Performance engineering of data-intensive applications

Performance Engineering datenintensiver Anwendungen
Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
Genehmigte Dissertation von M.Sc. Arya Mazaheri aus Esfahan, Iran
Tag der Einreichung: 2. August 2021, Tag der Prüfung: 21. Oktober 2021

1. Gutachten: Prof. Dr. Felix Wolf, TU Darmstadt
2. Gutachten: Prof. Dr. Ali Jannesari, Iowa State University
3. Gutachten: Prof. Dr. Morris Riedel, University of Iceland, Forschungszentrum Jülich
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department
Laboratory for Parallel Programming

Performance engineering of data-intensive applications
Performance Engineering datenintensiver Anwendungen

Accepted doctoral thesis by M.Sc. Arya Mazaheri

1. Review: Prof. Dr. Felix Wolf, TU Darmstadt
2. Review: Prof. Dr. Ali Jannesari, Iowa State University
3. Review: Prof. Dr. Morris Riedel, University of Iceland, Forschungszentrum Jülich

Date of submission: 2. August 2021
Date of thesis defense: 21. Oktober 2021

Darmstadt

To my parents, who continually encouraged me to pursue my dreams

&

to my loving wife, Salome, for her endless support and patience.

Erklärungen laut Promotionsordnung

§8 Abs. 1 lit. c PromO

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

§8 Abs. 1 lit. d PromO

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

§9 Abs. 1 PromO

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

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 2. August 2021

A. Mazaheri

Abstract

Data-intensive programs deal with big chunks of data and often contain compute-intensive characteristics. Among various HPC application domains, big data analytics, machine learning and the more recent deep-learning models are well-known data-intensive applications. An efficient design of such applications demands extensive knowledge of the target hardware and software, particularly the memory/cache hierarchy and the data communication among threads/processes. Such a requirement makes code development an arduous task, as inappropriate data structures and algorithm design may result in superfluous runtime, let alone hardware incompatibilities while porting the code to other platforms.

In this dissertation, we introduce a set of tools and methods for the performance engineering of parallel data-intensive programs. We start with performance profiling to gain insights on thread communications and relevant code optimizations. Then, by narrowing down our scope to deep-learning applications, we introduce our tools for enhancing the performance portability and scalability of convolutional neural networks (ConvNet) at inference and training phases.

Our first contribution is a novel performance-profiling method to unveil potential communication bottlenecks caused by data-access patterns and thread interactions. Our findings show that the data shared between a pair of threads should be reused with a reasonably short intervals to preserve data locality, yet existing profilers neglect them and mainly report the communication volume. We propose new hardware-independent metrics to characterize thread communication and provide suggestions for applying appropriate optimizations on a specific code region. Our experiments show that applying relevant optimizations improves the performance in Rodinia benchmarks by up to 56%.

For the next contribution, we developed a framework for automatic generation of efficient and performance-portable convolution kernels, including Winograd convolutions, for various GPU platforms. We employed a synergy of meta-programming, symbolic execution, and auto-tuning. The results demonstrate efficient kernels generated through an automated optimization pipeline with runtimes close to vendor deep-learning libraries, and the minimum required programming effort confirms the performance portability of our approach. Furthermore, our symbolic execution method exploits repetitive patterns in Winograd convolutions, enabling us to reduce the number of arithmetic operations by up to 62% without compromising the numerical stability.

Lastly, we investigate possible methods to scale the performance of ConvNets in training and inference phases. Our specialized training platform equipped with a novel topology-aware network pruning algorithm enables rapid training, neural architecture search, and network compression. Thus, an AI model training can be easily scaled to a multitude of compute nodes, leading to faster model design with less operating costs. Furthermore, the network compression component scales a ConvNet model down by removing redundant layers, preparing the model for a more pertinent deployment.

Altogether, this work demonstrates the necessity and shows the benefit of performance engineering and parallel programming methods in accelerating emerging data-intensive workloads. With the help of the proposed tools and techniques, we pinpoint data communication bottlenecks and achieve performance portability and scalability in data-intensive applications.

Zusammenfassung

Datenintensive Anwendungen arbeiten mit großen Datenmengen und sind daher häufig sehr berechnungsintensiv. Bekannte Beispiele solcher Anwendungen im HPC-Bereich sind Big Data Analyse, Machine Learning und seit Neuestem auch Deep Learning Modelle. Der Entwurf von effizienten datenintensiven Anwendungen erfordert umfangreiches Wissen in Bezug auf Zielhardware und -software, hierbei sind insbesondere die Speicher- und Cache-Hierarchie sowie die Kommunikation zwischen Threads und Prozessen von Bedeutung. Für die Softwareentwicklung stellt dies eine große Herausforderung dar, weil unpassende Datenstrukturen und Algorithmen zu suboptimaler Laufzeitperformanz führen und auch der Transfer auf neue Zielplattformen Kompatibilitätsprobleme mit der Hardware verursachen kann.

In dieser Dissertation werden Werkzeuge und Methoden für das Performance-Engineering von parallelen, datenintensiven Anwendungen vorgestellt. Zuerst wird die Analyse des Laufzeitverhaltens solcher Anwendungen behandelt, um Erkenntnisse über die Kommunikation zwischen Threads zu gewinnen und damit entsprechende Codeoptimierungen vornehmen zu können. Anschließend werden Werkzeuge speziell für Deep-Learning-Anwendungen vorgestellt, die die Performanz-Portabilität und Skalierbarkeit von Convolutional Neural Networks in der Inferenz- und Trainingsphase verbessern.

Der erste Beitrag ist eine neuartige Performance-Profiling-Methode, die potenzielle Engpässe in der Kommunikation aufzeigt, welche durch Datenzugriffsmuster und Threadinteraktionen verursacht werden. Die Untersuchungen hierzu haben ergeben, dass die gemeinsam verwendeten Daten zweier Threads in einem bestimmten Abstand zueinander wiederverwendet werden müssen, um die Datenlokalität zu bewahren. Existierende Profiler berücksichtigen stattdessen meist nur das gesamte Kommunikationsvolumen. Weiterhin werden neue hardwareunabhängige Metriken zur Charakterisierung von Threadkommunikation und Empfehlungen für die Optimierung relevanter Coderegionen vorgeschlagen. Experimente zeigen, dass solche Optimierungen in den Rodinia Benchmarks eine Verbesserung von bis zu 56 % erreichen.

Für den nächsten Beitrag wurde ein Framework zur automatisierten Generierung von effizienten und portablen Convolution-Kernels (u.a. Winograd Convolution) entwickelt, das verschiedene GPU-Plattformen unterstützt. Dazu wird eine Kombination von Metaprogrammierung, symbolischer Ausführung und Autotuning verwendet. Die Experimente zeigen, dass mit Hilfe dieser automatisierten Optimierungspipeline effiziente Convolution Kernel generiert werden, die nahezu die Laufzeiten der Deep-Learning Bibliotheken von etablierten Anbietern erreichen. Zudem unterstreicht der minimale Programmieraufwand die Leistungsportabilität dieses Ansatzes. Die Verwendung von symbolischer Ausführung erlaubt es, sich wiederholende Muster in der Winograd Convolution auszunutzen, sodass bis zu 62 % der arithmetischen Operationen eingespart werden können, ohne die numerische Stabilität zu beeinflussen.

Abschließend werden Methoden zur Skalierung der Performanz von Convolutional Neural Networks in der Inferenz- und Trainingsphase untersucht. Die vorgestellte, spezialisierte Trainingsplattform, die mit einem neuartigen Topologie-bewussten Netzwerkpruning Algorithmus ausgestattet ist, erlaubt schnelles Training von neuronalen Netzen, schnelle Neural Architecture Search und schnelle Netzwerkkompression. Dadurch können KI-Modelle einfach auf eine Vielzahl von Rechenknoten hochskaliert werden, was schneller zu fertigen KI-Modellen bei gleichzeitig geringeren Betriebskosten führt. Darüber hinaus reduziert die Netzwerkkompression die Größe der Convolutional Neural Networks, indem es überflüssige Ebenen entfernt und dadurch die Performanz der Inferenz steigert.

Diese Arbeit demonstriert die Notwendigkeit und die Vorteile von Performance Engineering sowie paralleler Programmiermethoden, um die immer häufiger vorkommenden datenintensiven Anwendungen zu beschleunigen. Mit Hilfe der vorgestellten Werkzeuge und Techniken wurden Engpässe in der Datenkommunikation aufgezeigt und Portabilität und Skalierbarkeit von datenintensiven Anwendungen erzielt.

Acknowledgements

Similar to many other doctoral graduates, my PhD journey involved several moments of bewilderness and uncertainties, yet the outcome gives me joy and pleasure, as I have become a more skilled person in many different ways. During my entire PhD program, I was fortunate to have the opportunity of doing independent research on the topics that excited me the most. All of which were not possible without the help and support of my advisors. Thus, first and foremost, I would like to express my highest gratitude to Prof. Dr. Felix Wolf and Prof. Dr. Ali Jannesari for giving me this level of freedom and inspiration to pursue my dreams. Under their guidance, I became competent in my field of research and got a keen eye for detail along with several soft skills. I cannot imagine finishing this work without their dedication in providing me with invaluable and prompt feedback.

I would also like to sincerely appreciate Dr. Matthew Moskewicz for his inspiring passion and ideas that motivated me to work on cutting-edge projects and to pursue new directions in my work. I am extremely thankful to Sixing Yu and Tim Beringer for our fruitful discussions and collaboration on the network compression method and training platform, respectively. I also thank Dr. Alexandru Calotoiu for our insightful discussions. Moreover, I am grateful to my industrial partner, Dr. Heiko Schick at Huawei's Munich research center, for his consultation at later stages of my research project. The scalability experiments were conducted on the Lichtenberg high performance computer of TU Darmstadt, for which I am highly thankful. Additionally, I am grateful to Prof. Dr. Morris Riedel for agreeing to review this dissertation.

I wish to thank all my colleagues in the Laboratory for Parallel Programming (LPP) at Technical University Darmstadt for the cheerful moments and the support they offered. Especially, I want to thank Petra Stegmann for her continual support during my entire stay at LPP and invaluable feedback she provided on my doctoral dissertation. Furthermore, I would like to thank my bachelor and master students who helped me with their coding support. Johannes Schulte implemented the Vulkan backend. I also had the pleasure of working with Tim Beringer during his bachelor and master studies on implementing Winograd convolution and adaptive training, for which I am highly grateful.

In addition to the people and institutions mentioned above, who directly helped me in my research, I would like to thank my friend, Alireza Mirian, who motivated me to take on the PhD journey. Moreover, I cannot thank Salome enough for her unconditional and unwavering understanding, patience, and support during busy or stressful times. Last but not least, I am deeply grateful to my parents for supporting me throughout all my studies and their never-ending moral and emotional support.

Contents

Acknowledgements **xi**

List of Figures **xv**

List of Tables **xviii**

1 Introduction **1**
- 1.1 Data-intensive applications . . . 1
  - 1.1.1 Deep-learning applications . . . 1
- 1.2 Performance engineering . . . 4
  - 1.2.1 Performance profiling . . . 4
  - 1.2.2 Performance portability . . . 5
  - 1.2.3 Performance scalability . . . 5
- 1.3 Contributions . . . 6
- 1.4 Structure of this dissertation . . . 7
- 1.5 Statement of originality . . . 7

2 Performance profiling: Unveiling communication bottlenecks **9**
- 2.1 Background and motivation . . . 9
  - 2.1.1 Communication in shared memory systems . . . 10
  - 2.1.2 Reuse distance analysis . . . 11
  - 2.1.3 DiscoPoP profiler . . . 12
- 2.2 Characterizing the communication behavior of parallel programs . . . 12
  - 2.2.1 Instrumentation and profiling . . . 12
  - 2.2.2 Communication pattern detection . . . 14
  - 2.2.3 CRD: Communication reuse distance . . . 15
  - 2.2.4 Communication bottleneck analysis . . . 18
- 2.3 Experimental results . . . 19
  - 2.3.1 Communication analysis validation . . . 19
  - 2.3.2 Communication scalability analysis . . . 22
  - 2.3.3 Communication bottleneck analysis . . . 23
- 2.4 Related work . . . 23
- 2.5 Discussion . . . 25
- 2.6 Conclusion . . . 26

3 A primer on deep learning **29**
- 3.1 Algebraic data structures – Tensors . . . 29
- 3.2 Deep neural networks . . . 30
  - 3.2.1 Training phase . . . 31
  - 3.2.2 Inference phase . . . 32

3.3 Performance analysis of deep-learning models . . . 32
3.4 Convolutional neural networks . . . 34
3.4.1 Building blocks of convolutional neural networks . . . 36
3.4.2 Winograd convolution . . . 38
3.5 Neural architecture search . . . 40
3.6 Reinforcement learning . . . 41
3.7 Graph convolutional networks (GCN) . . . 41

4 Performance portability: Efficient and performance-portable ConvNet deployment 43
4.1 Motivation . . . 43
4.1.1 Why do we need different convolution types? . . . 44
4.1.2 How can we make ConvNets more efficient? . . . 44
4.1.3 How can we achieve performance portability in ConvNets? . . . 45
4.2 Background on Boda . . . 47
4.3 Comparison of CUDA, OpenCL, and Vulkan . . . 48
4.3.1 Programming conventions . . . 49
4.4 Boda+ framework . . . 51
4.4.1 MetaGPU abstraction layer . . . 52
4.4.2 Winograd transformation optimization . . . 53
4.4.3 Code generation . . . 55
4.4.4 Variant selection and auto-tuning . . . 57
4.5 Experimental results . . . 57
4.5.1 Programmability analysis . . . 58
4.5.2 Winograd accuracy analysis . . . 59
4.5.3 Winograd transformation optimization results . . . 61
4.5.4 Performance portability analysis . . . 61
4.6 Related work . . . 67
4.7 Discussion . . . 68
4.7.1 Design trade-offs . . . 69
4.7.2 Generalization and extensibility . . . 70
4.8 Conclusion . . . 71

5 Performance scalability: An adaptive and scalable neural architecture search platform 73
5.1 Motivation and background . . . 74
5.1.1 Quantitative analysis of distributed DL training . . . 74
5.1.2 The necessity of DL-aware scheduling . . . 74
5.1.3 Dealing with large deep-learning models . . . 76
5.1.4 A holistic approach for network design and training . . . 76
5.2 Scalability via DL-aware scheduling . . . 76
5.2.1 Job-level scheduling . . . 77
5.2.2 Cluster-level scheduling . . . 78
5.3 Scalability via network pruning . . . 79
5.3.1 Computational graph coarsening . . . 79
5.3.2 Automated graph encoder-decoder . . . 80
5.3.3 Network pruning using reinforcement learning . . . 82
5.4 Experimental results . . . 83
5.4.1 Evaluating the DL-aware job scheduler . . . 83

5.4.2 The impact of adaptive batch size tuning on NAS 84
5.4.3 Evaluating the network pruning 86
5.5 Related work 91
5.5.1 Hyper-parameter tuning 91
5.5.2 DL-specific job schedulers 91
5.5.3 DNN model compression and acceleration 92
5.6 Discussion 93
5.6.1 Distributed gradient synchronization method 93
5.6.2 The real benefit of adaptive batch size tuning 94
5.6.3 Further insights on network pruning using RL 94
5.7 Conclusion 95

6 Conclusion and outlook 97
6.1 Summary of contributions 97
6.2 Outlook 98
6.2.1 Extended application domain 99
6.2.2 Supporting other hardware 99
6.2.3 Full-stack AI 99
6.3 Final remarks 99

List of Figures

1.1 The history and hype cycles of artificial intelligence along with the rise of deep-learning models. 2
1.2 Design concerns for design and deployment of AI applications. 3
1.3 Horizontal and vertical scaling. 6

2.1 Communicating memory accesses to a single memory location. Red box: true communication. Blue circle: communication reuse. 10
2.2 Sample communication patterns found in various applications in SPLASH benchmark suite. 11
2.3 The workflow of the thread communication analysis. 12
2.4 Proposed signature memory architecture. (a) read signature uses two-level approach, (b) write signature uses regular signature memory. 14
2.5 The comparison of (a) true communication and (b) communication reuse of function `INTERF()` in the benchmark application `water_nsquared`. 15
2.6 Computing CRD for a sample 3D communication matrix. Dark cells denote reused communication. 16
2.7 A sample CRD histogram. Max and min cutoffs are placed depending on the cache size. . . 17
2.8 Communication analysis results for `lu_ncb` and `lu_cb`. 20
2.9 Communication analysis results for `ocean_ncp` and `ocean_cp`. 21
2.10 Communication analysis results for `water_nsquared` and `water_spatial`. 22
2.11 The effect of two input sizes (`simdev` and `simsmall`) on CRD histograms for the applications lu, ocean, and water. 24
2.12 Popular types of task/thread communications categorized into point-to-point and collective communications. 26
2.13 The visualization of point-to-point and collective communication patterns identified in benchmark applications. 27

3.1 From scalars to tensors. Tensors are generalization of matrices. 30
3.2 An illustration of neural networks. (a) Layered structure of neural networks. (b) The structure of a neuron. 31
3.3 Deploying deep-learning models on a diverse set of edge devices. 33
3.4 A typical convolutional neural network. 35
3.5 An illustration of popular activation functions used in ConvNets. 36
3.6 An illustration of a convolution operation. 37
3.7 Visual representation of the computational steps within a sample $F(2^2, 3^2)$ Winograd convolution. 40

4.1 Tradeoff between key concerns in designing ConvNets. 44
4.2 Boda framework workflow. 47
4.3 Kernel execution space for (1) CUDA, (2) OpenCL, and (3) Vulkan. 49
4.4 An SGEMM kernel implemented with CUDA, OpenCL, and Vulkan (GLSL). 50
4.5 The workflow of Boda+. 51

4.6 A trivial sample of MetaGPU code. 53
4.7 Illustration of a Winograd filter transformation being optimized prior to code generation. . 54
4.8 L1-norm error analysis for various Winograd internal tile sizes. 60
4.9 Comparing the number of arithmetic operations of each Winograd transformation, before and after optimization, where $r \in \{3, 5, 7\}, m \in [2, 10]$. Our analysis indicates that the highest arithmetic reduction can be achieved when $\alpha = 8$. 62
4.10 Comparing the runtimes of optimized and non-optimized Winograd convolutions $F(m^2, r^2)$, where $r \in \{3, 5, 7\}, m \in [2, 9]$, and the batch size $B \in \{1, 5, 20\}$. 63
4.11 The runtime comparison of kernels generated by our method and cuDNN vendor library on Nvidia GTX 1080 Ti. 64
4.12 The runtime comparison of Winograd convolutions generated using our method with cuDNN on Nvidia GTX 1080 Ti. 64
4.13 The runtime comparison of kernels generated by our method and the MIOpen vendor library on AMD Radeon RX 580. 65
4.14 The runtime comparison of Winograd convolutions with MIOpen library on AMD Radeon RX 580. 65
4.15 Vulkan performance with and without auto-tuning on Mali G71. 66
4.16 Winograd convolution performance with and without auto-tuning on Mali G71. 66
4.17 The qualitative analysis of various tensor-based libraries. 69

5.1 Adaptive DL-aware deep-learning training platform. 77
5.2 An overview of our deep-learning training scheduler. 77
5.3 The tradeoff between statistical efficiency and throughput given different batch sizes. . . . 78
5.4 The workflow of auto graph encoder-decoder model compression (AGMC). 80
5.5 A sample computational graph coarsening applied on a ResNet block. 80
5.6 Average job completion time of our scheduler compared with DRF and DL^2. 84
5.7 Average job completion time for different number of GPU workers with and without adaptive batch size. 84
5.8 GPU utilization of two NVIDIA GeForce GTX 1080 Ti's. 85
5.9 Adaptive batch size tuning versus fixed batch size. 85
5.10 Adaptive batch size tuning of our method (no-threshold) versus Pollux (with threshold). . 86
5.11 Comparing the pruning stability of our method across different layers with random search on ResNet-20/56. 87
5.12 Validation accuracy comparison of random search and AGMC on ResNet-56 under different FLOPs. 88
5.13 An error-rate comparison for individual AMC layer embedding, overall AMC, and AGMC layer embedding. Our method has achieved roughly $2\times$ less error rate. 89
5.14 Adaptive batch size tuning versus using maximum possible batch size. 94
5.15 Accuracy recovery after different fine-tuning epochs. 95

6.1 The components of a full-stack AI platform. 100

List of Tables

2.1 The list of optimization types along with some examples categorized into coding and runtime optimizations. . . . 19
2.2 Detailed results of communication bottleneck analysis for a subset of the Rodinia [72] benchmarks. . . . 23

4.1 A comparison of the terminology used in CUDA, OpenCL, and Vulkan. . . . 49
4.2 The list of pre-defined keywords in the kernel body alongside their corresponding value within each target API. . . . 53
4.3 Tuning parameters for Winograd convolutions. . . . 58
4.4 Experimental setup. . . . 59
4.5 Lines-of-code comparison for different convolution implementations alongside computed effort metric. . . . 59
4.6 Polynomial points selected by our method alongside their relative error. . . . 60
4.7 List of sample convolutions for testing normal convolution kernels. . . . 72
4.8 List of sample convolutions for testing Winograd kernels. . . . 72

5.1 Detailed comparison of AGMC with other methods. . . . 90

1 Introduction

We begin this dissertation by providing an introduction to the main concepts that are necessary to understand the contributions. First, we describe the concept behind data-intensive applications and their essential characteristics, which are relevant to our work. Particularly, we focus on a subset of data-intensive applications, namely deep neural networks. We provide a brief history of such programs and the main design concerns that motivated us to work on them. After defining our target application domain, we briefly summarize performance engineering and its constituent methods that we use in our contributions. Specifically, we explain a trilogy of performance engineering methods that span from "performance profiling" and "performance portability" to "performance scalability". Last but not least, we conclude this chapter with an outline of this dissertation's structure.

1.1 Data-intensive applications

In essence, data-intensive applications are a subset of computer programs that deal with large volumes of data. They are typically parallelized using the data parallelism paradigm to benefit from multi-/many-core processors. Additionally, such applications also involve compute-intensive characteristics [1]. Thus, an efficient data-intensive program requires optimizations related to both memory and computations.

Big-data applications such as data analytics, web crawling, social network analysis, and weather prediction are among the well-known examples of data-intensive applications. In fact, a variety of system architectures have been developed for such applications, including parallel and distributed relational database management systems, MapReduce [2] programming model, and Apache Hadoop [3].

A recently emerged workload is large-scale machine learning and particularly deep-learning applications. Deep neural networks need to be trained with large datasets multiple times to reach to an acceptable accuracy. Moreover, the inference phase is often deemed to operate in real-time, requiring a highly efficient implementation to handle large amounts of data in a fraction of seconds. Thus, performance optimization plays a vital role in realizing the potentials of such applications. In this dissertation, we primarily focus on deep learning as the main example of data-intensive applications.

1.1.1 Deep-learning applications

Artificial neural networks and their more powerful variant, deep neural networks, have a long history that dates even back to the time that early computers became available. Former breakthroughs in the AI research led to several hypes and high expectations from the community. Nonetheless, regardless of all the ambitious goals and efforts to imitate the functionality of a human brain using neural networks, this field of research has experienced two extensive stagnations before 2012 (see Figure 1.1). The root cause was the prohibitive computational demand of such networks. Given that traditional machine-learning methods could provide better accuracy with lower computing complexity, neural networks lost the momentum and remained untended for a while. Other reasons that prevented neural networks from flourishing were insufficient labeled data, unexplainable outcomes, and extensive developing effort.

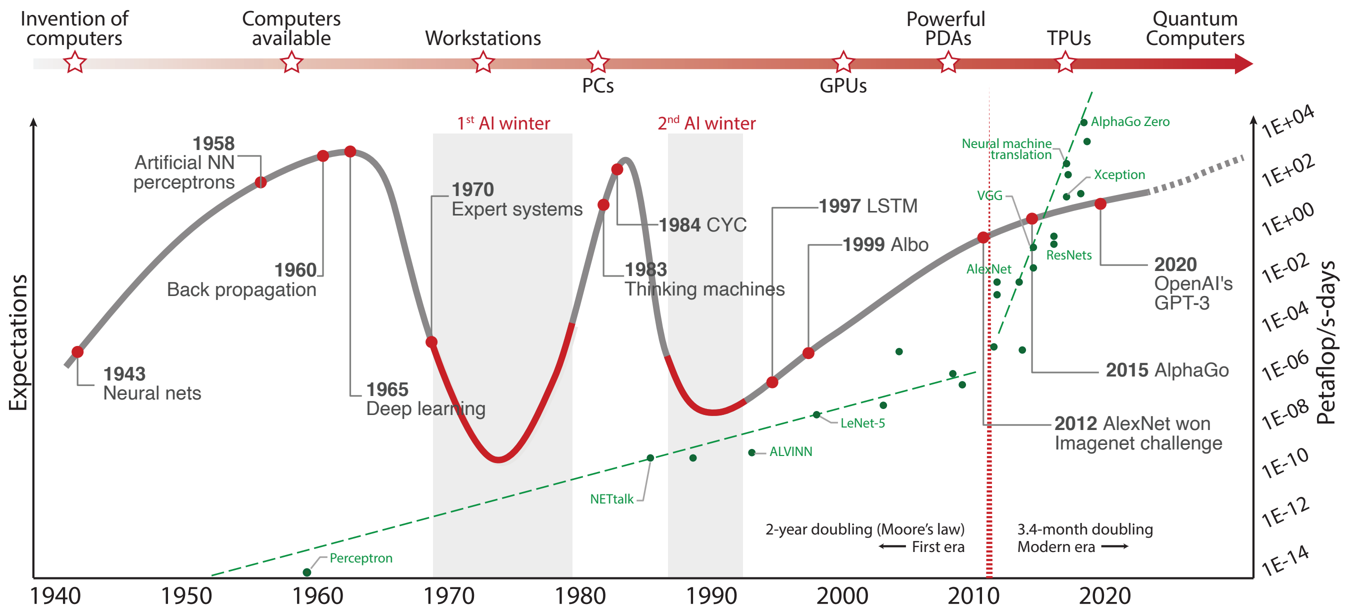


Figure 1.1: The history and hype cycles of artificial intelligence along with the rise of deep-learning models [9]. Before 2012, the amount of compute used in the largest AI training doubled every 2-years, similar to the Moore's law. Since 2012, this amount is increasing exponentially with a 3.4-month doubling time.

