisios Pnevmatikatos, Paolo Gai, Stefano Garzarella, David Oro, Javier Hernando, Nicola Bettin, Alberto Pomella, Marco Procaccini, and Roberto Giorgi. The axiom project: Iot on heterogeneous embedded platforms. *IEEE Design Test*, pages 1–1, 2019.

11 R. Giorgi, F. Khalili, and M. Procaccini. AXIOM: A Scalable, Efficient and Reconfigurable Embedded Platform. In *IEEE Proc.DATE*, pages 1–6, March 2019.

12 R. Giorgi, Farnam. Khalili, and Marco Procaccini. Translating Timing into an Architecture: The Synergy of COTSon and HLS. *Hindawi – IJRC*, 2019:1–18, December 2019.

13 Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010.

14 Jérôme Kunegis. Konect: The koblenz network collection. In *Proc. of the 22nd Int. Conf. on WWW*, pages 1343–1350, New York, NY, USA, 2013. ACM.

15 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, 2014. URL: `http://snap.stanford.edu/data`, (accessed 15 November 2023).

16 Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. An evaluation of unified memory technology on NVIDIA GPUs. In *2015 15th IEEE/ACM international symposium on cluster, cloud and grid computing*, pages 1092–1098. IEEE, 2015.

17 Yashuai Lü, Hui Guo, Libo Huang, Qi Yu, Li Shen, Nong Xiao, and Zhiying Wang. GraphPEG: Accelerating graph processing on GPUs. *ACM TACO*, 18(3):1–24, 2021.

18 M. Usman Nisar, Arash Fard, and John A. Miller. Techniques for graph analytics on big data. In *2013 IEEE International Congress on Big Data*, pages 255–262, 2013.

19 Amin Sahebi, Marco Barbone, Marco Procaccini, Wayne Luk, Georgi Gaydadjiev, and Roberto Giorgi. Distributed large-scale graph processing on fpgas. *Journal of Big Data*, 10(1):95, 2023.

20 Sherif Sakr and et al. Bonifati. The future is big graphs: A community view on graph processing systems. *Commun. ACM*, 64(9):62–71, August 2021.

21 Justin S Smith, Adrian E Roitberg, and Olexandr Isayev. Transforming computational drug discovery with machine learning and AI, 2018.

22 Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A Comprehensive Survey on Graph Neural Networks. *IEEE Trans. on NNLS*, 32(1):4–24, 2021.

23 Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *Proc. of the 2015 Conf. on Usenix Annual Technical Conference*, pages 375–386, USA, 2015.