With the ubiquity of high-throughput GPUs, the modern era began in 2012 with the emergence of the first real deep-learning models trained using GPUs. Previously, it was uncommon to perform training on GPUs. Thus, the amount of computation used in the largest AI training doubled every 2-years, similar to Moore's law. Since 2012, this amount is increasing with a faster pace—3.4-months doubling time [4] (see Figure 1.1). From 2012 to 2014, training jobs were running at most on eight GPUs. However, starting from 2014, large-scale training became possible with the help of GPU clusters and additional algorithmic improvements, enabling the training to scale up to hundreds of GPUs nodes. As a result, deep learning and its popular technique, convolutional neural networks (ConvNets), are now the mainstream machine-learning method for a wide variety of computer-vision tasks, including object detection [5], image segmentation [6], and video classification [7]. In some cases, neural networks can now even beat human performance. Such a wide and growing use-case in both academia and industry has been made possible by three enabling factors [8]: (1) more efficient end-to-end deep-learning algorithms, (2) availability of massively parallel compute platforms such as GPUs, and (3) large labeled datasets made available through crowd-sourcing. In this work, we mainly focus on the second factor and attempt to enable ConvNets to efficiently harness the full potentials of available computing resources.

The computations in deep-learning applications are divided into two separate phases: training and inference. The training phase is often highly parallelizable and scalable. It heavily uses computational resources, ranging from a high-end GPU to a large cluster of machines. Although such resources are readily available to the industry and product developers, not enough attention is being paid to the orchestration of AI jobs. Hence, AI projects have become the source of bottlenecks in research and development. As a remedy, an AI-tailored job scheduler can immensely enhance the speed and scalability of deep-learning training jobs.

Once the training of a ConvNet converges to an optimum level, the obtained model should be deployed

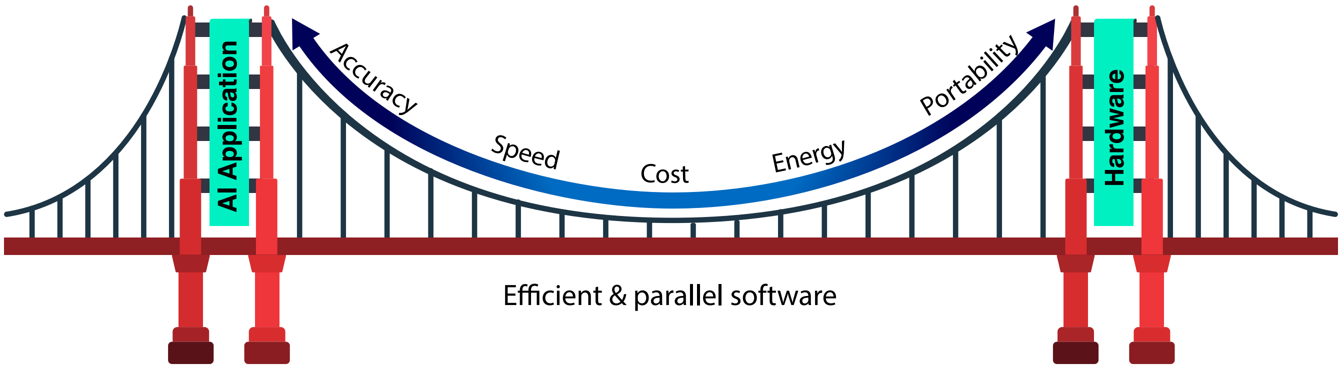


Figure 1.2: Efficient and parallel software solution acting as a bridge between AI applications and hardware (inspired by "A view from Berkeley" report [11] and Golden Gate Bridge). As a remedy for the existing tension between the two ends, we need to strike a balance between accuracy, speed, cost, energy consumption, and performance portability.

on the target hardware platform. However, due to several design concerns, high-end processors cannot be directly used on edge devices. Accuracy, speed, cost, energy consumption, and portability are among the main concerns during the development and deployment of AI models on target devices [10]. However, the contradictory nature of these concerns often demands extra attention from the developers, as they involve different constraints, priorities, and difficulties. For instance, for an AI model that is supposed to be deployed on a mobile device with limited computing power, energy consumption and speed will have a higher priority than one or more of the concerns mentioned earlier.

Nowadays, various efficient, low-cost embedded processors, including special-purpose processors for AI-related tasks, exist in the market. GraphCore, Huawei, Intel Mobileye, Renesas, and Texas Instruments are among the companies that have built various AI SoCs[1], each bundled with different hardware specifications, offering different performance levels. Nonetheless, deploying ConvNet models on such devices and exploiting the utmost performance is a cumbersome task, often demanding iterative rounds of tuning and code specialization. Therefore, a single recipe for optimization would not lead to the highest runtime efficiency on all processors. Furthermore, ConvNet models have to be reshaped to match hardware constraints. Such issues constitute a significant obstacle for AI developments in the industry sector. As a real example, autonomous vehicles with the highest level of automation often demand the concurrent execution of multiple AI jobs. Furthermore, they heavily rely on the real-time data gathered by many input sensors, all of which require massive computing power to perform instant analysis and appropriate maneuvers.

Various application domains such as big data perform their computation in a layered system and employ runtime frameworks such as Hadoop and Apache Spark for efficiency. Similarly, the computation behind deep learning can be embodied in layered software systems. As a result, such systems act as a unified software solution, bridging the gap between the high-level application implementation and complex hardware platform by striking a balance between all design concerns mentioned earlier (see Figure 1.2). In Chapter 4, we provide our bridge solution to realize the design concerns of deploying ConvNets on various GPU platforms.

[1]System on a Chip

1.2 Performance engineering

In many application domains, computing resources are limited and costly. Hence, different levels of optimization on code, architecture, and runtime scheduling are required to enhance the overall performance. Even if computing resources are readily available, drawing less energy with an eye towards more environment-friendly code is essential. To this end, a well-engineered process for performance analysis and tuning is largely beneficial to attain the expected performance. Such a systematic process is called *performance engineering* [12, 13]. Particularly, it is often regarded as the combination of different techniques applied to a given application to enhance its non-functional properties such as performance, throughput, energy consumption, memory usage, and performance portability. As mentioned earlier, in this dissertation, we introduce three techniques to build our performance engineering approach. These methods are not necessarily interdependent as each can be used separately for a specific use. In the following, we provide a brief explanation for each method.

1.2.1 Performance profiling

Monitoring and collecting performance data from an application to analyze its behavior is called *performance profiling*. Such data can later be analyzed to pinpoint potential performance bottlenecks. Upon the identification of poorly performing code regions, the programmer can resolve performance issues by applying appropriate code optimizations, if possible. Performance bottlenecks can occur due to various reasons such as inefficient algorithms, poor data locality, and inappropriate thread communications (assuming that the application is parallelized). Various methods with different overhead and profiling granularities exist for performance profiling. Two well-known category of methods are (1) offline/simulation-based profiling and (2) dynamic instrumentation. In the following, we will describe each category and enumerate a few notable communication profiling tools.

Offline/simulation-based profiling. By imitating the real environment, we can monitor the runtime behavior of a given application in a controlled fashion. Such methods either run the target application in a controlled sandbox (e.g., by using Virtutech's Simics simulator [14]) with many software probes attached to record the changes, or by using debugging helper libraries available in parallel programming frameworks (e.g., IPM [15] in MPI). For instance, using the former method, we can trace all memory accesses to analyze the communication pattern [16, 17]. However, despite compressing the analysis results, they produce extra-large output files, more than 100GB for a moderate input size [16]. Following the latter approach, we can employ IPM [18] to collect MPI communications between processors [19, 20, 21, 22, 15]. Such efforts, however, are mainly designed for distributed-memory applications and do not take shared-memory systems' characteristics into account. Additionally, they have high memory overhead since they use a 128-bit signature size for each MPI call. Therefore, besides having high execution runtime, simulation-based profiling do not seem to fit for on-the-fly analysis.

Dynamic instrumentation. By inserting a predefined function before every target instruction in the program, we can extract various runtime metrics. These API calls are invoked at runtime and allow us to record various performance data such as timestamps, memory accesses, and instruction counts. Well-known instrumentation tools are LLVM [23], Intel Pin [24], DynamoRIO [25], and Valgrind [26]. For example, DiscoPoP [27] uses LLVM instrumentation to analyze data dependences to identify potential parallelization opportunities. Another project called MACPO [28] uses LLVM to instrument source codes and analyze their data structures. Helgrind [29] and Helgrind+ [30, 31] are based on Valgrind to detect synchronization errors. They utilize the shadow memory approach with 32 and 64 bits shadow values, respectively. SD3 [32] is another profiler for data dependency analysis of sequential programs, which reduces space overhead of tracing memory accesses by compressing strided accesses using a finite state machine.

In addition to the code instrumentation methods, a number of tools rely on hardware performance counters [33, 34, 35, 36, 37] to monitor the performance counters available in the underlying hardware to collect the required information. They have much less overhead in comparison with the previous methods. As a result, they are more suitable for performing rapid performance analysis, however, with lower accuracy and the need for operating system tweaks. For instance, this technique can only indirectly estimate thread communications, leading to inaccurate results [38]. An alternative approach to overcome this issue is using translation lookaside buffers [38]. Although this approach can only be used for thread migration in shared-memory systems, it could be a great start point for future researches.

Given the pros and cons of each method described above, we opted for code instrumentation profiling and present our solution for communication profiling in Chapter 2.

1.2.2 Performance portability

Given the wide range of available computing platforms such as CPUs, GPUs, and FPGAs, attaining high utilization and efficiency on all platforms with the same code base is challenging. Even within the same line of hardware, we have a magnitude of different configurations for computing cores, memory hierarchy, and bandwidth. Such a task becomes even more complicated when new computing platforms emerge in the market, such as Google's tensor processing units (TPU), GraphCore, and Intel's Nervana platforms. All these varieties make the code development and efficient deployment on the end device an extremely cumbersome task since each hardware requires a specific set of optimizations to be applied to the code. Furthermore, each optimization method might have a different set of tuning parameters.

Various tools and methods have been introduced to the HPC community by experts to write a single code and run it on various platforms, such as Kokkos [39], Raja [40], and SYCL [41]. However, they are designed for general-purpose applications and do not necessarily consider the unique requirements of AI applications. All these tools are based on a concept called performance portability. Since there is no standard and universally accepted definition for such a term, we describe our point of view. We define *performance portability* as "to achieve a performance close to best-known vendor runtime on each platform with a single source code". This statement implies that a code is only performance portable if the programmer does not have to rewrite the program and the resulting performance is close to vendor libraries (usually within 20% of the best achievable runtime). Based on our definition for performance portability, in Chapter 4, we introduce a library for generating efficient and portable ConvNet kernels.

1.2.3 Performance scalability

Generally, scalability is a property that identifies the ability of a system to efficiently deal with growing amount of data if more computing resources are available. Often, scaling the resources is performed in two different dimensions, namely horizontal (a.k.a. "scaling out") or vertical (a.k.a. "scaling up"). Although both scaling schemes involve provisioning more computing resources, they are substantially different in implementation and attainable performance. As depicted in Figure 1.3, horizontal scaling works by adding more compute nodes to the pool of resources. In contrast, vertical scaling adds more computing power, such as CPU, GPU, and memory, to an existing machine. One key difference between the two is that horizontal scaling expects a high degree of parallelism so that each node can execute a portion of computations. Thus,

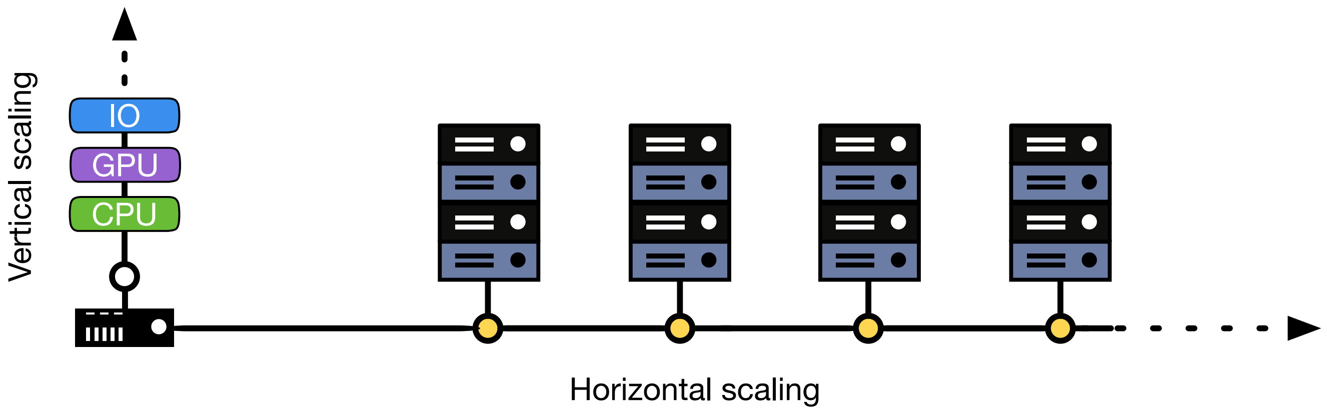


Figure 1.3: Horizontal and vertical scaling.

it involves additional effort to efficiently parallelize and prepare the code for such a scaling scheme. On the contrary, vertical scaling is more straightforward, as the logic stays the same, yet the program runs on a more powerful device.

Within the context of this dissertation, we studied the scalability degree of deep neural networks primarily at the training phase. Deep-learning training is a dominant workload in many HPC and cloud-computing centers, as such jobs are highly resource-/data-intensive, requiring access to large-scale computing resources to complete within a reasonable amount of time. To meet such an extensive resource demand, dedicated or shared clusters are often used, in which a job scheduler manages resource sharing between all the training jobs. From the scalability perspective, the job scheduler is responsible for allocating the right number of resources at the beginning and throughout the entire execution. In addition to introducing a DL-specific scheduler, we present a novel model compression solution for scaling the performance of deep-learning models in Chapter 5.

1.3 Contributions

The contributions of this work can be summarized as a class of methods for performance analysis and accelerating recently emerged HPC workloads. By focusing on a specific range of data-intensive applications (i.e., deep-learning workloads), we demonstrated that effective acceleration of such use cases requires a combination of different methods. A brief description of our threefold performance engineering methods is provided below:

Performance profiling: Unveiling thread communication bottlenecks. The first contribution is a novel performance profiling method to identify potential communication bottlenecks from data sharing among threads. Using our profiler, we observed that the data shared between a pair of threads should be reused within a reasonable distance to preserve data locality. Thus, we propose new hardware-independent metrics to characterize thread communication and suggest appropriate code optimizations.

Performance portability: Efficient and performance-portable ConvNet deployment. Our second contribution is a framework for the automatic generation of efficient and performance-portable convolution kernels for various GPU platforms that support CUDA, OpenCL, or Vulkan. By relying on a synergy of meta-programming, symbolic execution, and auto-tuning, we enabled the specialization of a challenging ConvNet operation named Winograd convolution. Achieving runtimes close to vendor deep-learning libraries with minimum programming effort confirmed the performance portability of our approach.

Performance scalability: An adaptive and scalable neural architecture search platform. Lastly, we developed a specialized training platform equipped with a novel topology-aware network pruning algorithm to enable rapid and scalable network training, neural architecture search, and model compression. As a result, an AI model can be easily scaled to a multitude of computing nodes, leading to faster model design with less operating costs.

1.4 Structure of this dissertation

The remainder of this dissertation is structured as follows. We start by reviewing the fundamentals of deep-learning applications in Chapter 3. Such programs are the main focus of our work, and understanding them is the prerequisite for comprehending the rest of this thesis. Our performance engineering trilogy, which constructs the main contributions, will follow in Chapters 2 to 5. First, we present our thread communication profiler in Chapter 2. Next, Chapter 4 describes our proposed solution for efficient and performance portable deployment of convolutional neural networks. As our last contribution, Chapter 5 presents a scalable training platform for deep-learning models. Finally, Chapter 6 concludes this document by highlighting the potentials of this work and providing a handful of future research directions.

1.5 Statement of originality

The content of this dissertation is primarily based on four relevant peer-reviewed manuscripts that are published in parallel computing, computer systems, and AI conferences, in which I have contributed as the main or secondary author. The manuscripts were published under the supervision of Prof. Dr. Felix Wolf (Department of Computer Science, Technical University of Darmstadt) and Prof. Dr. Ali Jannesari (Department of Computer Science, Iowa State University). In the following, we provide the complete bibliographic information of the manuscripts along with an explanation of their contribution to this dissertation.

- **Arya Mazaheri**, Felix Wolf, Ali Jannesari: Unveiling Thread Communication Bottlenecks Using Hardware-Independent Metrics. *In Proc. of the 47th International Conference on Parallel Processing (ICPP)*, Eugene, OR, USA, pages 6:1–6:10. ACM, August 2018 [42].

  ↪ The content of this paper appears mostly in verbatim text in Chapter 2. As the main author, I did the brainstorming, implementation, experiments, and wrote the manuscript. The feedback and supervision that I received from my co-authors greatly improved the quality of the paper.

- **Arya Mazaheri**, Johannes Schulte, Matthew Moskewicz, Felix Wolf, Ali Jannesari: Enhancing the Programmability and Performance Portability of GPU Tensor Operations. *In Proc. of the 25th Euro-Par Conference*, Göttingen, Germany, volume 11725 of Lecture Notes in Computer Science, pages 213–226, Springer, August 2019, (best paper award) [43].

  ↪ A part of Chapter 4 is based on the content of this paper, from which most of the text is taken as verbatim. My former undergraduate student, Johannes Schulte, participated in implementing the Vulkan backend [44]. I, as the first author, also participated in implementing the idea such as MetaGPU, conducted all the experiments on various GPU platforms, and wrote the manuscript. Dr. Moskewicz agreed to collaborate with us in this project, as the idea of this paper was based on his former work. We benefited a lot from his expertise to define a clear goal and a right experiment strategy for this project.

- **Arya Mazaheri**, Tim Beringer, Matthew Moskewicz, Felix Wolf, Ali Jannesari: Accelerating Winograd Convolutions using Symbolic Computation and Meta-programming. *In Proc. of the 15th EuroSys Conference*, Heraklion, Crete, Greece, pages 1–14, ACM, April 2020 [45].
  ↪ Similar to the previous paper, Chapter 4 has also taken a large portion of this paper as verbatim. Dr. Moskewicz proposed the idea of implementing an efficient Winograd and provided several constructive feedback. My former undergraduate student at that time, Tim Beringer, helped me in implementing a GPU version of the Winograd convolution in the Boda framework [46]. I, as the first author, proposed additional key ideas, implemented the symbolic computation method, conducted the experiments, and wrote the manuscript.

- Sixing Yu, **Arya Mazaheri**, Ali Jannesari: Auto Graph Encoder-Decoder for Neural Network Pruning. *In Proc. of the International Conference on Computer Vision*, Montreal, Canada, IEEE, October 2021 [47].
  ↪ The network pruning method that is described in Chapter 5 is derived from this paper. I, as the secondary author, participated in the brainstorming, conducting the experiments, and writing the manuscript.

In addition to the publications above, Chapter 5 contains a section related to a scalable training platform, which is derived from the Master thesis of my former graduate student, Tim Beringer [48]. At the moment of writing this dissertation, this work is still in progress.

2 Performance profiling: Unveiling communication bottlenecks

The prevalence of multi-core processors has made parallel programming and, therefore, performance analysis/optimization of parallel applications a must for exploiting utmost efficiency. However, despite the emergence of various parallel programming models, libraries, and tools to make software parallelization easier, the majority of programmers do not yet have detailed insights into performance bottlenecks and the appropriate solutions [49]. Thus, parallel computation tasks are often not efficiently developed. On the one hand, this could be due to undiscovered parallelism opportunities or inappropriate workload assignment to available processors. On the other hand, low efficiency and scalability can be explained by ineffective communications among the threads.

Thread communication plays an essential role in parallel applications, often not thoroughly investigated during performance-bottleneck diagnosis. Because memory is much slower than processors, the complete characterization of the memory access patterns of important code regions is critical for precise performance diagnosis. This turns out to be more vital in multi-core systems, where data sharing among cores is often performed in last-level caches [50]. Currently, the programmers' approach to performance tuning is manual code restructuring and applying compiler optimizations for each target platform. In general, however, performance optimization needs to improve data-cache locality and reduce communication bottlenecks [51].

The expense of communications among tasks is often correlated with the amount of communication and the degree of overlap with computations. Furthermore, performance depends on the size and sharing configuration of the underlying cache. Various studies have investigated the sharing behavior in parallel applications, but primarily only for a single cache size [52]. Therefore, they cannot tell how changes in cache size, configuration, or even thread allocation policies might ultimately affect performance. Multi-core reuse-distance analysis [53, 54] is a promising approach to assessing parallel applications' locality and their cache effectiveness. However, it fails to provide a detailed overview of communication among threads. Furthermore, it suffers from high complexity and error-prone results due to thread interleaving. The locality results are also often reported for the whole program. Therefore, we cannot focus on optimizing data-sharing patterns and thread communication.

In this chapter, we present our early work on performance analysis and also the first part of our performance-engineering trilogy. We conducted a comprehensive study using performance profiling to realize the most common pitfalls and sources of bottlenecks in shared-memory parallel applications. Additionally, we contemplated the applicable optimization methods for improving the efficiency of parallel programs, given their communication patterns. Our profiling method is generic and can be applied to any shared-memory application. The results and findings of this work paved the path to perform targeted code specialization and performance optimization.

2.1 Background and motivation

Communication in parallel programming models is either explicit or implicit, depending on how threads share data. For example, distributed-memory programming models like MPI follow explicit communication

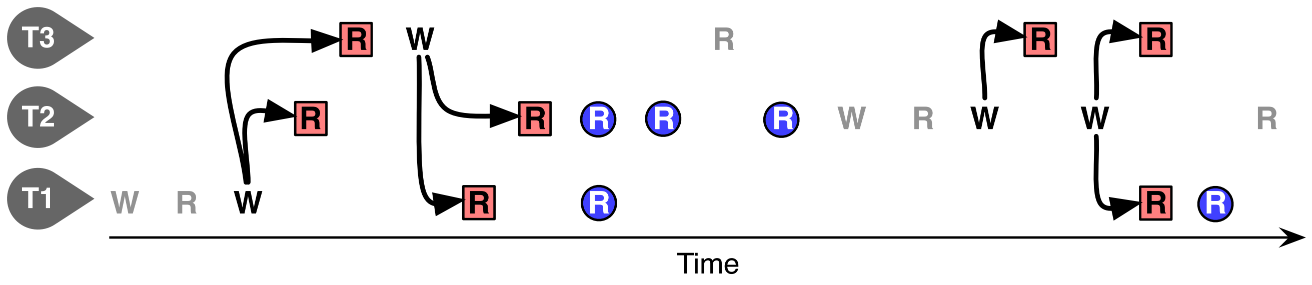


Figure 2.1: Communicating memory accesses to a single memory location. Red box: true communication. Blue circle: communication reuse.

through `send()` and `receive()` API calls. The same process happens in GPU communications libraries, such as Nvidia's collective communication library (NCCL, pronounced "Nickel") [55], which provides topology-aware inter-GPU communication primitives. Within these libraries, we often have access to collective communication primitives, such as AllReduce, Broadcast, Reduce, AllGather, and ReduceScatter. Undeniably, they also provide point-to-point send/receive communication, allowing scatter, gather, or all-to-all operations.

In shared-memory applications, however, exchanging data is implicit, and it is mostly accomplished through memory accesses to shared variables [56]. Such a method, implies different communication patterns [16] compared with distributed-memory applications, imposing additional irregularity and complexity on data sharing. For the same reason, we are interested in finding an appropriate method for detecting thread communications and discovering potential bottlenecks.

2.1.1 Communication in shared memory systems

We define a *communication event* as two memory accesses from different threads to the same memory address with a specific pattern. Four different combinations of memory accesses (RaR, RaW, WaR, and WaW) can occur, depending on whether the data is read or written by each thread. However, only the RaW pattern implicates true communication. No thread can be declared as sender/receiver due to performing the same operation in the other cases. In this case, one thread writes data (sender) which is then read by another thread (receiver). To avoid redundant communication, we only consider the first read after a write as a communication. Other read accesses to the exact location will be considered as reuse. Figure 2.1 illustrates an example of distinguishing between true communication and its reuse for a single memory being accessed by three threads.

Identified communication events can be represented by a directed acyclic graph (DAG), in which each node represents a thread, and edges denote the number of communication events between each pair of threads. Such a graph is often visualized as an adjacency matrix, which is called *communication matrix* or *communication pattern* [57, 56]. Each cell of the matrix contains the number of communication events for a given pair of threads. The diagonal of the matrix is always zero, as memory accesses by the same thread do not imply any communication. Figure 2.2 illustrates sample communication patterns that we found within the applications in the SPLASH benchmark suite.

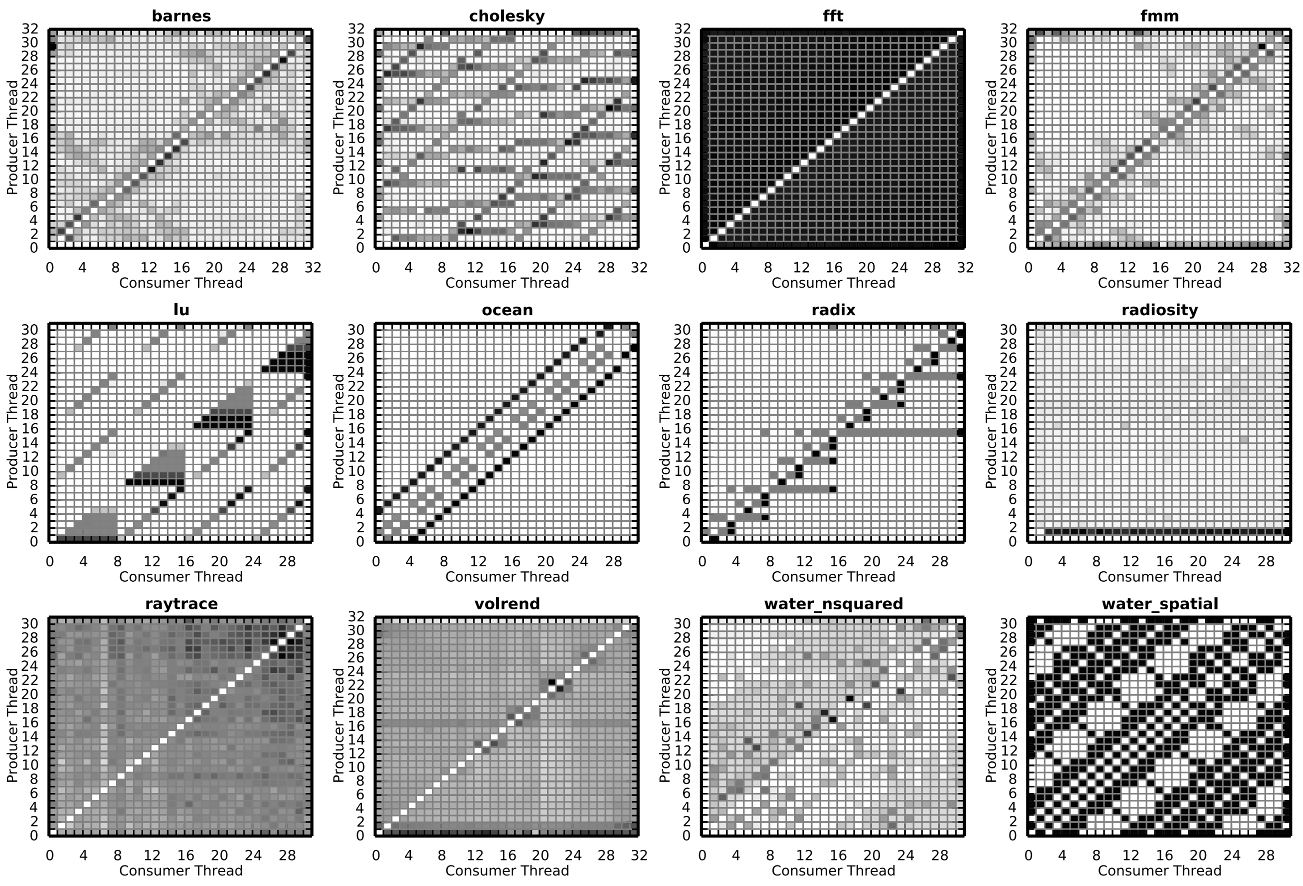


Figure 2.2: Sample communication patterns found in various applications in SPLASH benchmark suite.

2.1.2 Reuse distance analysis

Reuse distance has long been a hardware-independent metric for evaluating data reuse in programs [58]. The reuse distance of a given reference to element x is the number of distinct data elements accessed between two consecutive uses of x. Data granularity could be anything from pages, cache lines, memory words, or instructions. Reuse distance is typically used for predicting the cache hit ratio of a fully associative LRU cache with N one-word blocks, in which data accesses with reuse distance of N or less would hit. The reuse distance distribution is typically represented as a histogram diagram (a.k.a. locality signature) and shows the overall program data locality (see Figure 2.7). Such an analysis can model data locality for all cache sizes and can later be used as a reference for performance optimizations [58, 59, 60].

In multi-threaded applications, however, such an analysis is no longer hardware-independent because threads interact with each other and memory accesses might interleave [53]. Various researchers proposed an alternative method for computing concurrent reuse distance and private reuse distance for shared and private caches, respectively [54, 61, 62]. Such methods are based on statistical modeling methods and consider specific parallel regions, like loops, due to the high code divergence of task parallelism. Furthermore, these methods consider all memory accesses and do not provide specific insights into communication events and synchronization. Thus, we aim to address these issues and propose a locality-analysis method to study communication in shared-memory applications.

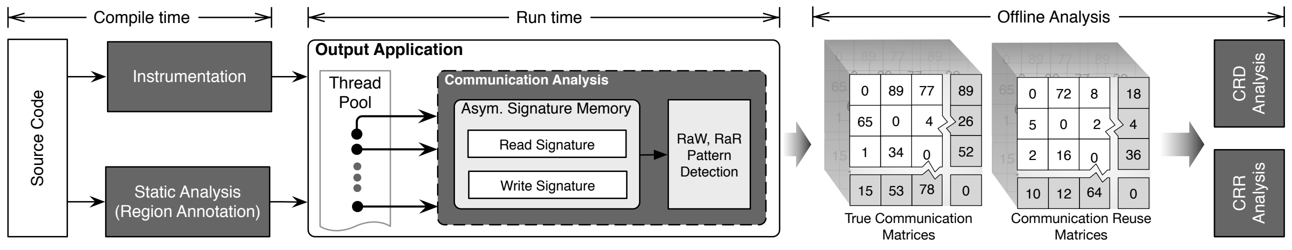


Figure 2.3: The workflow of the thread communication analysis.

2.1.3 DiscoPoP profiler

DiscoPoP [63, 27] is an LLVM-based dynamic dependence profiler mainly designed for detecting dependences inside sequential and multi-threaded programs. It detects write-after-read (WAR), read-after-write (RAW), and read-after-read (RAR) dependences among program instructions with the aid of code instrumentation. The main issue regarding software profiling is the excessive runtime overhead and memory consumption, preventing them from being used widely. However, DiscoPoP has succeeded in overcoming this challenge by employing software signatures for recording memory access history. Therefore, program profiling with less than 500MB of memory and 86× average slowdown has been made possible. A noteworthy feature about DiscoPoP is that its components can be easily extended to add required functionalities. In this work, we exploited this feature to implement our communication pattern profiler.

2.2 Characterizing the communication behavior of parallel programs

We argue that studying the communication pattern of parallel programs can characterize them and ultimately be used for performance bottleneck analysis. Each communication pattern has a unique data-sharing topology among processors/threads [19], enabling us to discern them quickly and apply relevant optimizations. Although various methods already exist for extracting communication patterns, they are either only directed toward distributed-memory applications [21, 64, 65] or not comprehensive enough to be used for memory performance characterization [56, 51]. They mainly produce a single communication pattern for the entire program and neglect the dynamic behavior of a parallel application. Additionally, they generate a simple communication topology, failing to provide further insights into improve data locality.

To address the shortcomings mentioned above, we developed a method for thread communication analysis, which consists of three different phases: (1) compile-time, (2) runtime, and (3) offline post-analysis. Figure 2.3 depicts a high-level overview along with the execution flow. In the following, we will explain the main components.

2.2.1 Instrumentation and profiling

In order to produce a nested structure of communication matrices in potential hotspots of the target program, we have devised a simple static analysis module. It analyzes the program and annotates each loop with a unique identifier (UID) using LLVM metadata nodes. If the instrumented memory access is inside a loop, the UID of the parent loop is fed into the pattern detection for further analysis.

To determine the communication pattern, we opted for the code instrumentation approach. As previously discussed in Section 1.2.1, various methods and frameworks are available for instrumenting programs and capturing inter-thread data dependencies. However, due to the following reasons, we chose compiler-based instrumentation over simulation, binary instrumentation, and hardware counter analysis:

- Compile-time instrumentation provides more flexibility and freedom in terms of code instrumentation with the aid of intermediate representation (IR). For instance, only specific functions, operations, or data structures could be selected for instrumentation. In contrast, other methods are unable to detect loop structures, boundaries, and data-structure-related operations [28].
- Compile-time instrumentation enables us to provide an on-the-fly analysis feature, unlike simulation methods. However, such a feature comes with a cost. Typically, instrumentation degrades the runtime performance by at least $80\times \sim 100\times$, yet it still outperforms the hardware analysis approach with regard to accuracy.
- Binary program that is generated after compile-time instrumentation could run natively on the target architecture and benefit from hardware-specific optimizations. On the contrary, simulators are often not comprehensive enough to support all bells and whistles of high-end processors [28].

Given the reasons above, we implemented our method with the help of the DiscoPoP dependence profiler [63] to obtain the required data for extracting communication patterns and communication reuse distance. The input is the target program, containing all memory accesses. DiscoPoP detects data dependences among program instructions using LLVM code instrumentation. We tweaked the instrumentation module in DiscoPoP to support POSIX threads (a.k.a. pthread) and OpenMP parallelization models and instrumented each memory access with its access type, memory address, executing thread ID, region information, and variable size. We also annotated each code region to obtain fine-grained information related to each region later on. The granularity of profiling is loop and function regions in pthread applications and OpenMP regions in OpenMP applications.

As mentioned before, the main drawback of dynamic profiling, which prevents it from being widely used, is runtime and memory overhead. Pairwise dependence analysis requires a lot of time and resources to detect inter-thread dependences. Such dependences can be viewed as data conflicts since a dependence exists only when the same memory location is accessed several times in a particular order by more than one thread. Thus, to overcome the existing challenge, we utilized a customized data structure, called asymmetric software signature, for recording memory accesses history [57]. The main reason is that signature memories are typically used for determining conflicts between two sets of items, such as read and write memory accesses. The concept of this data structure is borrowed from transactional memory systems, where it provides an approximate representation of an unbounded set of elements with a bounded set of states [66]. In other words, the memory consumption is constrained to a fixed value, but the user can still adjust the maximum size. Additionally, signature memories access the stored data through hash functions with $\mathcal{O}(1)$ access time to minimize the runtime overhead of detecting conflicts. However, the output might be incorrect due to the limited memory size, similar to any hash-based data structure. In fact, the probability of false-positive results correlates with the signature size and the number of memory accesses in a given application.

Our asymmetric data structure consists of two separate signature memories. The first one is a two-level signature memory, mainly designed for the *read signature*, as we need to store the list of all threads that have accessed the corresponding memory location. It uses a fixed-length array of size n, where n is the signature size, combined with an efficient MurmurHash [67] function that maps memory addresses to array indices. We opted for this hash function because it has a much lower time complexity while having fewer collisions compared to other hash functions. The first-level array stores pointers to the second-level arrays, all of which are bloom filters. The Bloom filter [68] is a simple and space-efficient probabilistic data structure for recording and representing a set of data to perform rapid membership queries. In our case, it has been used to save the list of threads that accessed the same memory address. Figure 2.4(a) demonstrates the proposed two-level signature approach. Memory location address hashes to an element in the first array. If the element is empty, a pointer to the second array will be allocated and points to the new bloom filter.

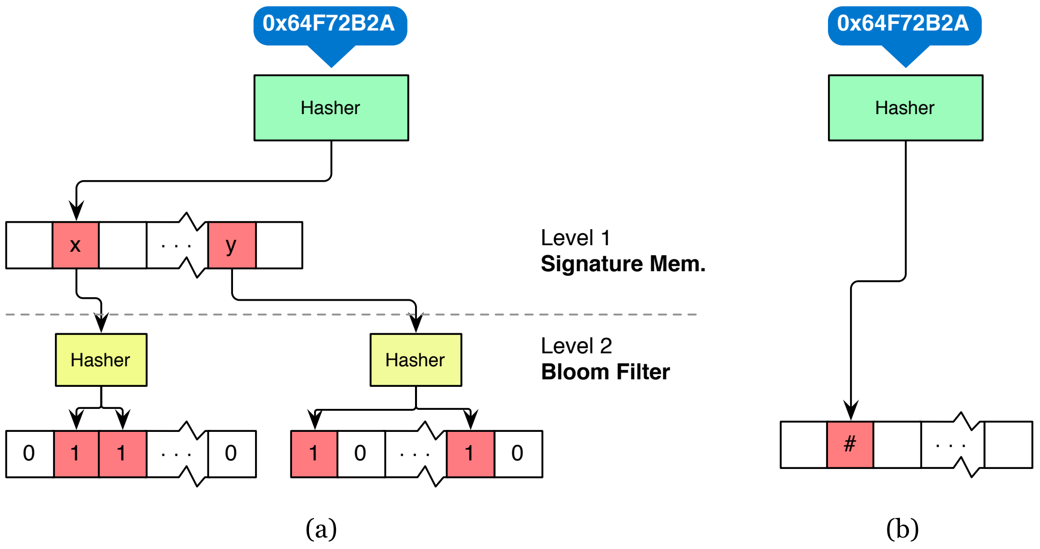


Figure 2.4: Proposed signature memory architecture. (a) read signature uses two-level approach, (b) write signature uses regular signature memory.

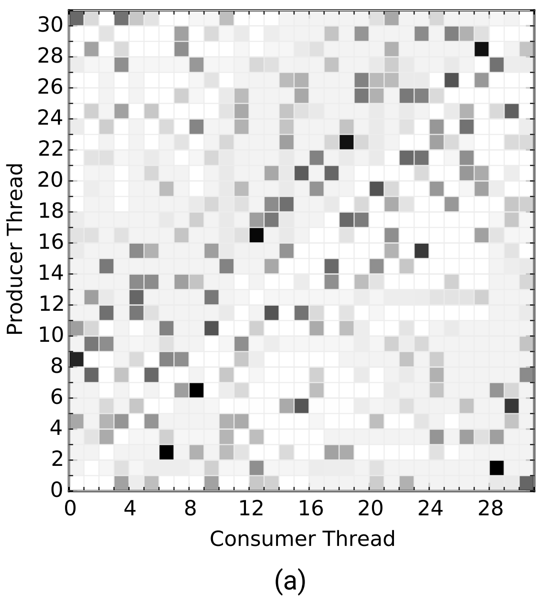
The size of signature elements and bloom filters is adjustable by the programmer to optimize for a particular program under test. However, the bloom filter has been designed so that its size does not need to be adjusted manually. The bloom filter uses a bit vector of size m, where m depends on the number of threads available in the target program. Furthermore, a linear combination of hash functions has been devised to automatically adjust the number of hash functions according to the false-positive rate required by the user. Since the maximum number of elements that are supposed to be stored in the bloom filter is limited by the number of threads that might access the same memory location, it is guaranteed that the false-positive rate does not go beyond the threshold limit.

On the other hand, as depicted in Figure 2.4(b), a plain signature memory is used for the *write signature* memory. In this case, we use a plain signature memory to store source thread IDs. Each value stored in this signature memory represents the last thread ID that accessed the corresponding memory location.

Although employing signature memories can significantly reduce the amount of memory overhead and offer fast memory lookups, they incur potential hash collisions. Defining a small signature size could lead to numerous collisions (i.e., $h(v_1) = x$ and $h(v_2) = x$ with $v_1 \neq v_2$). This will then produce inter-thread dependencies that do not actually exist ($a.k.a.$ false positive). Consequently, the accuracy of the algorithm decreases when the size of the signature decreases. Hence, picking the size of the signature shall be viewed as a trade-off between memory consumption and accuracy.

2.2.2 Communication pattern detection

We are interested in detecting both true communication events (RaW) and their reuse (RaR). We define *true communication* as a write operation on a given memory location followed by the first read of the same memory, provided that two different threads are involved. On the other hand, *communication reuse* relates to those re-reads after the initial RaW event. We distinguish between these two events because of their different effects on cache usage and memory performance. One critical assumption is that the target application is data-race free and synchronization points are used correctly. Otherwise, the gathered information would be



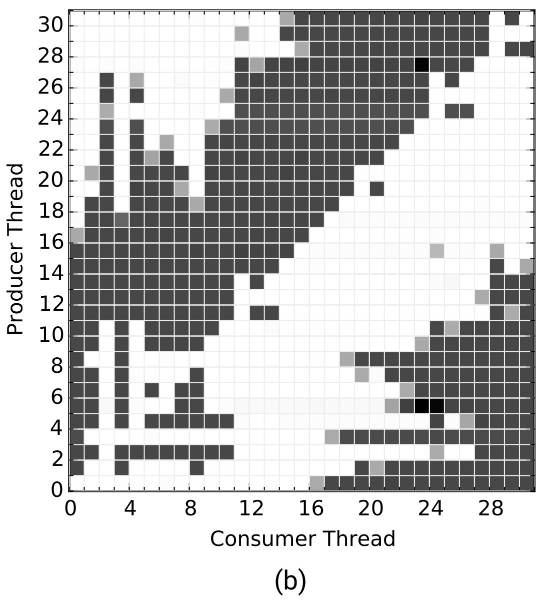


Figure 2.5: The comparison of (a) true communication and (b) communication reuse of function `INTERF()` in the benchmark application `water_nsquared`.

misleading.

The pseudocode for detecting dependences between threads with signature memories is shown in Algorithm 1. This algorithm processes memory accesses in a logical order to detect thread dependences. To achieve higher throughput and parallel analysis, we run this algorithm concurrently using the application's threads. Thus, the dependences are identified during program execution using lock-free primitives without spawning any new thread.

The output of our profiler consists of three-dimensional communication matrices – one for true communications and one for communication reuse. The third dimension contains a sequence of memory addresses shared between corresponding threads, which can be used to compute communication distances. Nonetheless, two-dimensional communication matrices can be easily extracted for the purpose of visualization. Figure 2.5 shows a side-by-side overview of true communication and communication reuse matrices extracted from a function in the application `water_nsquared`, in which the discrepancy between communication reuse and true communication matrices is easily observable. In this specific case, we can spot the threads that are reusing communication more than others. Such an observation can serve as the primary source of information for data-locality optimizations, such as thread/data mapping.

2.2.3 CRD: Communication reuse distance

Data locality analysis is an established method for evaluating the efficiency of memory accesses within an application. The traditional reuse distance analysis inspired us to propose *communication reuse distance (CRD)* to analyze thread communications. CRD is a metric for measuring the data locality of communication events between each pair of threads, with shorter distances having a higher chance to represent cache hits and longer distances less so. We created a tool to measure the CRD for each code region (loops and functions in pthread and parallel regions in OpenMP). The input consists of three-dimensional communication

Algorithm 1 Communication extraction using asymmetric signature memory.

```
for all memory access $a$ in the program do
  if $Type(a)$ is read access then
    if $a$ in write signature then
      if $a$ not in read signature & $lastWrite.tid \neq a.tid$ then
        add RAW dependence to comm. matrix;
      else if $lastWrite.tid \neq a.tid$ then
        add RaR dependence to reuse matrix;
      end if
    else {$a$ not in write signature}
      insert $a$ to read signature;
    end if
  else {$a$ is write access}
    clear out correspondent entry in read signature;
    insert $a$ to write signature;
  end if
end for
```

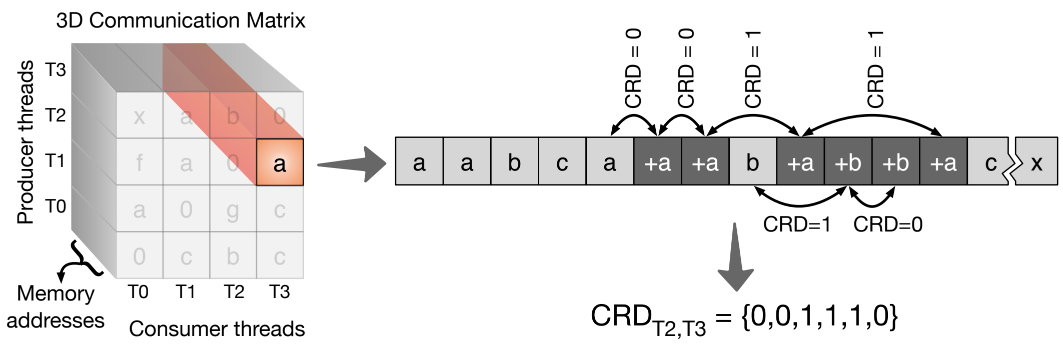


Figure 2.6: Computing CRD for a sample 3D communication matrix. Dark cells denote reused communication.

matrices that we generate using our profiler, including the sequence of true communication and their reuse. Communication reuse events are previously annotated with a plus sign to distinguish them from true communication. Given a communication trace for a pair of threads, we define the logical access time of a communication as its index position in the trace, counted from the first communication event. Thus, $CRD_{i,j}$ is the number of distinct data elements accessed between two consecutive usages of the same element among threads i and j.

Figure 2.6 shows an example of CRD analysis for a pair of threads. In this example, we have two memory locations, a and b, shared between threads $T2$ and $T3$. To obtain the CRD values, we need to compute the distance between the reused communication on each memory location. Consequently, by concatenating all $CRD_{i,j}$ together, we can achieve the final CRD value. Algorithm 2 shows the pseudocode for measuring the CRD.

We use a histogram diagram to provide additional insights into the CRD, and potential cache misses related to communication events. Moreover, we propose two cutoff distances: (1) maximum cutoff and

Algorithm 2 Computing communication reuse distance

```
CRD = [];
for all threadPair in commMatrix do
   uniqueEvents = unique(threadPair);
   for all element $x$ in uniqueEvents do
      indices = find occurrence indices of $x$ and $+x$ in threadPair;
      distances = compute pairwise distance of indices
      append distances to CRD
   end for
end for
```

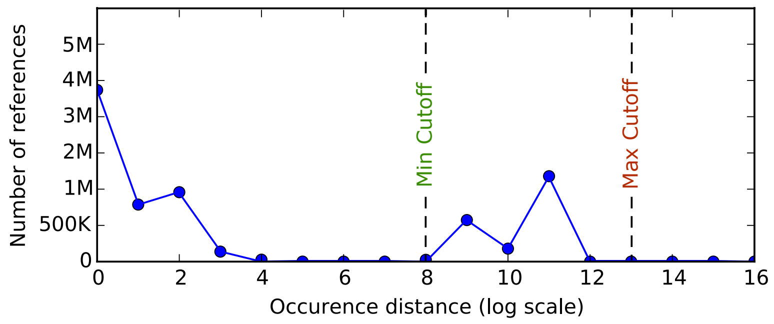


Figure 2.7: A sample CRD histogram. Max and min cutoffs are placed depending on the cache size.

(2) minimum cutoff, where the former is defined as the total size of cache available in the target system, whereas the latter is defined as the cache size minus the total size of non-sharing variables in the application. Depending on the CRD value, the following conditions might apply, explaining the cache behavior:

- CRD[$x >$ Max Cutoff] $\Longrightarrow$ Definite capacity miss
- CRD[Min Cutoff $< x <$ Max Cutoff] $\Longrightarrow$ Probable capacity miss
- CRD[$x <$ Min Cutoff] $\Longrightarrow$ No capacity miss

Figure 2.7 shows a sample histogram, where most of the communication reuse distances are below the minimum cutoff. Thus, they would not suffer from cache capacity misses. However, a fraction of reuse events fall between the minimum and maximum cutoffs, and therefore the chance of data locality deficiency exists. Nevertheless, based on this diagram, we can quickly evaluate the efficiency of communication reuse and apply further optimizations and data structure modifications to prepare the application for the target platform.

CRR: Communication reuse ratio matrix

We introduce the concept of *communication reuse ratio (CRR)* as a matrix similar to a standard communication matrix, where each cell contains the ratio of reuse to true communication instead. To obtain such a matrix, we first extract true communication and reuse matrices and then use element-wise division to generate the CRR matrix. The CRR matrix of a program region provides an overall understanding of the amount of reuse compared to total communication. Hence, the threads that are not reusing communication or are not mainly involved in repeated communication events could be easily identified. Our empirical experiments showed that optimized applications have more homogeneous and balanced CRR matrices. Thus, inspired by Diener et al. [69], we propose two quantitative metrics called homogeneity and balance to characterize and compare CRR matrices. Such a comparison guides a programmer to reach an optimal implementation.

Homogeneity We believe that accessing the outcome of a communication event multiple times is more desired than a long sequence of true communication. Thus, we would like to see homogeneous communication reuse over all threads rather than some threads having no communication reuse at all. Hence, we define *communication reuse homogeneity* as Equation (2.1), where T denotes the total number of threads and var denotes the variance function. For each row of the CRR matrix, we compute the variance and finally compute the average of variances over all threads. A lower value indicates more homogeneous communication reuse.

$$Homogeneity = \frac{\sum_{i=1}^{T} var(CRR[i])}{T} \tag{2.1}$$

Balance It is essential to determine whether some threads have a more communication reuse ratio than others, as such information can be used for a more optimized thread placement. To characterize this property, we introduce a metric called *communication reuse balance*. We first calculate the total reuse ratio for each row in the CRR matrix ($CommReuseRow$), where each element i of $CommReuseRow$ contains the sum of all communication reuse ratios for thread i. Similar to computing load balance, we compute the communication reuse balance as Equation (2.2). Typically, we seek to have a lower value to have a more balanced CRR matrix.

$$Balance = (\frac{max(CommReuseRow[1...T])}{\sum_{i=1}^{T} \frac{CommReuseRow[i]}{T}} - 1) \times 100\% \tag{2.2}$$

2.2.4 Communication bottleneck analysis

We believe that communication patterns and CRD analysis are valuable methods for identifying bottlenecks and optimizing programs, regardless of their different aspect in characterizing communication issues. CRD analysis informs a programmer which region's *communication* is likely to trigger cache misses, while communication and CRR matrices report which *threads* are likely to cause communication and reuse. The combination of CRD and CRR analyses along with homogeneity and balance metrics can provide an architecture-independent instrument suitable for targeted optimization. For instance, if CRR identifies several threads as communicating or reusing the communication among each other, the CRD analysis can detect whether this communication would be problematic or not. Furthermore, when CRD is high, data structure and data access optimizations are commonly suggested. Alternately, if CRR metrics are high, it is recommended to apply thread-affinity optimizations.

Table 2.1 contains several optimization methods that we found helpful, each categorized into optimization type and stage. Within data-affinity optimization, we can apply various code optimization array blocking, loop reordering, or loop fusion. We can also perform data-affinity scheduling during runtime. In the case of thread-affinity optimization, we are only aware of applying thread mapping optimization. The mapping policy should be selected based on the processor's specification.

	Code optimization	Run-time optimization
Data-affinity opt.	Array blocking, loop reordering, loop fusion	Data-affinity scheduling
Thread-affinity opt.	-	Thread mapping (various policies)

Table 2.1: The list of optimization types along with some examples categorized into coding and runtime optimizations.

2.3 Experimental results

We conducted various experiments on several benchmark applications, including linear-algebra programs, to validate the benefit of our method for identifying performance bottlenecks and applicable optimizations. We tested the benchmarks with 16 threads and various input sizes, with an average of five independent executions to ensure the correctness of the results.

As mentioned earlier, previous approaches [16, 17, 64, 65] often rely on program simulation and sandboxing to extract communication patterns, which impose excessive runtime and memory overhead to store intermediate data such as memory traces. On the contrary, our proposed profiler can detect communication patterns during execution with an average slowdown of $110\times$ while occupying less than 1 GB memory for allocating software signatures. Naturally, the runtime and memory overhead largely depend on the inherent communication behavior of the target application – the more communication and memory accesses are involved, the more slowdown will be caused. Nonetheless, we believe that the level of details that we can obtain by using our profiler can justify the incurred overhead.

In this section, we evaluate various aspects of our method. First, we demonstrate the applicability and usefulness of our proposed metrics. Then, we analyze the impact of input size on the communication reuse distance. Finally, we provide our findings on appropriate optimization methods based on the metrics that we proposed.

2.3.1 Communication analysis validation

To demonstrate the validity of the proposed metrics, we looked for multi-threaded applications that come in two different versions, each with different optimization levels. Our investigation in popular benchmark applications that have this property led us to *lu*, *ocean*, and *water* from the SPLASH benchmark suite. We analyzed these three applications (in total, six programs) using our method to show the effect of communication locality issues and code optimizations on communication reuse metrics. For each application, we present the results in a tabular format (Figures 2.8 to 2.10), which includes relevant diagrams for the most communication-intensive functions. Each row relates to a function within those programs and contains a CRD histogram and two CRR matrices, where the first matrix is for the non-optimized version and the second for the optimized version. Each CRR diagram is annotated with two numbers at the top, representing homogeneity and balance.

lu. `lu_ncb` and `lu_cb` are two different versions of the same application, where the former is non-contiguous, and the latter is a contiguous-block implementation. `lu_ncb` uses a one-dimensional array, in which the matrix to be factored is stored. On the contrary, `lu_cb` uses a two-dimensional array, in which all data points in a block (touched by the same processor) are allocated contiguously and locally.

We analyzed both versions using our profiler and provided the results for communication-intensive functions in Figure 2.8. The CRD histogram for function `bmod()` shows that the contiguous version has a lower distance, leading to fewer cache capacity misses. For validation, we analyzed the cache miss rate

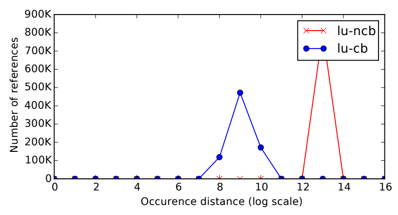
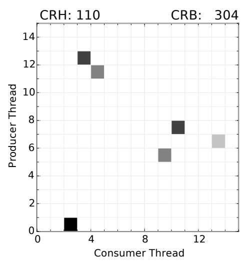
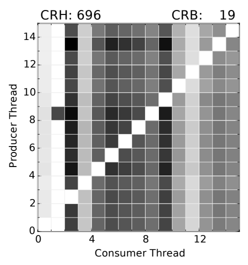
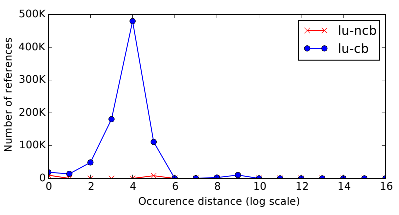
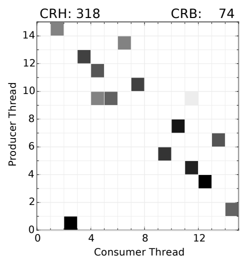
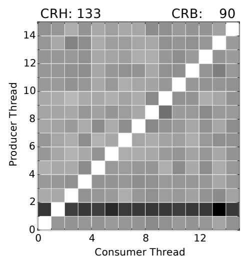
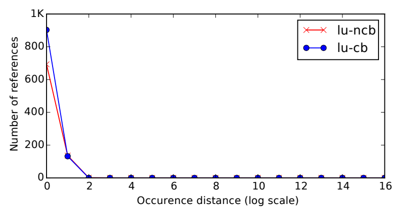
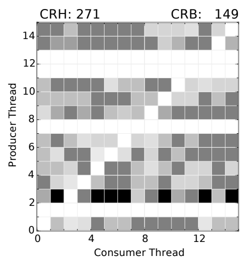
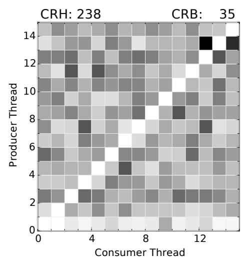
	CRD	CRR (`lu_ncb`)	CRR (`lu_cb`)	LL Cache Miss Ratio
`bmod()`				ncb: 0.65% cb: 0.38%
`daxpy()`				ncb: 0.09% cb: 0.12%
`lu()`				ncb: 0.01% cb: 0.007%

Figure 2.8: Communication analysis results for `lu_ncb` and `lu_cb`.

using Cachegrind. Detailed cache miss ratios reported for each function in Figure 2.8 show that the CRD histograms comply with the miss rates. Balance and homogeneity metrics shown on top of the CRR matrices denote higher communication imbalance in the non-contiguous version. However, the optimization increased the heterogeneity. We believe that the balance of communication compensated for the heterogeneity. In function `daxpy()`, the number of reuse in the contiguous version is much higher than the non-contiguous version. However, the distance of communication is similar. A noticeable change is the homogeneity of communications, which is improved in the contiguous version. We do not notice any significant improvement in the last function, as the CRD histogram and CRR matrices follow the same pattern. Overall, our fine-grained analysis was able to report slightly better results for the optimized version. Such an observation was not made by Biena et al. [70], who reported that the cache behavior of both versions is similar.

ocean. The two versions of this benchmark application solve the same problem but with a different memory layout. The non-contiguous implementation uses two-dimensional arrays, which avoids allocating contiguous partitions. The contiguous version is implemented with three-dimensional arrays. The first dimension specifies the processor which owns the partition so that partitions can be allocated contiguously [70].

Figure 2.9 shows the most communication-intensive functions of both `ocean_ncp` and `ocean_cp`. The CRD histograms of functions `laplacalc()` and `copy_red()` clearly show a much higher communication distance for the contiguous implementation, which is in line with the reported cache misses. The CRR matrices of both versions of function `laplacalc()` demonstrate a similar pattern, while the non-contiguous

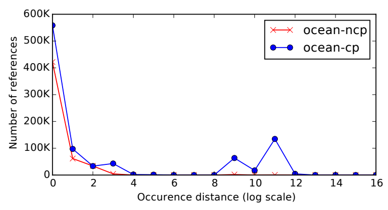
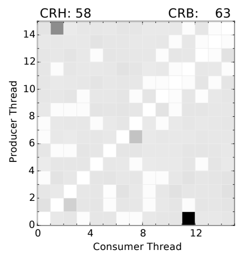
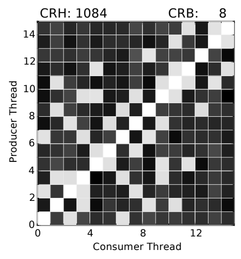
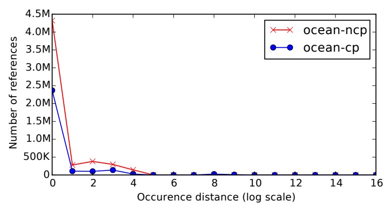
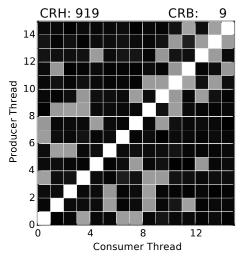
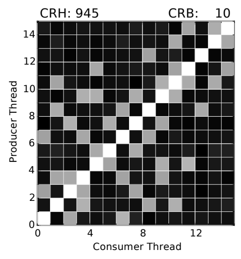
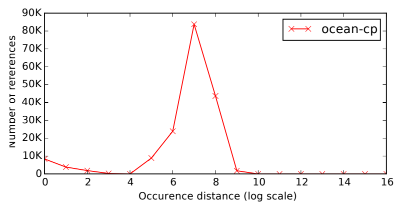
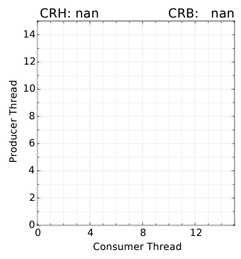
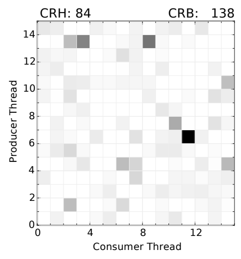
	CRD	CRR (`ocean_ncp`)	CRR (`ocean_cp`)	LL Cache Miss Ratio
`laplacalc()`				ncp: 3.70% cp: 6.25%
`jacobcalc()`				ncp: 2.19% cp: 2.39%
`copy_red()`				ncp: N/A cp: 0.09%

Figure 2.9: Communication analysis results for `ocean_ncp` and `ocean_cp`.

version seems more homogeneous. In this case, the CRD profile is more helpful in finding the source of a locality issue. We further observe that a number of functions, such as `jacobcalc()`, do not experience a significant change by the optimization, despite having a high cache miss rate and communication insensitivity. We conclude that just by focusing on the hotspot regions, we cannot effectively apply optimizations. Thus, we investigated the reason for high CRD distance and poor CRR metrics for the contiguous version. We found out that this issue can be attributed to the lower amount of shared write operations at the cost of a much higher cache miss rate on multi-core devices [70]. Our analysis successfully identified the region where the optimization failed to perform better than the straightforward implementation.

water. `water_nsquared` and `water_spatial` are non-optimized and optimized implementations for the same program, respectively. In `water_nsquared`, the forces and potentials are computed using an $\mathcal{O}(n^2)$ algorithm [71], whereas `water_spatial` uses a more efficient algorithm with a computational complexity of $\mathcal{O}(n)$. Moreover, the optimal version uses a uniform three-dimensional grid of cells on the problem domain. Therefore, processors which own a cell will only access the neighboring cells, improving the overall data locality.

Our analysis results reported in Figure 2.10 clearly show the superiority of the spatial version over non-spatial implementation. All CRD histograms for the representative functions show lower communication reuse distance for the spatial version. Our cache analysis results also show that the cache miss ratio in the spatial version is much lower than the `nsquared` alternative. The CRR matrices for the spatial implementation also seem more regular and homogeneous. The CRR matrices of function `PREDIC()` are a

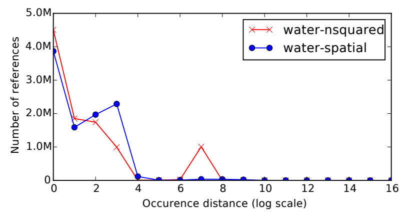
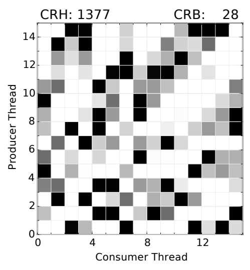
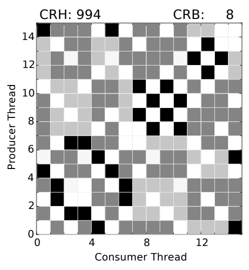
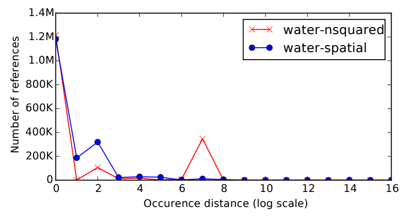
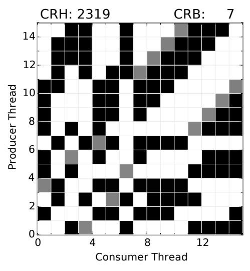
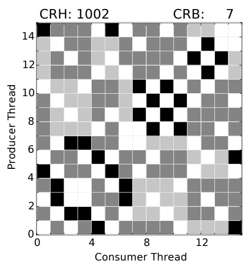
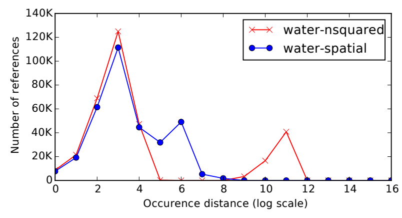
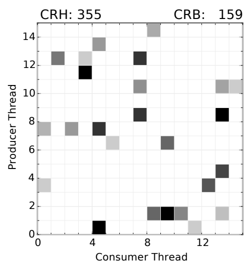
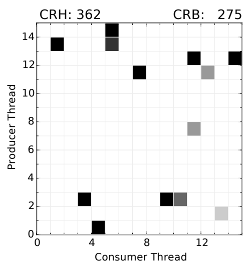
	CRD	CRR (`water_nsquared`)	CRR (`water_spatial`)	LL Cache Miss Ratio
`INTERF()`				nsq: 0.24% spat: 0.02%
`POTENG()`				nsq: 0.16% spat: 0.007%
`PREDIC()`				nsq: 1.45% spat: 0.76%

Figure 2.10: Communication analysis results for `water_nsquared` and `water_spatial`.

good example for demonstrating that relying solely on CRR is not enough for analyzing the communication behavior, as no significant difference is observable between the two CRRs. However, the CRD diagram shows a shorter distance for the spatial version.

2.3.2 Communication scalability analysis

Although the proposed metrics are hardware agnostic and do not change on different systems, the application's input might largely affect our metrics. Thus, it is favorable that the user analyzes the target application using different inputs to reach an optimal version. Nevertheless, we noticed that such an input dependency could be used to our advantage. Particularly, we observed that the amount of change, either the amount of reuse or distance values, can be used for analyzing the scalability of a given application with larger input sizes. Such a feature is not only helpful for understanding the application's requirements but also for selecting the region that suffers from increased communication overhead.

Figure 2.11 demonstrates the effect of two input sizes (i.e., `simdev` and `simsmall`) on the CRD histograms of our test cases. The results of both versions of `lu` show that only function `bmod()` is affected by a larger input size. Other diagrams show that a larger input size increases both the number of communication events and their distances. Hence, there is a risk of higher cache misses related to communication events after increasing the input size. We observed similar behavior in the `ocean` benchmark, in which the results show that the optimized version still suffers from high communication overhead. In contrast to `lu` and

		Before optimization			after optimization					
Benchmark	Bottleneck region	CRD	H	B	CRD	H	B	Data-affinity opt.	Thread-affinity opt.	Speedup
Back Propagation	backprop.c: L321 - L327	**5**	84	21	2	54	14	Loop reordering, Array blocking	-	15%
Needleman-Wunsch	needle.cpp: L161 - 167, L176-L181	**10**	278	139	5	239	45	Data affinity (32 OMP chunk size), Array blocking	-	29%
SRAD	main.c: L254 - L290, L296-L320	**13**	**1124**	153	4	981	89	Loop reordering	Thread mapping	35%
Particle filter	ex_particle_OPENMP_seq.c: L488 - L495, L480-L482	**8**	321	89	3	384	121	Loop fusion	-	56%
Streamcluster	streamcluster_omp.cpp: L540 - L548	3	**1409**	6	-	-	-	-	Thread mapping	19%

Table 2.2: Detailed results of communication bottleneck analysis for a subset of the Rodinia [72] benchmarks. Each metric is reported twice. Once for the non-optimized and once for the optimized version. H and B denote homogeneity and balance, respectively. Highlighted bold numbers indicate high metric values, instructing to apply relevant optimizations.

`ocean`, `water_spatial` shows better scalability, as its CRD profile does not seem to be affected a lot. Such an observation clearly shows that the optimized algorithm performs better for larger inputs.

2.3.3 Communication bottleneck analysis

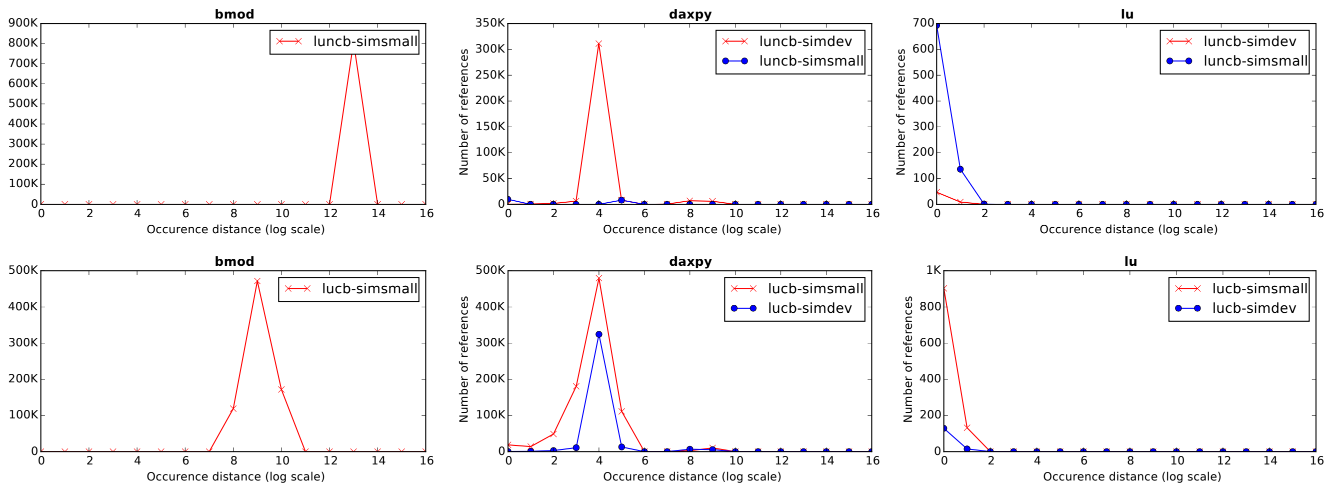
We used the Rodinia benchmark suite [72] to show the effectiveness of the proposed metrics in finding communication bottlenecks. We were able to detect bottlenecks in the five benchmarks of Rodinia listed in Table 2.2. The problematic code regions are reported for each application and the metric results before and after optimization. For example, high CRD values demand to apply data-locality optimizations, which are mostly done by code optimizations like array blocking and loop reordering. Conversely, high homogeneity and balance values are a hint to apply runtime thread-affinity optimizations.

Computing the minimum cutoff distance for each application identified high median CRD values in the benchmarks Backpropagation, Needleman-Wunsch, SRAD, and Particle Filter. High CRD indicates potential communication locality issues, which we tried to address with code optimizations and runtime OpenMP scheduling. These optimizations lowered the communication reuse distance and led to a considerable amount of speedup. In Streamcluster and SRAD, we noticed high communication reuse heterogeneity among threads, which calls for locality-aware thread mapping optimization. We could not measure the effect of thread mapping on our metrics because our analysis currently focuses on the data and thread mapping only alters the thread placement. However, a notable speedup was achieved just by optimized thread mapping.

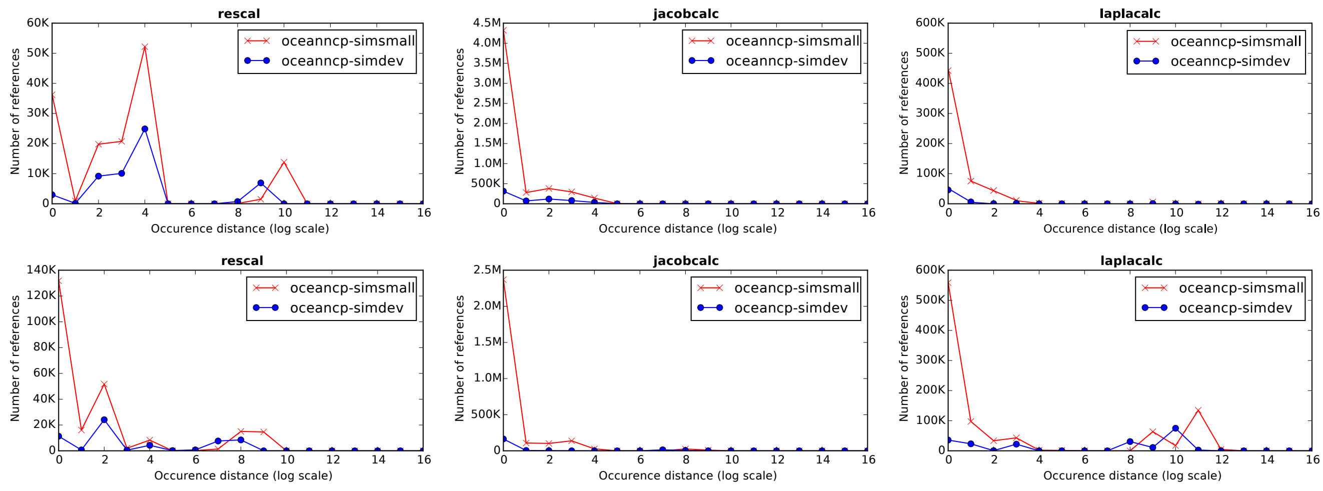
Our analysis results provided optimization insights, related to data and threads, which improved the runtime performance by up to 56%. To the best of our knowledge, other methods do not provide such detailed information on program communication for selecting the appropriate type of optimization. Thus, we believe that our method can be used in companion with other performance debugging tools to extend the scope of performance analysis.

2.4 Related work

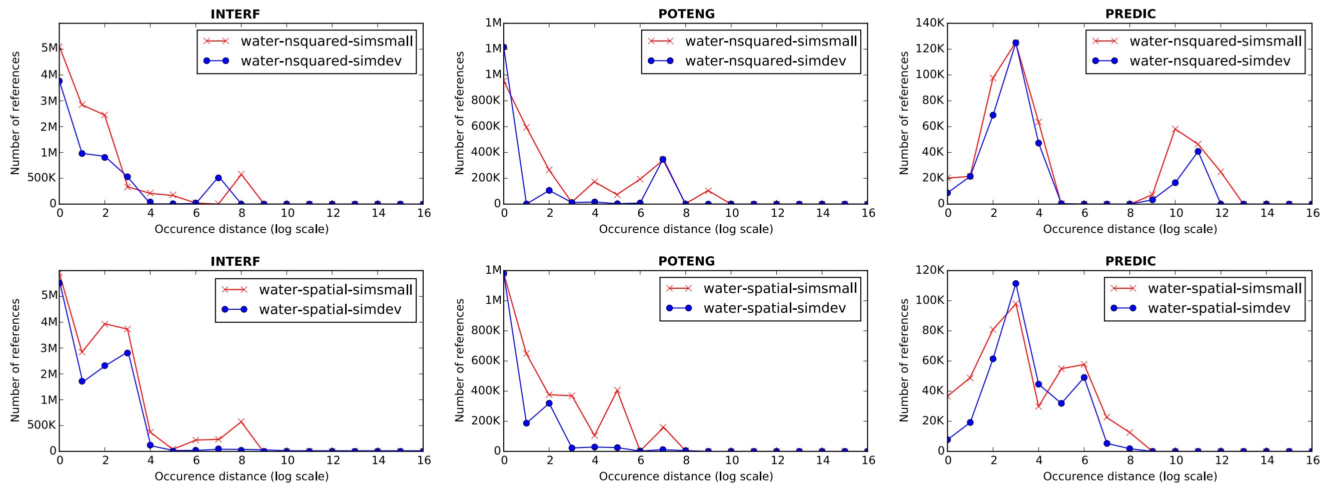
To the best of our knowledge, no similar paper is published on evaluating the locality of communication in shared memory systems. Gprof [73] and Threadspotter [74] are two well-known methods for optimization suggestion, yet they are unable to produce comparable results. Profiling multi-threaded applications using Threadspotter will primarily generate results only valid for a single thread (master). Furthermore, these methods do not necessarily focus on communication bottlenecks, which is the main target of this paper. Nevertheless, we discuss previous efforts on communication pattern detection and multi-core reuse distance below.



(a) lu



(b) ocean



(c) water

Figure 2.11: The effect of two input sizes (`simdev` and `simsmall`) on CRD histograms for the applications lu, ocean, and water.

Although various methods have been proposed for analyzing the communication pattern in parallel applications, most of them target distributed-memory applications [75, 76, 19, 21, 15]. Such a communication analysis is straightforward but cannot be extended to shared-memory programs. Simulation-based methods attempt to simulate the system by logging and recording every change during program execution [16, 17], preventing a practical analysis as it implies high runtime overhead and excessively large output files with more than 100 GB for a moderate input size [16]. On the contrary, our method uses a memory-efficient profiler. Other studies [77, 78, 79] utilized Intel Pin to extract communication patterns and used them for thread and data mapping. However, their output is a single communication matrix. Thus, they are not able to detect the dynamic behavior of an application in different code regions.

Reuse distance analysis has been studied extensively for single-core architectures [58, 80]. However, due to the nondeterministic runtime scheduling of threads in multi-threaded applications, alternative methods have been introduced to preserve hardware independence [54, 61, 62, 81, 82]. Recent methods [53] are based on statistical models to predict the concurrent reuse distance. However, this method only works for specific regions like loops. Furthermore, none of these methods can be used to analyze communication events. Our method combines communication reuse distance with thread communication analysis, which yields more comprehensive results.

Two studies [50, 51] tried to combine reuse distance analysis with other metrics. Rane et al. [50] proposed a tool comprised of different analyses and metrics, such as reuse distance analysis, cycles per access, hit ratios, and access strides to investigate data structures and find out their bottlenecks. However, they only focus on data structures and neglect the effect of threads on communication. Another study [51] introduces architecture-independent metrics, including multi-core reuse distance and communication analysis, to perform performance analysis. However, no additional insights into thread communication are provided.

2.5 Discussion

In Section 2.3, we demonstrated the effectiveness of the proposed metrics in detecting communication bottlenecks. We specifically evaluated our method on several linear-algebra benchmark applications, which often access a large volume of data. Since tensor operations found in deep-learning applications often fall into the same category, we believe that the lessons and findings presented in this chapter can be transferred and used for optimizing tensor-based applications. Although such metrics proved to be useful on multi-core systems, obtaining such fine-grained information on many-core processors, such as GPUs with hundreds of threads, could be extremely challenging. In such a scenario, the communication matrices for GPU threads become excessively large and make the analysis itself infeasible in terms of the required processing time and memory. Nonetheless, not all stages of the AI research pipeline are running on GPUs. For instance, training data set preparation, including cleaning and augmentation, have the potential to be parallelized on CPUs and thus benefit from our approach.

Furthermore, we believe that our proposed profiler is helpful for analyzing the communication behavior of a given application and then suggest the usage of efficient communication primitives available in specialized libraries such as Nvidia's NCCL [55]. As we already introduced NCCL in Section 2.1, this library provides fast and efficient communication primitives for inter-GPU data sharing. NCCL supports various communication patterns that can be categorized into point-to-point and collective communications. Point-to-point communication relates to ordinary data sharing between pairs of threads. Anti-pattern and nearest-neighbor patterns fall into this category. On the flip side, collective communications deal with bulk data exchange among all processes simultaneously. Broadcast, gather, scatter, all gather, and all-to-all patterns belong to collective communications. Figure 2.12 illustrates these patterns using communication matrices.

Our analysis found a number of communication patterns within different regions of our target applications resembling the same patterns shown in Figure 2.13. For instance, the barrier implementation in the PARSEC

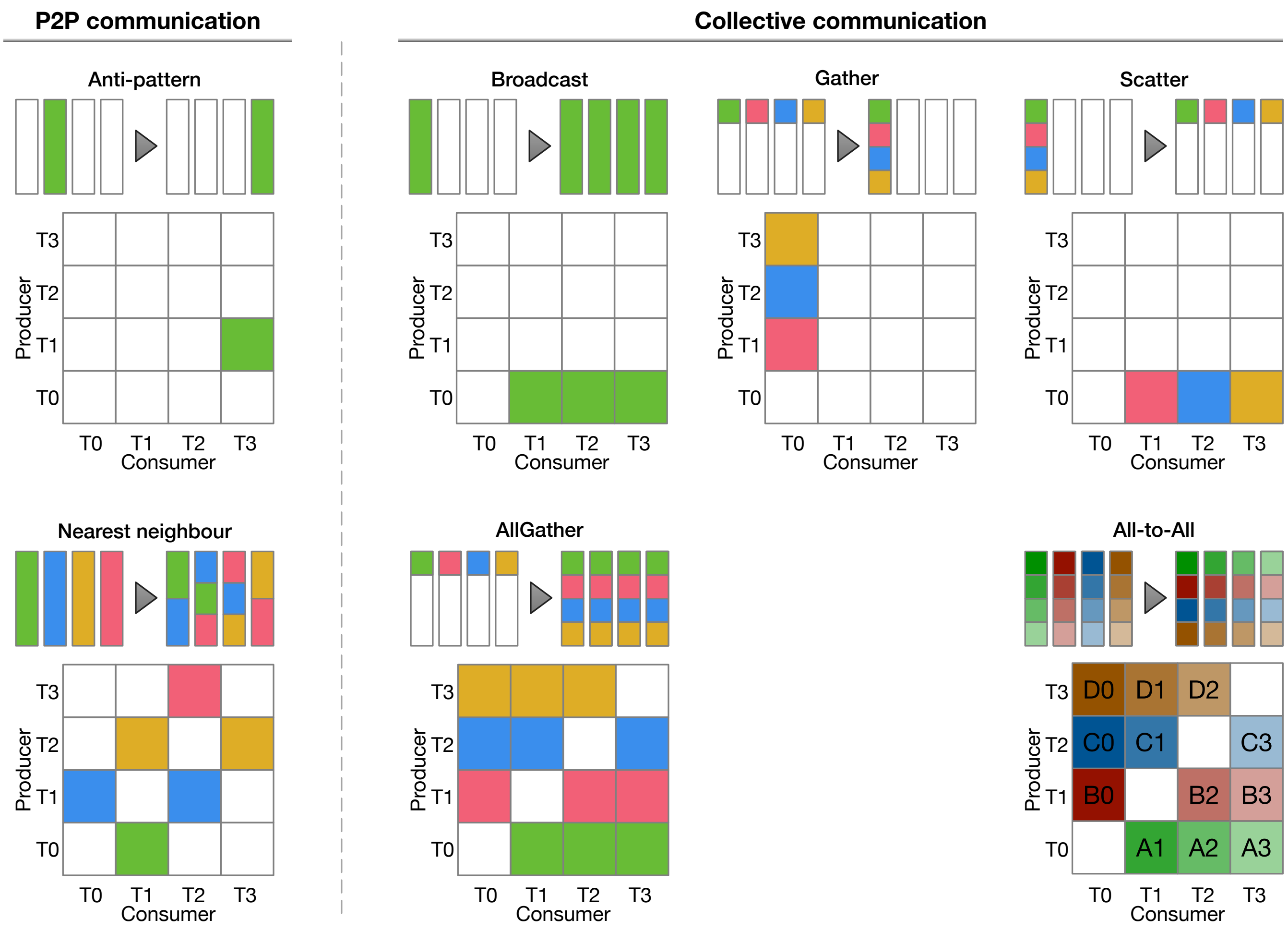
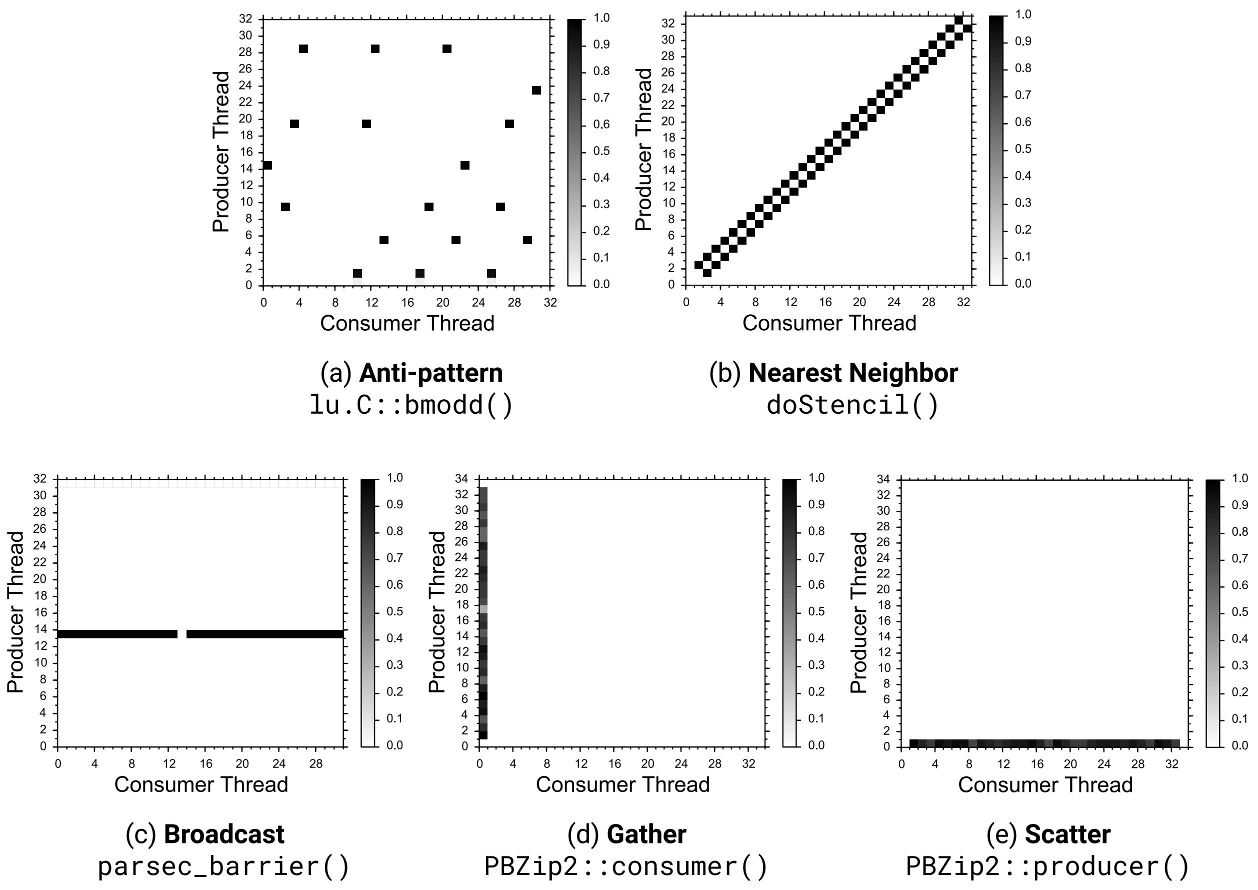


Figure 2.12: Popular types of task/thread communications categorized into point-to-point and collective communications.

benchmark suite uses a spin-lock and behaves like a broadcast pattern. On the other hand, we noticed gather and scatter patterns in the two main functions of the PBZip2 program. Such an observation in generated communication patterns validates that our method can identify the communication type (i.e., P2P or collective) and suggest the relevant communication method available in specialized libraries (e.g., NCCL).

2.6 Conclusion

This chapter demonstrated that our proposed platform-independent communication metrics enable the analysis of parallel programs in terms of their inherent data communication without being tied to particular hardware. Additionally, we were able to detect data communication bottlenecks and address them with an appropriate optimization method. Communication reuse distance (CRD) reflects the effect of communication on the cache, while the communication reuse ratio (CRR) matrix sheds light on the amount of reuse after a true communication between threads. With the help of two additional quantitative metrics, the homogeneity



(a) **Anti-pattern** `lu.C::bmodd()`

(b) **Nearest Neighbor** `doStencil()`

(c) **Broadcast** `parsec_barrier()`

(d) **Gather** `PBZip2::consumer()`

(e) **Scatter** `PBZip2::producer()`

Figure 2.13: The visualization of point-to-point and collective communication patterns identified in benchmark applications.

and balance of CRR matrices can be easily evaluated. We showed that our method is not only able to detect communication bottlenecks in different code regions, but also can help programmers apply a suitable type of optimization. Additionally, we demonstrated that such an analysis is helpful for determining the input scalability of a given application with regard to its cache usage.

We are aware that our proposed analysis might be only feasible when the number of threads is not immensely large. Thus, a potential future line of work could be the extension of this work to consider specific threads and memory locations (e.g., shared-memory regions in GPUs) that are involved in data communication. Such approach can potentially support the large pool of threads that are typically found in GPU programs.

3 A primer on deep learning

This chapter provides a brief introduction to the basics of deep learning, which serves as the basis for understanding the following chapters, particularly Chapters 4 to 5. We begin with the fundamental data structure used in deep-learning applications. Afterward, we primarily focus on convolutional neural networks (ConvNets)—a subset of deep-learning models mainly used in computer vision tasks. Moreover, we focus on the system side of ConvNets by explaining their algorithms, constituent layers, and underlying computations. Finally, we briefly review related techniques such as neural architecture search, reinforcement learning, and graph convolutional neural networks, as we have used them in our methods.

3.1 Algebraic data structures – Tensors

Machine-learning applications, particularly deep neural networks, usually work with numeric values stored in a data structure. Such cornerstone structures are commonly known as *tensors* (a.k.a. ND-Arrays) that house a collection of numbers in N dimensions. Within the context of image processing, we usually deal with ND-Arrays of up to $N = 4$. Scalars are 0D-Arrays that contain a single number and can represent the intensity of a single pixel. Vectors are 1D-Arrays, containing a list of numbers that can potentially represent the colors (R, G, and B) of a given pixel. By adding another dimension, we obtain a 2D-Array or a matrix, which can represent a grayscale image as it only has two dimensions (height and weight). Consequently, adding another dimension leads to a 3D-Array that can fully represent a colored image since each pixel can now become a vector of image channels.

Lastly, 4D-Arrays, or often called tensors, are mostly used for representing a group of multi-channel images. As Figure 3.1 depicts, tensors can be viewed as generalizations of matrices to N-dimensional space. Typically, tensors have the following dimensions:

Channel dimension. The dimensionality of the feature space is represented by channels (C). For example, for colored images that we typically feed into the neural networks, we have three channels per pixel according to the primary colors (red, green, and blue). However, the channel size is not always three, as the neural network might add additional channels in the mid-layers of the network to increase the feature space and extract additional information from the given input.

Spatial dimension. Within the context of image-based convolutional networks, the spatial dimensions describe the size of a given tensor. We usually specify the size with height (H) and width (W) parameters.

Batch dimension. For more efficient computation, multiple inputs can be grouped together as batches with a length of B prior to any computation. Batching the inputs provides higher efficiency as it improves data reuse, mostly due to reusing convolution filters' weights during the computation.

Similar to vectors and matrices, arithmetic operations such as addition and multiplications can also be performed on tensors. Often the ordering of tensor dimensions plays an important role in data locality and efficient memory access. Thus, the compiler is responsible for the correct ordering of the tensor dimensions.

Scalar (0D-Array)	Vector (1D-Array)	Matrix (2D-Array)	Tensor (3D-Array)
1	$\begin{bmatrix}1\\2\end{bmatrix}$	$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$	$\begin{bmatrix}[1 \quad 2][3 \quad 4]\\ [5 \quad 6][7 \quad 8]\end{bmatrix}$

Figure 3.1: From scalars to tensors. Tensors are generalization of matrices.

Similar to many other projects, we extensively utilize tensors as the key data type for dealing with deep neural networks and accelerating the underlying operations, such as convolutions. A tensor must have a fixed number of dimensions and a fixed length for each dimension. Therefore, the total number of elements in a tensor is the product of all dimension lengths. Recently, specifically designed tensor processors can perform tensor arithmetics within a single instruction, offering higher efficiency than manually implemented libraries.

3.2 Deep neural networks

Until recently, nearly all software was written based on a predefined algorithm solely designed to solve a specific problem. This way of computer programming is often called *Software 1.0* and contains explicit instructions written by the programmer. To build an application, the programmer has to consider every corner case that the user might encounter, each requiring a specific rule to be devised. Clearly, such a design principle demands an extensive amount of time to find all the corner cases, devise the rules, and develop the appropriate algorithms. The resulting codebase would potentially become extremely large, making code maintenance an immensely challenging task.

Despite the popularity of the Software 1.0 programming paradigm, various attempts have been made to automate the process of devising rules for taking appropriate actions. Recent efforts in this area have led to the introduction of machine-learning methods, also often called *Software 2.0*. A class of machine-learning applications is deep neural networks, which are programmed more abstractly often via setting the weights of a collection of neurons and connections [83].

Most of the terminology and conventions used in neural networks are historically inspired by biological neural networks that constitute animals and the human brain. As depicted in Figure 3.2a, neurons within a neural network are often organized in *layers*, where internal layers are often called hidden layers. There is no strict threshold, but among the AI community, networks with more than eight layers are considered deep neural networks [84]. For instance, modern networks often contain hundreds of layers [85], making them extremely memory and compute-intensive applications.

A neuron receives multiple inputs from its predecessor layers and generates one output by computing the weighted sum followed by the neuron's activation function (see Figure 3.2b). The connection between the neurons i and j contain a weight $w_{i,j}$ that intensifies or weakens the input values. Hence, when a sample input is fed to the networks, the network triggers specific output neurons according to the goal of the network. The main task within deep neural networks is to learn the best values for such weights, which leads to the highest accuracy for the network. Nevertheless, due to the huge number of weights, no programmer is directly involved in setting the weights. Instead, the programmer uses supervised-learning methods to *train* the weights based on the given input dataset. Once the model is trained, the model can be used for making predictions in the inference phase. In the following, we will explain these phases in more detail.

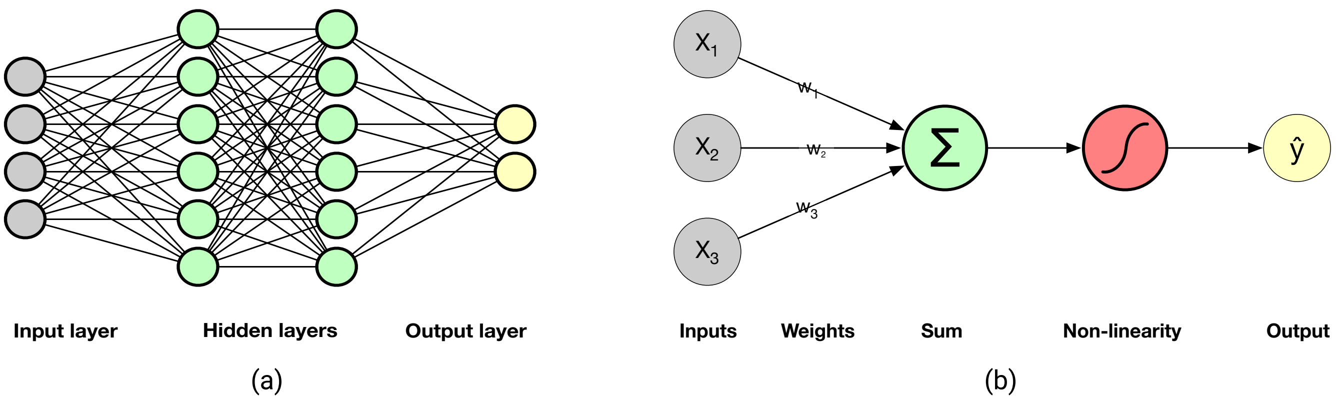


Figure 3.2: An illustration of neural networks. (a) Layered structure of neural networks. (b) The structure of a neuron.

3.2.1 Training phase

The aim of training a deep-learning model is to minimize a loss function given a dataset and a set of trainable weights. This process is formulated as:

$$\mathcal{L}(w) = \frac{\sum_{x_i \in X} \ell(w, x_i)}{|X|}, \tag{3.1}$$

where w is the set of weights, X is the dataset, and ℓ is the loss function. Gradient descent is the most popular method to minimize the loss function and selects the appropriate weights for the connections within the neural network. It is a first-order iterative optimization algorithm that can find the local minimum of a differentiable function. The main idea is to take repeated steps to calculate the loss function over the variable and move along the negative direction of the gradient (i.e., backward propagation). The step size for moving backward is proportional to the absolute value of the gradient. Here, we describe the main concepts and terms often used for training deep neural networks:

Back-propagation algorithm. To calculate the gradients, we use the back-propagation algorithm. Stochastic gradient descent (SGD) and its variants like AdaGrad and Adam are some of the well-known methods. SGD works by calculating layers' activation by performing a feed-forward pass from the input to the output neuron. Next, we calculate the gradient of the loss function for each neuron and each weight. According to the chain rule, the gradients are obtained iteratively and backward from the output to the input layer. Lastly, we update the weights using the formula $w_{i,j}^{t+1} = w_{i,j}^{t} - \eta \frac{\partial \mathcal{L}}{\partial w_{i,j}}$, where η is the learning rate multiplied with the gradient of the loss function. Stochastic gradient descent evaluates the gradient using a random mini-batch $\mathcal{M}^{(t)} \subset X$. Thus, the gradient computation requires an additional batch size parameter $M = |\mathcal{M}^{(t)}|$ and would perform the following formula.

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}} = \frac{\sum_{x_i \in \mathcal{M}^{(t)}} \Delta \ell(w^{(t)}, x_i)}{|X|} \tag{3.2}$$

In the case of distributed training, when synchronous data-parallelism is used, the mini-batch $\mathcal{M}^{(t)}$ will be equally divided into K partitions, where K is the number of workers (e.g., GPUs). Each worker is responsible for computing the local gradient and then averaged across all workers to obtain the global gradient. Such an averaging is typically performed via a collective communication operation called AllReduce, which reduces the tensors in all processes to a single tensor and returns the result to all processes.

Training hyper-parameters. To configure an optimal training process various hyper parameters are involved. Often, setting these parameters is quite time-consuming. Thus, various tuning methods exist to select the right parameters automatically. In the following, the most important hyper-parameters are listed:

Learning rate (η) is a scalar value that determines the ratio between step size and the absolute value of the gradient. In other words, it controls the magnitude of gradient updates.

Batch size (B) defines the number of samples propagated through the network at once.

Epoch relates to one forward pass and one backward pass of all training samples. In other words, an epoch finishes, when we go through the entire dataset once.

Often, there is a correlation between the hyper parameters. For instance, increasing the batch size requires a lower learning rate to avoid sudden updates of the weights and converging issues.

Dataset. The collection of samples that a machine-learning algorithm can use to train its weights are called a dataset. ImageNet [86], CIFAR-10/100 [87] are among the most popular datasets for image classification. Often, the dataset contains the sample and respective labels that describe the sample. Furthermore, the dataset is divided into training, validation, and test sets. The training set is a set of samples used to fit the network parameters, while the validation set is a separate sample of data that is used to assess the model quality while being training. The test set contains the samples that the network has not seen before to make sure that the network can provide correct results on unseen data.

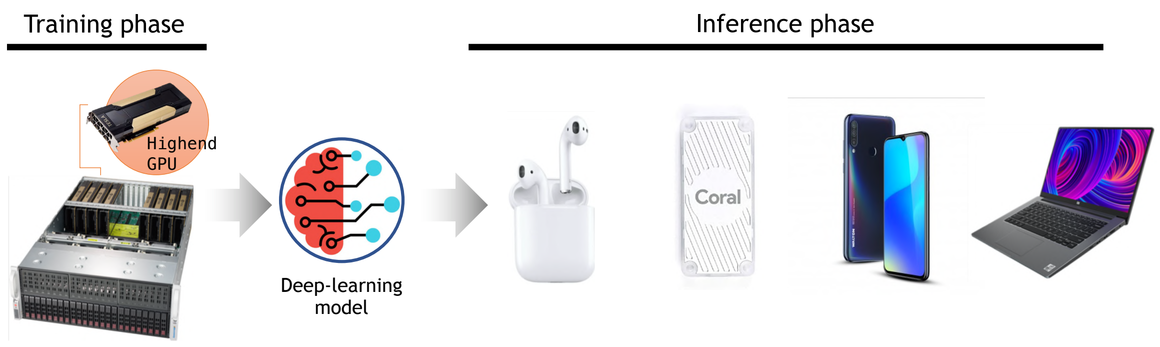
Top-1 and Top-5 accuracy. To report the prediction accuracy of a given deep-learning model and compare it with other models, top-1 and top-5 accuracy metrics are used. Top-1 is the conventional accuracy metric that reports the percentage of the correct top class (the one having the highest probability). On the other hand, top-5 accuracy means that any of the model's five highest probability answers must match the expected answer. For instance, assume that a deep-learning model is designed to perform object recognition, and a picture of a cat is shown. If the cat is not the first output but is among the first five outputs, using top-1 accuracy, we count the output as wrong. However, top-5 accuracy counts the output as a correct answer, because cat is among the top-5 guesses.

3.2.2 Inference phase

Once the training phase of a given deep-learning model is finished, the obtained model (i.e., trained weights) will be used to make predictions on previously unseen inputs. This process is called inference phase and the same forward pass used in the training phase is utilized for performing the predictions. Although the deployment of DNNs on the target device might seem trivial, obtaining real-time performance requires extensive optimizations and network adaptations. In this dissertation, we particularly focus on a deployment framework for optimizing the runtime of DNNs. Furthermore, the trained model may also need to be modified or simplified before being deployed for inference.

3.3 Performance analysis of deep-learning models

For many deep-learning researchers, a deep network's top-1 and top-5 accuracy are the key metrics to observe and tune, particularly while designing a model. Figure 3.3 clearly shows that the training and deployment hardware platforms are inherently different in computing, power budget, available disk storage,



	Cloud HPC
	Data center
Compute	200+ TOPs
Power budget	200+ W
Model size	300+ MB
Latency	ms - s
SoC scale	Max

Edge device			
Earphone	Always-on	Smartphone	Laptop
20 MOPs	100 GOPs	1-10 TOPs	10-20 TOPs
1 mW	10 mW	1-2 W	3-10 W
10 KB	100 KB	10 MB	10-100 MB
< 10 ms	~10ms	10-100 ms	10-500 ms
Nano	Tiny	Lite	Mini

Figure 3.3: Deploying deep-learning models on a diverse set of edge devices.

latency, and SoC (System on a Chip) scale. Furthermore, edge devices that host a given deep-learning model have a variety of different hardware characteristics. Deploying deep-learning models on such memory-/computation-constrained devices demands additional consideration to other key factors [88], as follows:

Computation speed. The amount of computations involved in the forward pass of a given deep-learning model is correlated with its runtime speed. Thus, counting these computations can provide a theoretical ball-park number for how fast a given model is. The number of computations is typically reported as *FLOPs*[1] or a slightly different metric called *MACCs*[2]. Since many of the computations in deep neural networks are based on dot-products, we use MACCs to compute the number of operations. For instance, the following dot-product code has n MACCs. In terms of FLOPs, there are $2n - 1$ FLOPs since n multiplications and $n - 1$ additions are involved.

```
y = w[0]*x[0] + w[1]*x[1] + w[2]*x[2] + ... + w[n-1]*x[n-1]
```

A MACC can roughly translate to two FLOPS. However, due to the widely available FMA[3] support on various hardware, we can perform fused multiply-add operations in a single instruction. It is worth mentioning that FLOPS or MACCs estimations can just provide a rough idea of the computational cost. Other parameters, such as memory consumption and access pattern, can largely affect the runtime efficiency. Depending on the application domain, other metrics are also used as a proxy to define the speed. For instance, within the context of the computer-vision application, frames-per-second (FPS) becomes rather important, as a higher FPS leads to a more real-time runtime nature.

[1]Floating point operations per second
[2]Multiply-accumulate operations
[3]Fused multiply accumulate

Memory consumption. Unlike high-end GPUs with gigabytes of RAM, low-budget processors often have access to a low amount of memory space with limited memory bandwidth. Obviously, the deep-learning model cannot be executed if the required memory exceeds the available capacity. More importantly, the memory access pattern governs the required memory bandwidth. For each layer in the deep network, the device is supposed to:

1. read the input or feature-map tensor from main memory,
2. read the layer's weights (filters) from main memory,
3. compute the dot-products,
4. write the result back to the main memory.

Such operations imply many memory accesses and, if not appropriately optimized, turn the application into a memory-bound version. Thus, the effect of memory read/writes on overall efficiency is equally important, if not more than the number of computations. In general, the fewer weights the model contains, the faster it runs [88].

Disk storage. The amount of storage space the model takes to be saved on the device might be important for some edge devices. Not only might disk storage be of concern, but a single model can add hundreds of megabytes to the download size to update the target application, preventing a quick over the air (OTA) update.

Power consumption. Whether the target device uses a battery or not, power consumption is an important topic. The deep-learning model should not draw a considerable amount of energy to drain the battery or make the device too hot. Furthermore, energy consumption is mostly dominated by memory accesses. Han et al. [89] provided a ballpark figure for the power consumption of a large neural network running on a 45nm CMOS processor. It was assumed that a 32-bit floating-point add operation draws $0.9pJ$, accessing a 32-bit SRAM cache takes $5pJ$, and a 32-bit DRAM memory access takes $640pJ$. Given such a processor, running a large deep network with one billion connections at 20fps would draw a total of $(20Hz)(1G)(640pJ) = 12.8W$ just for DRAM memory accesses as the cache is probably too small to accommodate all the values. This estimation exceeds the power envelope of a typical mobile device, proving that large models cannot be directly deployed on power-constrained devices without any model compression or pruning.

Despite the importance of all the factors above, the number of computations and memory consumption are the most important ones, as they affect other factors. For instance, more inefficient memory access leads to higher power consumption. Thus, performance engineers often focus on these two main factors while optimizing a deep-learning model.

3.4 Convolutional neural networks

Since this work is mostly related to the computer system's side of AI in general, we do not provide an extensive background on the historical terms, origin, and the wide range of available neural networks (e.g., RNNs, LSTMs, and GANs). We instead focus on deep convolutional neural networks (ConvNet) as they are the target applications that we aim to accelerate. In this section, we provide a summary of such networks along with their constituent layers. For more in-depth information, we suggest the reader to refer to more comprehensive books and tutorials [90, 91].

The essential idea behind ConvNets is their ability to learn a broad set of filters (a.k.a. kernels), organized into a hierarchy of layers, to extract meaningful information from a given image. Convolutional layers are the

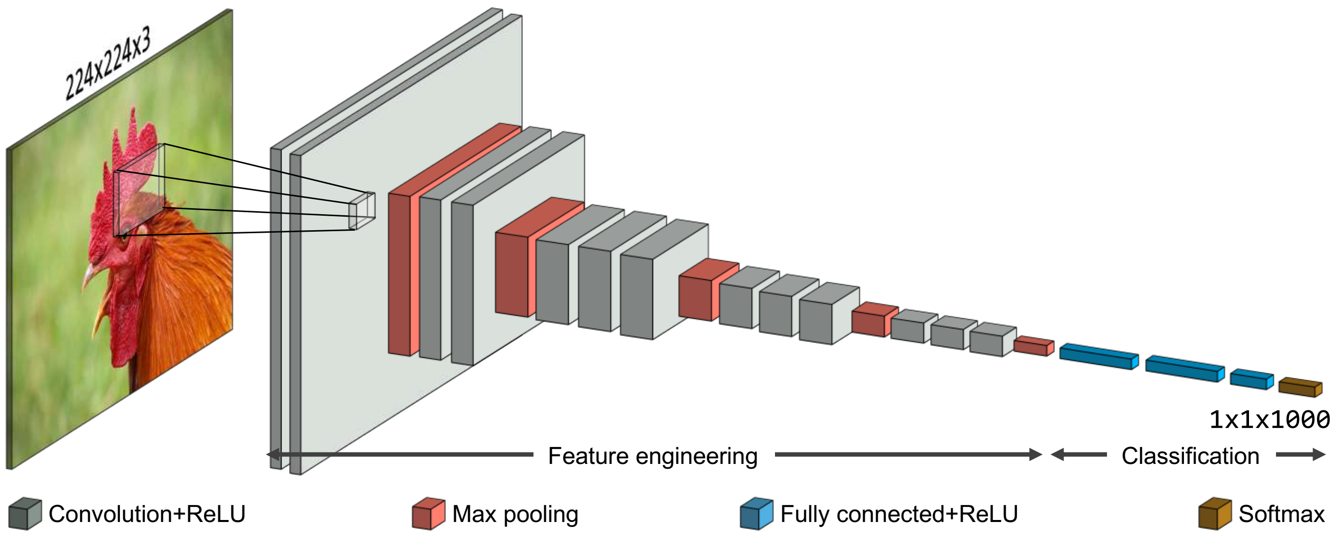
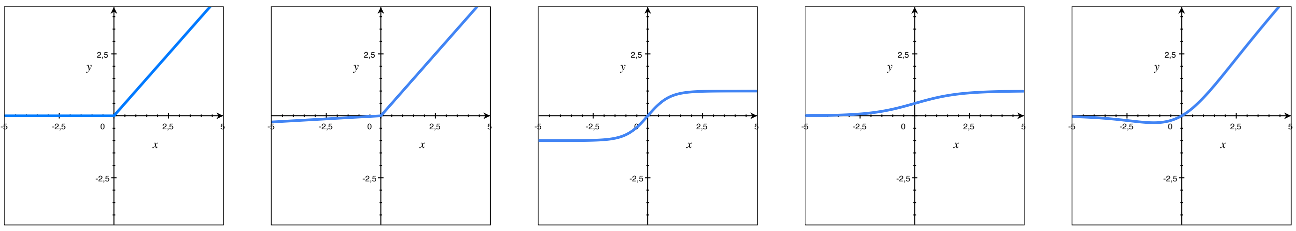


Figure 3.4: A typical convolutional neural network.

main constituents of ConvNets, which can be defined as the function $output = conv(input, filters)$, where $output$, $input$, and $filters$ are all multi-dimensional arrays (i.e., tensors). The main task is to cross-correlate a set of learned filters uniformly on various scopes of a given image using the sliding window approach. The output is a tensor called *feature map*, which contains abstract information, such as curves and edges. As we proceed from the first to the last layers of the network, we can observe the abstraction level of the features to rise. Therefore, deeper ConvNets are likely to perform better than shallow networks in various computer vision tasks. Figure 3.4 illustrates a typical ConvNet containing various layers.

The deeper a network becomes, the more parameters it usually includes. Hence, more data-dependent arithmetic operations will be involved, prolonging training and inference time. Furthermore, most computations take place in convolutional layers and accelerating them can significantly decrease the total inference time. It is often possible to reshape the convolution operation as a matrix-multiplication operation. Thus, we can use highly efficient linear algebra libraries (BLAS), such as single-precision general matrix multiply (SGEMM). Such libraries are often highly parallelized and use GPUs for obtaining the highest speedups. Higher bandwidth, latency hiding via thread parallelism, and easily programmable registers make GPUs a lot faster than CPUs.

As the overall size of ConvNets grew, which is influenced by their depth, input size, and kernel size, the efficient execution of such networks gained more importance, leading to the introduction of inference frameworks such as TVM [92], TensorFlow's XLA [93], and Glow [94]. Such frameworks usually perform convolution operations as a series of dot-products, often with the assistance of a highly efficient BLAS library implementation to maximize parallelism and runtime efficiency. They also perform various graph-level and code-level optimizations to minimize the memory footprint and accelerate the network inference time. Among the existing open-source frameworks, Boda [95] is mainly designed to accelerate ConvNet inference using template meta-programming to generate specialized code for various GPU platforms. Boda provides a flexible platform to write meta-code, from which low-level code optimized for given hardware can be generated. For instance, we can unroll the loops, generate a long sequence of memory instructions, and handle different input regimes.



(a) ReLU $y = max(0, x)$ (b) Leaky ReLU $y = \begin{cases} 0.01x & x < 0 \\ x & x \geq 0 \end{cases}$ (c) Tanh $y = tanh(x)$ (d) Sigmoid $y = \frac{1}{1+e^{-x}}$ (e) Swish $y = \frac{x}{1+e^{-x}}$

Figure 3.5: An illustration of popular activation functions used in ConvNets.

3.4.1 Building blocks of convolutional neural networks

As depicted in Figure 3.4, various layers and functions are stacked up to build the final ConvNet. Although we are free to employ any computation layer to build our network, a limited set of relatively simple primitive operations exists that is often used in designing a ConvNet. For each computer-vision task, the choice of layer type, interconnections, and layer orderings might be different. Thus, a considerable amount of time and effort is required to explore the design space and ultimately finding the suitable network architecture. Neural architecture search (NAS) [96, 97, 98] is a trendy line of research aiming to find an efficient way to explore the design space of neural networks. For a more detailed description, please refer to Section 3.5.

In the following, we review the most essential computation layers/functions typically found in popular ConvNets.

Activation layers. Among all ConvNet layers, these layers are the simplest and least computationally intensive. Nonetheless, from a machine-learning perspective, they are critically important for introducing non-linearity to the output results. Otherwise, the neural network becomes a linear regression model. A wide variety of activation functions are used by deep-learning researchers, such as ReLU (Rectified linear unit), LeakyReLU, Tanh, Sigmoid, and Swish. Figure 3.5 illustrates these functions. Nonetheless, from a computational perspective, all these functions perform a function over each channel and pixel. Thus, the input and output have the exact dimensions. Additionally, the computation overhead of such functions is mostly related to reading the input and writing the output.

To avoid the unnecessary read/write overhead, we can fuse the activation layer to the previous layer. For instance, merging the activation layer with the convolution or fully connected layer is often possible. This way, we can save a set of extra data movements. Therefore, the overhead of activation layers becomes reasonably negligible compared to other layers, such as convolutions.

Pooling layers. Another source of non-linearity for modern deep neural networks is pooling layers. The principal idea behind such layers is motivated by *translation invariance*. Usually, convolution layers tend to record the precise location of available features in the inputs. Thus, small movements of such features result in a different output. Preferably, we want to avoid this problem and have a generalized solution for predicting the presence of a feature within the input image.

Pooling layers are the response to this problem. They use the sliding window approach to spatially pool each channel. Thus, depending on the pooling window size and stride, the output dimension will change. The most common flavors are *max pooling* and *average pooling*. The former replaces the

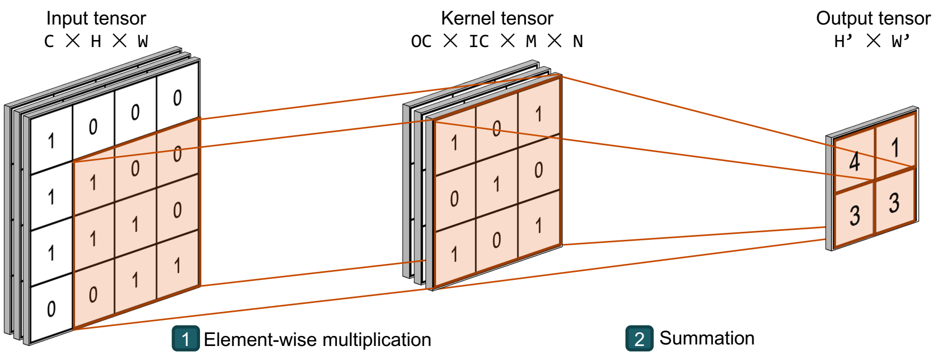


Figure 3.6: An illustration of a convolution operation.

values in the sliding window by the maximum value, while the latter averages all the values. The overall benefit of such layers is to intensify the effect of those channels/feature maps that are more useful for predicting the right label.

Convolutional layers. As briefly mentioned before, convolution operations are the main building block behind ConvNets. They are linear functions that apply a filter f on the input tensor. The output tensor is often called a feature map, as it contains the features extracted by the convolution layer. The semantics of a convolution operation is similar to pooling operations, where a sliding window is employed to apply the filter on the input, as shown in Figure 3.6. However, in contrast to pooling layers, convolutions apply a filter on all the input channels. Mathematically speaking, a convolution operation involves the dot-product of a set of weights (filter f) with the input. A dot-product is the element-wise multiplication of the filter and input values, which is then summed up to obtain a single value.

Compared with other layers, convolution layers are the most computationally intensive layers, as the value of each output pixel depends on all input channels. Concretely speaking, within a 2D convolution, the total number of FLOPs required for computing each output pixel is $IC \times K_x \times K_y$. The input and output feature maps of convolutional layers are three-dimensional matrices of size $H \times W \times C$, where H, W, and C are height, width, and the number of channels, respectively. Convolutional layers often use square kernels. Thus, a convolutional layer with kernel size K will have $K \times K \times C_{in} \times H_{out} \times W_{out} \times C_{out}$ MACCs, given that we do not count the bias values. As an example, for a 3×3 convolution with 128 filters, on a 224×224 input with 64 channels, we need to perform $3 \times 3 \times 64 \times 224 \times 224 \times 128 = 3.699.376.128$ MACCs.

Depth-wise separable convolution (DWConv). To make the convolution operations more efficient by requiring less memory and computation, we can factorize a regular convolution into two smaller ones, namely depth-wise and point-wise convolutions. Although such a factorization results in an approximation and not the original convolution output, the benefits outweighs the accuracy loss. In the worst case, we can stack multiple DWConvs up to get the same accuracy. Recent mobile-friendly ConvNets, such as MobileNet-v1/v2/v3, heavily use such layers to fit the model on memory-constrained mobile devices.

The depth-wise convolution is from many aspects similar to a normal convolution, except that we do not combine the input channels, causing the output channel to remain the same as the input channels. Thus, the total number of MACCs would be $K \times K \times C_{in} \times Hout \times Wout$, avoiding a factor of C_{out}. The other part is called point-wise convolution, which is a 1×1 convolution. The number of MACCs is $C_{in} \times H_{out} \times W_{out} \times C_{out}$, since $K = 1$.

Altogether, the DWConv needs $C_{in} \times H_{out} \times W_{out} \times (K \times K + Cout)$ MACCs. Thus, using the same example (a 3×3 convolution with $K = 128$, on a 64×224×224 input), we only need $64 \times 224 \times 224 \times (3 \times 3 + 64) = 176.619.520$. Clearly, the resulting MACCs value is way smaller than the MACCs required by a normal 3×3 convolution.

Fully-connected (FC) layers. As the name "fully-connected layer" suggests, all the inputs are connected to all the outputs. An FC-layer with M input and N output values requires $M \times N$ weights stored as a matrix W and a bias vector b. The following computation is then performed:

```
y = matmul(x, W) + b
```

Compared to a convolutional layer, an FC-layer demands a lot more memory and computing power. Since FC and convolutional layers are very similar, modern ConvNets do not use such layers anymore. In a convolutional layer, each output value is connected to $K \times K$ inputs rather than all of them. Thus, a convolutional layer is principally an FC-layer with most of the connections set to zero, making them much more efficient.

Batch normalization. Training a ConvNet is sensitive to random initial weights and the learning algorithm's setting. As the name "batch normalization" also suggests, this layer standardizes the inputs to a layer for each mini-batch. In modern ConvNets, it is very common to use batch normalization after each convolutional layer to stabilize the learning process and reduce the number of required training epochs.

Batch normalization receives the output of the previous layer (normally a convolutional layer) and applies Equation 3.3 to every single output value y:

$$z = \gamma * \frac{(y - \mu_B)}{\sqrt{\sigma^2 + \varepsilon}} + \beta, \tag{3.3}$$

This formula normalizes y by subtracting the batch mean for that output channel and divides by the standard deviation $\sqrt{\sigma^2}$. The addition of ε is a small value, often around 0.001, to avoid division by zero. γ and β are then used for scaling and adding bias. In modern networks, batch normalization is often applied after a convolution layer and right before the activation layer (e.g., $conv \rightarrow BN \rightarrow ReLU$). Since this operation is applied to each element in the output feature-map, we can simply merge the computation with the previous layer, avoiding extra data movement.

3.4.2 Winograd convolution

Various algorithms exist aiming at optimizing the computations of convolution operations. Since one of the contributions of this dissertation is based on the Winograd algorithm, we provide an in-depth explanation for such convolutions.

A. Toom [99] and S. Cook [100] originally proposed optimal filtering algorithms using polynomial residuals. Afterward, Shmuel Winograd generalized these algorithms and proposed a method for the efficient computation of finite impulse response (FIR) filters [101]. Within this algorithm, computing m outputs with an r-tap FIR filter, which is denoted by $F(m, r)$ for a 1D convolution, requires $m + r - 1$ multiplications. Such a reduction is quite significant in comparison with the direct method, which requires

$m \times r$ multiplications [102]. To explain how such a reduction can be achieved, we use $F(2,3)$ as an example. For instance, for an input vector $d = (d_0, d_1, d_2, d_3)$ and $g = (g_0, g_1, g_2)$, the Winograd algorithm transforms the input data and the filter to $v = (v_0, v_1, v_2, v_3)$ and $u = (u_0, u_1, u_2, u_3)$, respectively, using the following equations:

$$v_0 = d_0 - d_2, \qquad u_0 = g_0 \tag{3.4}$$

$$v_1 = d_1 + d_2, \qquad u_1 = \frac{g_0 + g_1 + g_2}{2} \tag{3.5}$$

$$v_2 = d_2 - d_1, \qquad u_2 = \frac{g_0 - g_1 + g_2}{2} \tag{3.6}$$

$$v_3 = d_1 - d_3, \qquad u_3 = g_2 \tag{3.7}$$

Then, we multiply u and v element-wise and store the result in $c = u \odot v$, such that each element within c is denoted as $c_i = u_i \times v_i$. Lastly, the final result $y = (y_0, y_1)$ is computed using the following equation:

$$y_0 = c_0 + c_1 + c_2, \qquad y_1 = c_1 - c_2 - c_3 \tag{3.8}$$

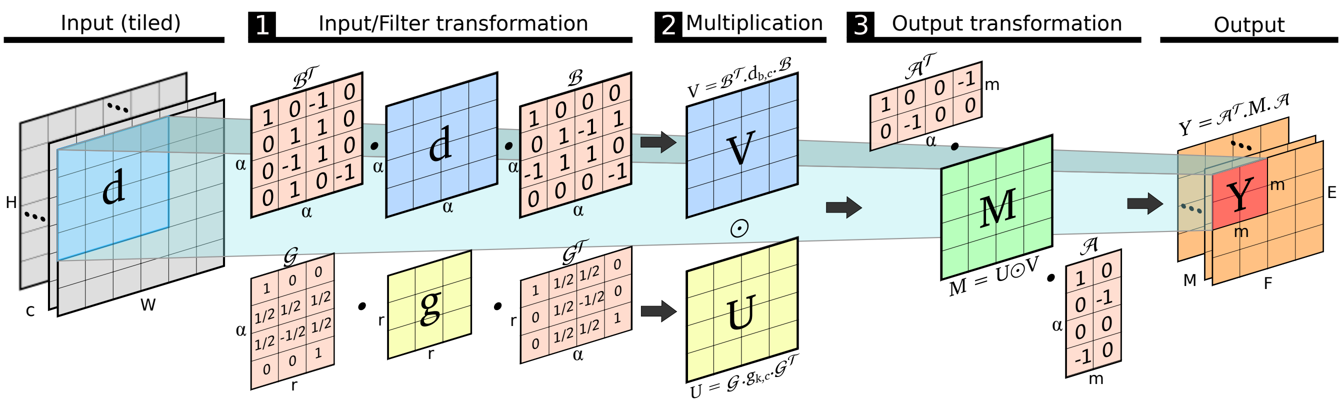
The Winograd algorithm generalizes the transformations mentioned above and summarizes all these steps into a single equation, such that $y = \mathcal{A}[(\mathcal{G}g) \odot (\mathcal{B}d)]$, where the transformation matrices for $F(m,r)$ are:

$$\begin{gathered} \mathcal{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}, \\ \mathcal{G} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}, \mathcal{B} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \end{gathered} \tag{3.9}$$

Thus, the Winograd algorithm consists of three main stages, of which the first and the last stages perform domain transformations. All transformations are done by matrix multiplications using pre-computed matrices ($\mathcal{A}$, $\mathcal{G}$, and $\mathcal{B}$) for each transformation. These matrices are fixed and are usually generated using the Toom-Cook method with a set of heuristically chosen polynomial points [103]. In a given $F(m^2, r^2)$ 2D Winograd convolution, filter size r and the output tile size m define the internal tile size (i.e., $\alpha = m + r - 1$), which in turn determines the shapes and values of the transformation matrices. Thus, each Winograd algorithm with distinct α demands a particular set of transformation matrices. Although the convolution operation determines the filter size r of the Winograd algorithm, the output tile size m can be freely chosen. Theoretically, by choosing a larger m, we can save more operations in the element-wise matrix multiplication step. However, it causes Winograd transformations to involve more elements, allowing floating-point rounding errors to jeopardize their numerical stability.

Figure 3.7 depicts all three stages of the computation for a sample Winograd convolution $F(2^2, 3^2)$. Before we explain each step, we define the following symbols, which are used by a Winograd convolution $F(m,r)$:

- $g_{k,c}$: k-th filter with channel c
- $d_{c,b}$: b-th input tile of c-th channel
- $Y_{k,b}$: b-th output tile for the k-th filter
- m: Winograd's output tile size
- r: kernel or filter size
- $P := N\lceil H/m \rceil \lceil W/m \rceil$: number of internal image tiles



Input (tiled)
1 Input/Filter transformation
2 Multiplication
3 Output transformation
Output
d
V
U
M
Y
g
$V = \mathcal{B}^T.d_{b,c}.\mathcal{B}$
$U = \mathcal{G}.g_{k,c}.\mathcal{G}^T$
$M = U \odot V$
$Y = \mathcal{A}^T.M.\mathcal{A}$
H
W
C
E
F
M
α
r
m

Figure 3.7: Visual representation of the computational steps within a sample $F(2^2, 3^2)$ Winograd convolution.

- $\alpha := m + r - 1$: Winograd's internal working tile size
- $\mathcal{G}$, $\mathcal{B}$, and $\mathcal{A}$: transformation matrices for input, filter and output, respectively

Input and filter transformation First, the input is decomposed into $\alpha \times \alpha$ tiles with the vertical and horizontal stride of $\alpha - r + 1$. This stride causes neighboring tiles to overlap by $r - 1$ elements. Each input tile and filter is then transformed by two transformation matrices $V = \mathcal{B}^T d_{c,b} \mathcal{B}$ and $U = \mathcal{G} g_{k,c} \mathcal{G}^T$.

Matrix multiplication The main computation happens in this stage, where element-wise matrix multiplication is used for multiplying the transformed filter U with the transformed input of the same channel V. Then, all channels of the same image should be summed up ($M = \sum_{c=1}^{C} U_{k,c} V_{c,b}$).

Output transformation Finally, similar to the first stage, the output tiles are transformed back into the original space as $Y = \mathcal{A}^T M \mathcal{A}$. The $\alpha \times \alpha$ tiles are transformed into $m \times m$ tiles first, and then placed into the output image at their corresponding position.

To fully benefit from the Winograd convolution, we need to pick a suitable output tile size m, which meets the expected accuracy level and memory limitations. Furthermore, various code optimizations (i.e., data layout and fast matrix multiplication) are usually needed to improve the overall runtime performance [104, 105]. However, such optimizations depend on Winograd specifications and the target hardware platform. Such a challenging task can be addressed effectively using a compiler, capable of generating specialized code. Our proposed method in Chapter 4 follows this idea and is integrated into an inference framework.

3.5 Neural architecture search

Neural architecture search (NAS) is a design space exploration method for the automatic creation of deep neural networks. NAS offers to solve the following key challenges [106] when designing a deep-learning model:

Intractable design space. Given the combinatorial design space of neural networks, designing a deep model with various layers and each with a multitude of different parameters gets quickly out of hand. Thus, various NAS methods tend to suggest an optimal method to reduce the search space and find the candidate model in a feasible amount of time.

Nontransferable optimality. The efficiency of a given model depends on various parameters that are not necessarily tied to the model architecture. For instance, any modification of the input resolution or target device would affect the runtime efficiency of the model. NAS can effectively resolve this issue by quickly adapting the model before deployment.

Inconsistent efficiency metrics. Often, the accuracy is not the only efficiency metric for designing a model. Other metrics such as FLOPs, latency, and memory/power consumption could be more vital given the circumstances. Using NAS, it is possible to change the objective or even strike a balance between all these metrics.

Among various NAS methods such as random search, genetic search, and reinforcement learning, differential NAS [106] (a.k.a. super-networks) is the latest trend that attracted a lot of attention. Differential NAS is an enabling technique for simultaneous learning of the architecture of a neural network and its weights. It is achieved by transforming the NAS problem, typically discrete, into a continuous one, which can then be solved via gradient descent algorithms. FBNet [106], DARTS [107], ProxylessNAS [108], and Once-for-all [109] are among the well-known differential NAS methods.

3.6 Reinforcement learning

Reinforcement learning (RL) has gained tremendous momentum in the past decade with numerous successful use cases in robotics, computer games, and optimization algorithms. RL is one of the fundamental branches of machine-learning paradigms that do not need pre-defined labeled data. It attempts to follow the trial-and-error mechanism to learn how to take proper sequential actions in an environment to increase the reward function [110]. Conventionally, an RL-based method consists of various concepts and components. In the following, we briefly touch upon the most important concepts of RL.

Algorithm (agent): The agent is the main component that evaluates the state, performs the action, and analyzes the reward.

Episode (rollout): The whole sequence of states and actions that are played until we reach the termination state.

Current (s_t) and next states (s_{t+1}): The representation of the world for the current and one after applying the action.

Action a: Any change that the RL agent performs that changes the state is called an action.

Policy $\pi(s, a)$: A deterministic or stochastic mapping between each state to an action.

Reward r: A function that defines the reward after taking an action. The goal of the RL agent is to optimize the cumulative reward.

3.7 Graph convolutional networks (GCN)

Many AI-powered tasks such as computer vision, machine translation, and speech recognition that use various deep-learning models deal with Euclidean space. However, various datasets do not fit in classical geometry and cannot be represented in the Euclidean space [111]. One of the most well-known examples are graphs with their extensive use case in social networks, search engines, and even in molecular physics and chemistry. Unlike typical inputs that can be represented by numerical values, graphs represent entities and their relations using nodes and edges. Thus, to perform deep-learning on graph datasets, a new technique is required.

Similar to ConvNets, graph convolutional networks perform similar operations, where the model learns graph features by inspecting the nodes, neighboring nodes, and edges. In other words, GCNs are a generalized version of ConvNets that work on the unstructured data format. Often, the scale of graph analysis done by GCNs is categorized into node-level, edge-level, and graph-level. As the names suggest, the extracted features from each category contain different levels of information useful for node, edge, and graph classification. Moreover, GCNs can be implemented using spectral and spatial methods. Spectral-based GCNs assume that graphs are undirected and fixed-size, making their application scope very limited. On the other hand, spatial-based GCNs do not have this limitation and use a technique similar to convolution operations to extract the information from neighboring nodes to compute a representation for a given node.

4 Performance portability: Efficient and performance-portable ConvNet deployment

The high development cost in providing performance portability is a paramount obstacle, particularly in the practical deployment of ConvNet models on GPU platforms. Although the semantics of ConvNet operations can be easily defined and implemented in popular programming languages, efficient GPU implementations require many programmer-years of effort. Not to mention that moving to another GPU platform demands further code optimizations and additional programming effort to keep up the performance.

In this chapter, following the importance of proper performance optimization (see Section 2), we present our second contribution, named Boda+, to address the concerns above by generating efficient, portable, and accurate ConvNet computation code. Instead of focusing on parallel programming aspects, such as language and compiler design, we propose a more pragmatic approach by utilizing existing languages and compilers available on the target platform. Particularly, we leveraged the template metaprogramming technique in order to alleviate the developers' effort. At the end of this chapter, we show that our approach represents a novel trade-off between portability, efficiency, and productivity. In summary, this chapter aims to answer the following research questions:

- Is there a novel way to target various GPU platforms using a single sourcecode?
- How can we strike a fair balance between development productivity and runtime performance?
- To which degree new convolution operations, such as Winograd, can accelerate ConvNet runtime?
- Is it possible to write an optimization recipe and automate the necessary code adjustments for a given operation?

4.1 Motivation

Although ConvNets are created through a combination of different layers (e.g., convolutions, pooling, and non-linear activation functions), convolution layers alone are used abundantly across the whole network and typically take up to 90% of the total runtime [10]. Such layers, particularly small ones with 3×3 filter sizes, are the main constituents of modern deep networks, as they achieve higher accuracy with fewer parameters than shallow networks with larger filters [112, 113]. Therefore, speeding them up would greatly alleviate the inference time and promote the usage of ConvNets. Thus, we also aim to accelerate such layers via optimized algorithms and obtain reasonably high performance on a wide variety of GPUs. Nevertheless, we are aware of various, often contradictory, concerns that require careful consideration. Figure 4.1 illustrates the four main concerns along with their contradictory nature. On the one hand, we have efficiency and deployment cost, and on the other hand, portability and development cost. Thus, an appropriate solution would need to find a balance for such a tradeoff. In the following, we touch on the main motivations behind our method by answering three main questions.

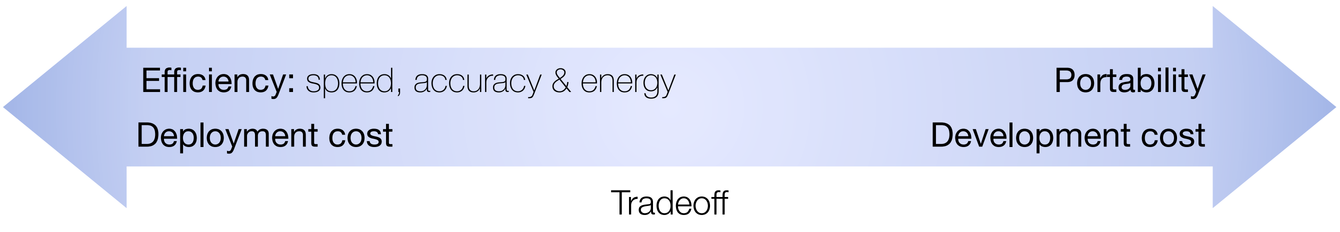


Figure 4.1: Tradeoff between key concerns in designing ConvNets.

4.1.1 Why do we need different convolution types?

Direct convolution is the most straightforward implementation variant with its computational requirements often exceeding available resources on commodity hardware. Therefore, to further speed up the convolutional layers, new algorithmic improvements had to be introduced. For instance, GEMM convolution is a method to express a convolutional layer as a matrix multiplication operation and benefit from hardware intrinsics to accelerate the computations. Winograd's minimal filtering algorithms are another attempt to minimize the number of arithmetic operations for performing small convolutions [102] (as introduced in Section 3.4.2). The key idea is similar to the FFT-based convolution, where multiplication in the frequency domain corresponds to convolution in the time domain. FFT convolution transforms the input into the frequency domain using discrete Fourier transformation (DFT), multiplies it by the frequency response of the filter, and then transforms it back into the time domain using inverse DFT [114]. The Winograd convolution follows the same principle. Inputs and filters are first transformed into another space before the element-wise multiplication. After the multiplication step, the output will be transformed back to the original pixel space to obtain the final result. Unlike the FFT-based convolution, which uses complex numbers, all arithmetic operations of the Winograd convolution use real numbers, thus requiring fewer operations [102]. Lavin and Gray [102] showed that these algorithms could be around 2$\times$ faster than the direct convolution. However, the results are not as accurate as of the direct method due to the additional floating-point rounding errors.

As explained above, we have various algorithms to implement a convolution operation, namely (1) direct, (2) GEMM, (3) FFT, and (4) Winograd convolution. Each algorithm can have a specialized implementation or variant. For instance, manually optimized convolution for the layers with $filter\_size = 1$ can yield additional performance. Additionally, each variant works differently depending on the convolution specifications, such as filter and input size. For instance, for small 3$\times$3 convolutions, the Winograd convolution often provides better performance, however, at the cost of additional memory consumption. Thus, both the convolution specifications and hardware constraints should be considered when selecting the right variant.

4.1.2 How can we make ConvNets more efficient?

As already explained in Chapter 2, a critical factor for developing robust shared-memory applications is the efficient use of cache and thread communications [42]. Furthermore, inappropriate data structures, algorithm design, and inefficient data locality may result in superfluous communication between threads/cores and severe performance problems. Within the context of ConvNets, convolutional layers are inherently linear-algebra computations and can be viewed as a generalization of matrix-matrix multiplication. Thus, we used our code optimization findings from Chapter 2 and aimed to generate efficient convolution operations.

Listings 4.1.2 and 4.1.2 show sample matrix multiplication and convolution C-pseudocodes, respectively. Note that both functions iterate over output elements in their outer set of loop nests and then iterate over the elements of a dot-product in their inner loop nests. In order to compute convolutions using matrix-matrix multiply, we only need to flatten all the inner loops (which iterate over all values in the filter and corresponding input window) and treat them as the K dimension. Then, we flatten the `out_x` and `out_y` dimensions and treat the combination as the M dimension. Hence, many of the same algorithms and optimizations used for matrix-matrix multiplication also apply to convolutions [10]. However, instead of three dimensions (M, N, and K) as in the case of matrix-matrix multiplication, a convolution operation has a considerably larger and more complex input space, such as:

- Small/enumerated integers (e.g., padding, stride, kernel size),
- Large integers (e.g., input X and Y sizes, input and output number-of-channels), and
- Booleans (i.e., fusing activation function with the output).

```
// the first 2 loop nests iterate
    over all elements of C

for(n=0; n<N; ++n) {
  for(m=0; m<M; ++m) {
    // the remaining loop nest
    calculates the dot-product for a
     single output element
    C[m][n] = 0;

    for(k=0; k<K; ++k) {
      C[m][n] += A[m][k] * B[k][n];
    }
  }
}
```

Listing 4.1: Matrix-matrix multiplication.

```
// the first 3 loops iterate over all output pixels
for(out_chan=0; out_chan<out_chan_sz; ++out_chan) {
  for(out_y=0; out_y<out_y_sz; ++out_y) {
    for(out_x=0; out_x<out_x_sz; ++out_x) {
      // the remaining 3 loop nests calculate the dot
     -product for a single output pixel
      out[out_chan][out_y][out_x] = 0;
      for(in_chan=0; in_chan<in_chan_sz; ++in_chan) {
        // wind_[x/y]_sz are the filter size (1,3,5)
        for(wind_x=0; wind_x<wind_x_sz; ++wind_x) {
          for(wind_y=0; wind_y<wind_y_sz; ++wind_y) {
            out[out_chan][out_y][out_x] += filters[
     out_chan][in_chan][wind_y][wind_x] * in[in_chan][
     out_y+wind_y][out_x+wind_x];
}}}}}}
```

Listing 4.2: Convolution operation.

Varying the input values leads to different runtime behavior. For instance, the data reuse patterns for kernel sizes 1, 3, and 11 are quite different. Currently, the programmers' approach to performance tuning is manual code restructuring and applying compiler optimizations for each target platform. Clearly, such a task demands extensive programming effort and experience. However, given that only a few sparse points in a convolution's input space are needed for any particular application, convolution operations can particularly benefit from metaprogramming to be specialized for each use case. Therefore, we can automatically generate an efficient convolution kernel without individual code manipulation on each platform.

4.1.3 How can we achieve performance portability in ConvNets?

Despite benefiting from rewarding code optimizations, deploying convolution operations on different types of GPUs involves various challenges. The first fundamental challenge is the efficient implementation of the GPU kernel itself. It requires deep knowledge of the operation's semantics and the target GPU for effective code optimization. For instance, performance engineers often develop multiple versions of the Winograd convolution, each supporting a different filter and output tile size. Such a rigid design prevents machine-learning compilers from using Winograd convolutions more effectively for layers with different specifications, as the flexibility in choosing the output tile size is essential for achieving higher speedups. Within our experiments, we observed that inference frameworks and engines (e.g., cuDNN) often pick Winograd convolutions for a limited number of convolutional layers.

We learned that attaining the highest achievable performance across various platforms requires endless rounds of manual code tuning to specialize the code to new hardware specifications. Porting the code to other GPU platforms exacerbates the problem and it is, indeed, our second underlying challenge. To overcome the code portability challenges [115], the following high-level tasks need to be addressed:

Syntax incompatibilites among GPU programming APIs. Discrepancies across GPU architectures and programming interfaces, such as CUDA, OpenCL, and Vulkan, make the efficient execution of tensor operations, which are the constituents of ConvNets, a challenging task. Among those, CUDA and OpenCL are the two widely used programming models that alleviate the effort required to harness the compute power of GPUs. While CUDA works only on Nvidia devices, OpenCL and Vulkan have been designed with portability in mind to run on any compatible device. Nonetheless, performance is not necessarily portable [116]. Furthermore, some vendors, such as Nvidia, are reluctant to fully support OpenCL as they see it as a rival to their own standard. This becomes even worse on a number of mobile GPUs for which there is no official support.

As a remedy, the Khronos group released a new programming API called Vulkan [117] along with an intermediate language named SPIR-V [118] to address the portability of GPU programs. Vulkan is inherently a low-level cross-platform graphics and compute API, much closer to the behavior of the hardware, resulting into a more efficient execution on modern GPUs. Unlike others, Vulkan is supported by all major mobile and desktop GPUs. This single feature makes Vulkan a more attractive programming interface compared with OpenCL, not to mention its unique low-level optimizations. However, such worthwhile features come at a price, as it requires significantly higher programming effort. Particularly for newcomers, rewriting their code with Vulkan is a cumbersome task.

Hardware-specific constraints. Each GPU is equipped with a different memory size and might have a specific memory hierarchy (e.g., global, shared, constant, and texture memory), data access method, and hardware execution primitives, such as multiply-accumulate (MAC) instructions.

Data movement. Memory bandwidth is the main factor behind the memory wall and impedes the increase of performance beyond a certain point, especially on mobile and edge devices with low memory bandwidth. Therefore, the effective movement of data between off-chip and compute units is of great importance. Furthermore, locality-aware memory access is the key for efficient ConvNet runtime, as inefficient data access patterns or unnecessary memory transfers may cost even more than the time saved by performing fewer computations.

Runtime specialization. It is crucial to determine when, where, and how the computations take place. Specialized scheduling, resource management, and parallelization methods can effectively address these issues.

Managing overheads. GPUs are throughput-optimized processors that run threads in chunks (a.k.a. warps). Thus, minimizing the use and impact of conditionals, control flow, and indexing is crucial for avoiding warp divergence and potential performance loss.

In this chapter, we conduct a comparative analysis of CUDA, OpenCL, and Vulkan, which we call *target APIs* in the rest of this dissertation. We then use the outcome to design our inference framework named Boda+ and propose an abstraction layer that enables GPU tensor code generation using any of the target APIs. Equipped with metaprogramming and auto-tuning, our code generator can create multiple implementations and select the best-performing convolution on a given GPU. In the experimental results section, we will show that our proposed framework offers a performance-portable ConvNet code generator with minimal programming effort.

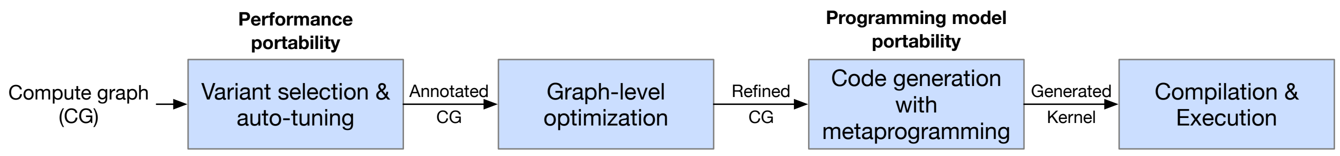


Figure 4.2: Boda framework workflow.

4.2 Background on Boda

Since our solution is based on the Boda framework [95], we provide a brief introduction and explain its main design principles. Boda employs general-purpose metaprogramming and dynamic compilation to generate specialized code and thus acts similar to a traditional compiler. However, in contrast to a fully-fledged compiler, Boda does not intend to support general-purpose programming and avoids typical intermediary layers in the compilation flow, such as the LLVM IR [23]. An overview of the Boda framework for mapping ConvNet computations to GPU hardware is illustrated in Figure 4.2. A compute graph is the input to Boda, which goes through various components to be mapped into different target back-ends. Boda is front-end/back-end-agnostic and only requires the target platform to provide mechanisms for runtime compilation (for metaprogramming/specialization), memory allocation, code execution, and per-function-call timing (for profiling/auto-tuning). Conveniently, all modern GPUs support similar programming models and input languages, such as CUDA for Nvidia GPUs and OpenCL for other GPUs. However, Boda does not provide support for Vulkan-enabled devices.

Variant selection and auto-tuning Convolution operations have a wide range of possible input sizes and parameters. Even with metaprogramming, creating a single function that optimally runs on a broad range of inputs and platforms is almost impossible. Thus, Boda expects the user to develop multiple variants of certain operations, such as convolution. Additionally, each variant might have various tuning parameters that affect code generation and runtime performance. For instance, Boda receives the variants and their values as `MNt=4:4,MNb=16:16,Kb=4,vw=4`. These parameters specify 4×4 register blocking, 16×16 thread blocking, an inner-loop-unroll-factor of 4, and a vector/SIMD width of 4. Boda's auto-tuning component automatically selects the optimal value for these parameters using a brute-force search to (1) find the best parameters for a given operation, as well as (2) learn much about a new target platform.

Graph-level optimization Convolution operations are commonly followed by element-wise activation functions. Often, the overhead to read and re-write the output tensors to apply the activation function is relatively high. As a remedy, merging the code for the activation function into the output-writing portion of the convolution operation can avoid unnecessary extra memory accesses. Therefore, Boda performs the fusion of adjacent convolution and activation operations.

Another optimization is the insertion of data-format-conversion operations, which is necessary as some variants may use different layouts or padding of their input or output tensors. Since we are able to choose the internal tensors format freely, Boda can exploit this optimization to achieve higher efficiency within each variant.

Code generation with metaprogramming Boda provides programming model portability based on the cross-compatible intersection of CUDA and OpenCL and forms a custom-made language named CUCL. However, CUCL is not a new language and just abstracts away the syntactical differences between CUDA and

OpenCL. The performance portability issue is not addressed at this layer and Boda tackles the performance optimizations via metaprogramming and auto-tuning. Boda allows the user to write only mildly restricted native GPU code in CUCL. Compared to directly using CUDA or OpenCL, CUCL provides language-neutral idioms, requires all functions' tensor arguments to be decorated with their dimension names, and requires access to tensor metadata (sizes, strides) to use a special template syntax: `%(myarray_mydim_size)`. To produce OpenCL or CUDA functions from a CUCL function template, the framework replaces CUCL idioms with OpenCL or CUDA ones. Additionally, Boda replaces references to tensor sizes and strides with either constant values for the specific input tensor sizes or references to dynamically-passed tensor metadata. Typically, Boda cares most about the case where the sizes are replaced with constants, as this gives the most possibility for optimizations and, therefore, efficiency. Then, for general metaprogramming support, Boda employs a string-template-based approach, using the framework's host language (C++) to write code-generators that set the values of string template variables inside CUCL function templates.

4.3 Comparison of CUDA, OpenCL, and Vulkan

CUDA and OpenCL share a range of core concepts, such as the platform, memory, execution, and programming model. Furthermore, their syntax and built-in functions are fairly similar to each other. Thus, converting a CUDA kernel to an OpenCL version and vice versa is relatively straightforward [119, 120]. On the other hand, Vulkan does not fully conform to CUDA and OpenCL standards, as it is geared both towards general-purpose computation and graphics while being portable and efficient. Various OpenCL offline compilers exist for converting C code to an intermediate language, from which later platform-specific assembly code can be easily generated. In contrast, Vulkan is able to target different platforms using a single input code and SPIR-V, a new platform-independent intermediate representation for defining shaders and compute kernels. Currently, SPIR-V code can be generated from HLSL, GLSL, and C with OpenCL.

Vulkan has been designed from scratch with asynchronous multi-threading support [121, 122]. Moreover, each Vulkan-capable device exposes one or more queues that can also process work asynchronously to each other. Each queue carries a set of *commands* and acts as a gateway to the execution engine of a device. These commands can represent many actions, from data transfer and compute-shader execution to draw commands. Each command specifies the requested action along with input/output data. The information about the available actions and the corresponding data is encapsulated in a so-called *pipeline*. This pipeline is then bound to a *command buffer*, representing a sequence of commands that should be sent in batches to the GPU. These buffers are created prior to execution and, to save time, can be submitted to a queue for execution as many times as required. However, creating command buffers is a time-consuming task. Therefore, the host code often employs multiple threads, working asynchronously, to construct command buffers in parallel. Once finished, a thread may submit these command buffers to a queue for execution. Right after the submission, the commands within a command buffer execute without any interruption in order or out of order—depending on the ordering constraints.

Despite these conceptual design differences, we prepared a mapping for the key concepts within each API in terms of memory regions and execution models in Table 4.1. The table shows that the memory hierarchy abstractions of the three interfaces are quite similar. Figure 4.3 illustrates the kernel execution space of the target APIs in more detail. Each point in the space is occupied by a thread/work-item/invocation. Each item is an execution instance of the kernel, with multiple of them combined into a thread block or group. The whole execution space is called grid or NDRange. Note that this mapping only covers the concepts shared among these APIs and does not fully cover the features of Vulkan.

Further comparison shows that Vulkan is more explicit in nature rather than depending on hidden heuristics in the driver. Vulkan provides more fine-grained control over the GPU on a much lower level.

		CUDA	OpenCL	Vulkan (SPIR-V)
Memory region		Global mem.	Global mem.	CrossWorkGroup, Uniform
		Constant mem.	Constant mem.	UniformConstant
		Texture mem.	Constant mem.	PushConstant
		Shared mem.	Local mem.	Workgroup
		Registers	Private memory	Private memory
Execution model		Thread	Work-item	Invocation
		Thread block	Work-group	Workgroup
		Grid	NDRange	NDRange

Table 4.1: A comparison of the terminology used in CUDA, OpenCL, and Vulkan.

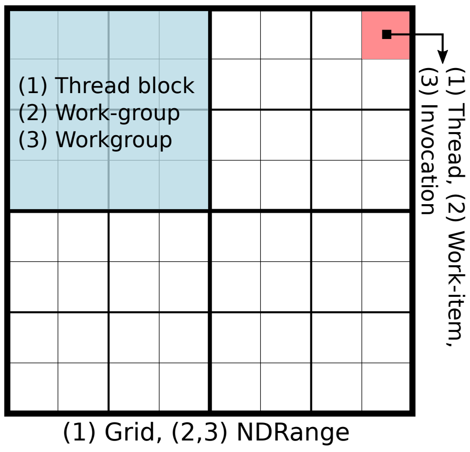


Figure 4.3: Kernel execution space for (1) CUDA, (2) OpenCL, and (3) Vulkan.

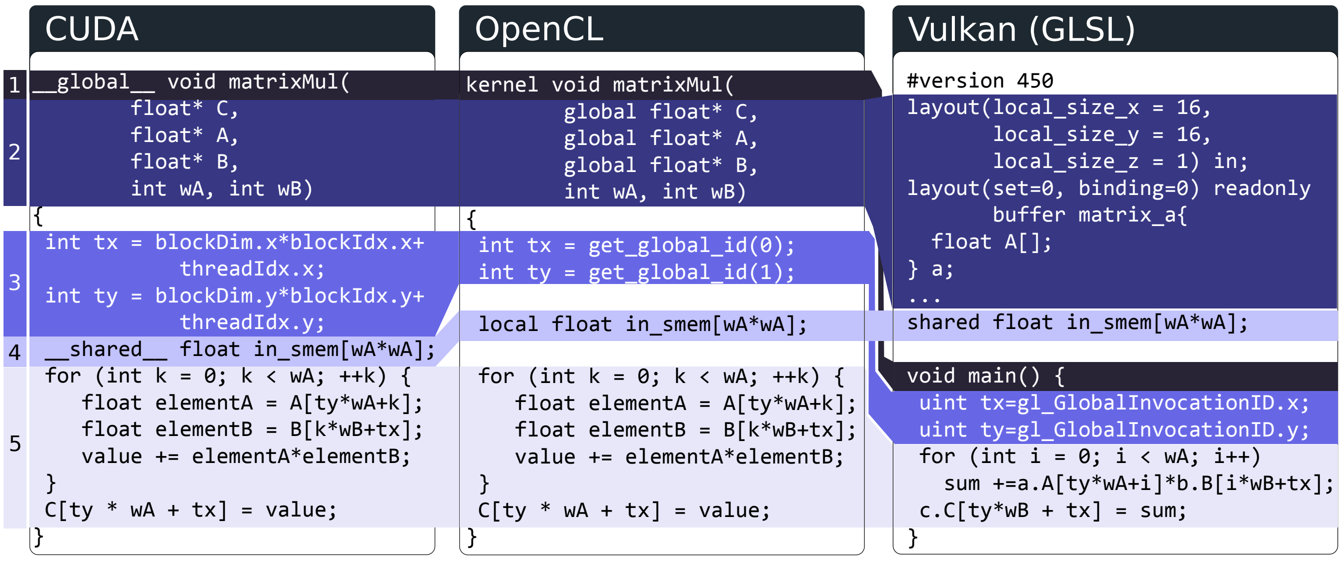
This enables programmers to enhance performance across many platforms. Even though such privilege comes with an extra programming effort, this feature can immensely increase the overall performance. Operations such as resource tracking, synchronization, memory allocation, and work submission internals benefit from being exposed to the user, making the application behavior more predictable and easier to control. Similarly, unnecessary background tasks such as error checking, hazard tracking, state validation, and shader compilation are removed from the runtime and instead can be done in the development phase, resulting in lower driver overhead and less CPU usage [122] compared with other APIs.

Particularly, synchronization mechanisms require the developer to be explicit about the semantics of the application but, in return, save a significant amount of overhead. While other APIs tend to insert implicit synchronization primitives between invocations and constructs, such as kernel executions and buffer reads, Vulkan is asynchronous by default. All synchronization between kernels or buffer I/O must be added explicitly to their respective command buffer via built-in synchronization primitives, including fences, barriers, semaphores, and events. Therefore, if no synchronization is required, we can strictly avoid the overhead of such operations.

Another difference is how Vulkan allocates memory, both on the host and the device. While CUDA and OpenCL often provide a single device buffer type and primitive functions for copying data between the host and device buffers, Vulkan puts the programmer in full control of memory management, including buffer creation, buffer type selection, memory allocation, and buffer binding. Furthermore, by making an explicit distinction between host-transparent device buffers and device-local buffers, we can implement explicit staging buffers or decide if they are not necessary—either because the amount of I/O to the buffer is negligible or because host memory and device memory are actually shared, as it is the case on many mobile platforms. Such explicit querying and handling of the underlying hardware can reduce unnecessary work and utilize the hardware more efficiently.

4.3.1 Programming conventions

In contrast to other APIs, Vulkan has its own programming conventions. Therefore, code similarities might not seem evident at first glance. Figure 4.4 shows a naïve matrix-multiplication kernel implemented using each programming interface. For Vulkan, we chose GLSL as our kernel language because of its better compatibility. We trimmed off some parts of the code for brevity. Regions with the same color and number



CUDA

```
__global__ void matrixMul(
        float* C,
        float* A,
        float* B,
        int wA, int wB)
{
 int tx = blockDim.x*blockIdx.x+
            threadIdx.x;
 int ty = blockDim.y*blockIdx.y+
            threadIdx.y;
 __shared__ float in_smem[wA*wA];
 for (int k = 0; k < wA; ++k) {
    float elementA = A[ty*wA+k];
    float elementB = B[k*wB+tx];
    value += elementA*elementB;
 }
 C[ty * wA + tx] = value;
}
```

OpenCL

```
kernel void matrixMul(
        global float* C,
        global float* A,
        global float* B,
        int wA, int wB)
{
 int tx = get_global_id(0);
 int ty = get_global_id(1);

 local float in_smem[wA*wA];

 for (int k = 0; k < wA; ++k) {
    float elementA = A[ty*wA+k];
    float elementB = B[k*wB+tx];
    value += elementA*elementB;
 }
 C[ty * wA + tx] = value;
}
```

Vulkan (GLSL)

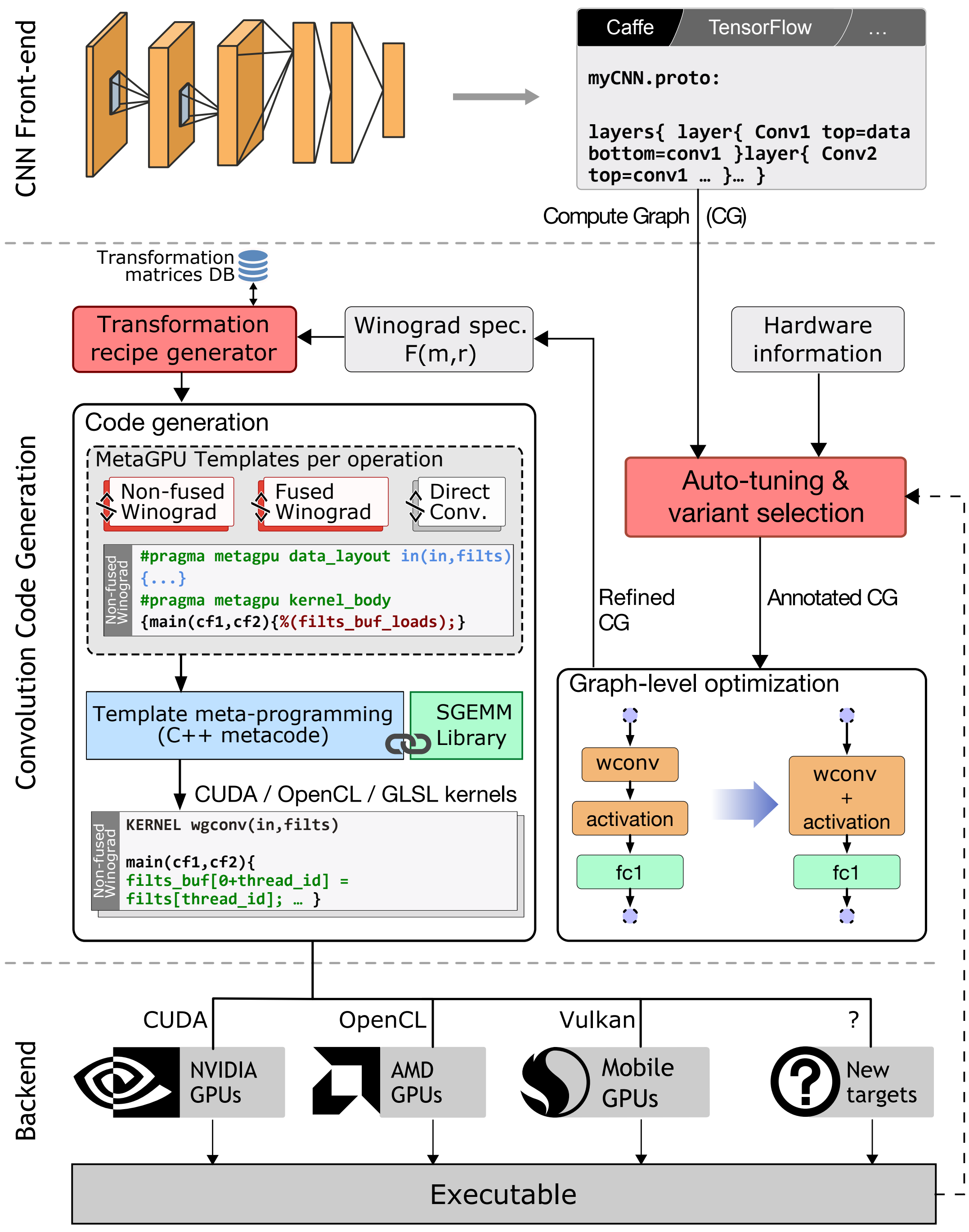
```
#version 450
layout(local_size_x = 16,
       local_size_y = 16,
       local_size_z = 1) in;
layout(set=0, binding=0) readonly
       buffer matrix_a{
  float A[];
} a;
...
shared float in_smem[wA*wA];

void main() {
 uint tx=gl_GlobalInvocationID.x;
 uint ty=gl_GlobalInvocationID.y;
 for (int i = 0; i < wA; i++)
   sum +=a.A[ty*wA+i]*b.B[i*wB+tx];
 c.C[ty*wB + tx] = sum;
}
```

Figure 4.4: An SGEMM kernel implemented with CUDA, OpenCL, and Vulkan (GLSL). Numbers on the left denote: (1) function declaration, (2) kernel arguments and data layout, (3) API-specific keywords, (4) shared-memory allocation.

share the same functionality. Syntactically, GLSL is similar to OpenCL and CUDA. However, GLSL is more restricted in specific ways, which requires rewriting some parts of the code. The biggest three differences are:

- Arguments to a kernel are not declared in the function header. Instead, they are declared in the global scope as so-called bindings, which can then be set with Vulkan. The compiler expects the entry function for the kernel to take no arguments. However, accessing the arguments within the kernel is the same as in other APIs.
- Workgroup dimensions have to be defined in the kernel and not in the host code. Moreover, each workgroup contains many work items or compute-shader invocations.
- GLSL does not provide explicit support for pointer objects. Instead, all pointers are represented as arrays of undefined length.
- Shared-memory objects are not declared within the kernel body. Instead, they are defined in the bindings.

Due to the conceptual discrepancies between Vulkan and the other APIs, the host code of Vulkan is radically different. For example, we can create a simple buffer in CUDA (`cudaMalloc`) or OpenCL (`clCreateBuffer`) with a single line of code. To create the same buffer in Vulkan, we have to: (1) create a buffer object, (2) get the memory requirements for that object, (3) decide which memory heap to use, (4) allocate memory on the selected heap, and (5) bind the buffer object to the allocated memory. This requires more than 40 lines of code. Clearly, host code programming in Vulkan is proportionally more complex, which stems from its explicit nature. Such code verbosity not only increases the programming effort but also makes the code more error-prone.



```
myCNN.proto:

layers{ layer{ Conv1 top=data
bottom=conv1 }layer{ Conv2
top=conv1 … }… }
```

```
#pragma metagpu data_layout in(in,filts)
{...}
#pragma metagpu kernel_body
{main(cf1,cf2){%(filts_buf_loads);}
```

```
KERNEL wgconv(in,filts)

main(cf1,cf2){
filts_buf[0+thread_id] =
filts[thread_id]; … }
```

Figure 4.5: The workflow of Boda+.

4.4 Boda+ framework

A generic yet portable implementation invariably involves low-level programming and a significant degree of metaprogramming [123, 124]. Thus, we embrace both of them in an attempt to create a system for generating efficient and portable convolution codes, specially Winograd convolution, with any specification.

To this end, we extended the Boda [95] framework and created a more comprehensive inference engine named Boda+ to generate tensor kernels for a wider variety of GPU programming APIs. Specifically, we aimed to use a single code template to support any GPU platform that supports CUDA, OpenCL, or Vulkan. We also added support for Winograd convolution code generation and uncovered the great potentials of Boda+ in operation-specific code specialization. Similar to Boda, our framework benefits from template metaprogramming approach to generate efficient GPU tensor code. However, our kernel-abstraction method is more structured, understandable, and generic. Relying only on metaprogramming made Boda+ a lightweight framework with minimal external software dependencies. The major required software packages are the C++ compiler and Python for building Boda+ itself, and a compatible GPU backend compiler, such as NVCC, Clang with OpenCL enabled, or GLSL to compile GPU tensor codes.

The input is a ConvNet model, which Boda+ parses as a computational graph suitable for graph optimization and variant selection. Figure 4.5 depicts a high-level overview of our approach. Once the framework picks a Winograd convolution according to the hardware and the convolution parameters, we start with the specification of the selected convolution, denoted by $F(m, r)$. First, we generate corresponding transformation matrices, after which we use Winograd templates to generate efficient code. Depending on the desired GPU platform, we can generate CUDA, OpenCL, or GLSL code. Finally, the resulting GPU kernels are compiled alongside their host code into a binary file, which can be executed in a standalone fashion on the target device. In the following, we explain the main components of our method in more detail.

4.4.1 MetaGPU abstraction layer

Given the considerable code discrepancies among the target APIs (see Figure 4.4), generating a GPU kernel out of a single code template might seem implausible at first sight. Nevertheless, we propose MetaGPU, a compatibility layer over our target APIs. It abstracts away the syntactic differences for the basic GPU programming concepts shared by our target APIs. This interface allows a specialized GPU kernel to be generated without modifying the main source code for every platform and convolution specification. We did not want to invent a new language because it creates additional learning overhead for programmers. Instead, we keep the coding convention very similar to CUDA and OpenCL and simply ask the user to separate the code into three regions using `#pragma` directives, similar to OpenMP. Figure 4.6 shows a MetaGPU code sample. In the following, the main regions within a MetaGPU template is described.

1. *Tuning parameters*: The first region defines tuning parameters. We can either access them in the kernel code or in the host program.

2. *Data layout*: The kernel arguments and required memories that need to be allocated in the shared memory are defined within this region. Additionally, the scope of each argument should be defined with any of `in`, `out`, or `shared` keywords.

3. *Kernel body*: As the name suggests, this region contains the actual kernel logic. A subtle difference is that using pointers is not allowed. Furthermore, the user has to use pre-defined keywords for accessing the GPU threads, workgroups, and synchronization barriers. Table 4.2 shows the list of keywords and their corresponding string in each target API. MetaGPU also supports template metaprogramming to generate adaptive code. Template placeholders are defined by `%(placeholder_name)%` and, using Boda, the user can populate them with C instructions or any desired string. Such a feature can help dynamically generate code and unroll loops to improve performance further.

1 Tuning parameters

```
#pragma metagpu tuning_knobs
{
  int wg_size_x;
  int unroll_lvl;
}
```

2 Data layout

```
#pragma metagpu data_layout \
in(a,b) out(c) shared(in_smem)
{
  float const * const a;
  float const * const b;
  float * c;
  float in_smem[%(dim)*%(dim)];
}
```

3 Kernel body

```
#pragma metagpu kernel_body {
 for(k=0;k<%(dim);k+=unroll_lvl){
   %(sm_loads);
   BARRIER_SYNC;
   %(inner_loop_body);
 }
}
```

Figure 4.6: A trivial sample of MetaGPU code.

Table 4.2: The list of pre-defined keywords in the kernel body alongside their corresponding value within each target API.

	CUDA	OpenCL	Vulkan
GLOB_ID_1D	blockDim.x*blockIdx.x+threadIdx.x	get_global_id(0)	gl_GlobalInvocationID.x
LOC_ID_1D	threadIdx.x	get_local_id(0)	gl_LocalInvocationID.x
GRP_ID_1D	blockIdx.x	get_group_id(0)	gl_WorkGroupID.x
LOC_SZ_1D	blockDim.x	get_local_size(0)	gl_WorkGroupSize.x
BARRIER_SYNC	__syncthreads()	barrier(CLK_LOCAL_MEM_FENCE)	barrier()

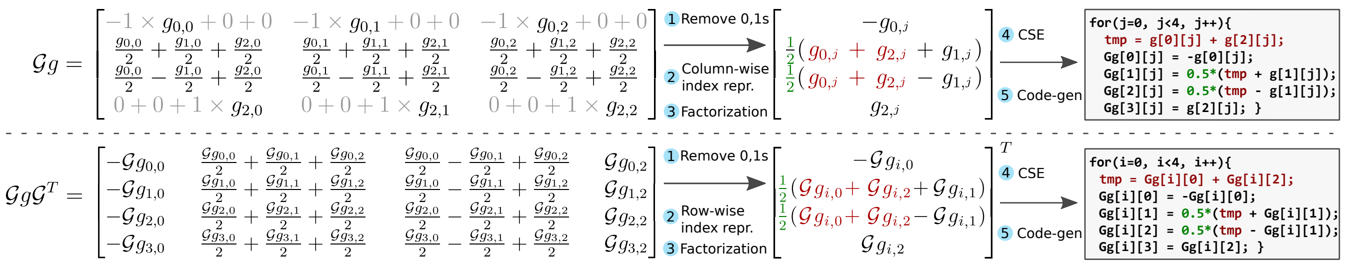
4.4.2 Winograd transformation optimization

As already explained in Section 3.4.2, small convolutions are the primary beneficiaries of the Winograd algorithm. Moving toward larger filters or output tiles makes the Winograd algorithm prone to low precision and low performance. The accuracy degrades after a multitude of data-scaling and floating-point operations with finite-precision [125, 103]. The performance also deteriorates because larger Winograd convolutions demand larger transformation matrices. Existing Winograd implementations attempt to perform a multitude of matrix multiplications in order to transform input tiles, filters, and outputs into the desired domain. The growing number of arithmetic operations involved in the transformation steps gradually becomes a burden. In this section, we provide a solution to optimize these steps and alleviate the adverse effects of the Winograd transformations.

Selecting polynomial points

A critical factor in improving the numerical accuracy of Winograd convolutions is to find a good set of polynomial points as the basis for generating transformation matrices. Inspired by B. Barabasz et al. [103], our method heuristically finds the polynomial points and uses the modified Toom-Cook method to generate transformation matrices—based on the idea of evaluating polynomials at given points using the Lagrange interpolation theorem [103].

For a Winograd convolution $F(m^2, r^2)$, we need $m+r-2$ points. We begin with the ordered set $(0, -1, 1)$, which has been proven to provide ideal points for reducing arithmetic operations and maintaining high accuracy because multiplication by 1 or -1 can simply be skipped, and multiplication by zero enables us to skip both the scaling and addition [103]. When more than three points are required, we perform an exhaustive search for the remaining points. Empirical evaluations show that small and simple integers and fractions are good candidates for reducing the required number of scalings and additions [103]. We follow the same approach and select the points P as rational numbers, where $P = \{\frac{a}{b} | a, b \in \mathbb{Z}, -9 \leqslant a \leqslant 9, 1 \leqslant b \leqslant 9\}$. To find the most accurate set of points, we iteratively examine the precision of Winograd convolution results.



```
for(j=0, j<4, j++){
  tmp = g[0][j] + g[2][j];
  Gg[0][j] = -g[0][j];
  Gg[1][j] = 0.5*(tmp + g[1][j]);
  Gg[2][j] = 0.5*(tmp - g[1][j]);
  Gg[3][j] = g[2][j]; }
```

```
for(i=0, i<4, i++){
  tmp = Gg[i][0] + Gg[i][2];
  Gg[i][0] = -Gg[i][0];
  Gg[i][1] = 0.5*(tmp + Gg[i][1]);
  Gg[i][2] = 0.5*(tmp - Gg[i][1]);
  Gg[i][3] = Gg[i][2]; }
```

Figure 4.7: Illustration of a Winograd filter transformation being optimized prior to code generation.

In each iteration, we create random input and filter tensors with a uniform distribution in the range of (-1, 1) because, in practice, the weights of deep neural networks are primarily concentrated in this range. To obtain the highest precision, we compare the results (FP32) with direct convolution (FP64) and compute the error rate using the L1 norm. We perform this analysis 10,000 times, a relatively high iteration count to make the results stable. We select the median value as the representative error rate of the chosen points.

Transformation recipe generation

Transformation matrices for a given Winograd convolution are always the same and often follow a regular pattern. Thus, we avoid running an ordinary matrix-multiplication kernel and, instead, we use symbolic computation to intensively simplify the transformation steps into a sequence of instructions for constructing the transformed matrices. First, we use the modified Toom-Cook method to generate the transformation matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{G}$ based on the selected polynomial points. One crucial feature is that we use rational numbers instead of real floating-point numbers to avoid rounding errors. Then, we create a symbol matrix with its size equal to the input size, similar to:

$$\mathcal{G} = \begin{bmatrix} -1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}, g = \begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} \\ g_{1,0} & g_{1,1} & g_{1,2} \\ g_{2,0} & g_{2,1} & g_{2,2} \end{bmatrix} \tag{4.1}$$

We multiply the transformation matrix with the symbol matrix and obtain the result. Next, we apply the following sequence of steps to optimize the transformed matrices:

1. **Elimination of unnecessary arithmetic operations:** We observe many multiplications by one or zero, and additions with zero (i.e. $1 \times g_{i,j} + 0$ or $0 \times g_{i,j}$), which we eliminate and simplify down to $g_{i,j}$ or 0.

2. **Column-/row-wise index-based representation:** We transform the resulting matrix into a vector with the variable subscripts replaced by an induction variable symbol. A 2D Winograd transformation consists of two consecutive matrix multiplications. We realized that we can always apply a column-wise and row-wise representation generalization to these multiplications, respectively. The ultimate goal is to generate the transformed matrix using only a single loop construct. Additionally, we can unroll the loops if necessary.

3. **Factorization:** Each row within the resultant vector contains terms with rational coefficients. In the case we find common coefficients across the terms, we apply factorization to save redundant multiplications.

4. **Common sub-expression (CSE) elimination:** We use the CSE algorithm to find the common terms among the vector rows. Thus, we can compute them once and reuse them multiple times. This method reduces both the number of additions and multiplications.

Figure 4.7 illustrates the above steps applied to the sample filter defined in Equation 4.1. These optimizations need to be performed only once before the actual Winograd convolution execution to obtain the transformation recipes. Since these recipes remain the same for every specific $F(m, r)$, we store them in a database to facilitate their reuse and avoid generating them again.

4.4.3 Code generation

At a high level, we choose to take a generic and flexible approach to metaprogramming. Rather than using language-level metaprogramming, we write code generators directly in C++. We use Boda's native support for tensors at the meta-code layer to allow code generation to exploit fixed, exact sizes for all inputs and outputs. As described in Section 4.4.1, MetaGPU enables us to abstract away the syntactical differences between our target APIs. Thus, we can use the same template code for generating CUDA, OpenCL, Vulkan kernel codes. We will evaluate the performance of the kernels generated using this approach in Section 4.5. However, in practice, we mostly use CUDA and the rather new Vulkan API to target the GPU platforms within our study. Particularly, we decided to use Vulkan instead of OpenCL on non-Nvidia GPUs, as it supports a broader range of GPUs, including mobile platforms. Furthermore, evidence shows that the Vulkan compiler can produce more optimized GPU codes compared with OpenCL compilers [43].

We first parse the input MetaGPU code and extract the three regions. The tuning parameters can later be used for auto-tuning. Then, the data layout of the kernel is parsed to find out the kernel arguments for CUDA/OpenCL code and the bindings for Vulkan GLSL code. Based on the target programming interface, we can then generate the kernel by generating corresponding argument declarations and merging them with the kernel body. We observed that platform-specific compilers often do not successfully unroll loops and remove unneeded conditionals. In such cases, we directly emit a sequence of instructions for iterating through tensor elements, loading and storing data from/to global memory, shared memory, and registers. To do this, we move the loop to the meta-code level and replace it entirely with a template placeholder, such as `%(filts_buf_loads)`, `%(winograd_filt_transform)`, and `%(store_results)`. Then, at the meta-code level, we write code to generate the required sequence of instructions.

In general, we aim to make the code as simple as possible by reducing the usage of loops and conditions. Such an optimization increases the chance that the platform-specific compiler generates efficient binary code. Furthermore, the code generator has the privilege to employ the highly-tuned BLAS libraries that exist on the target platform, such as cuBLAS and CLBLast [126]. In this study, we used CLBlast to perform the matrix multiplications.

We also added Vulkan support by creating a new backend to support host programming. All the required buffers, synchronizations, and timings will be handled by the Vulkan backend. Therefore, the end-user does not have to write any host code using Vulkan. Since the programming effort of Vulkan is very high, this feature will significantly enhance programmer productivity. Furthermore, we use the kernel batching feature in Vulkan and submit up to eight compute shaders at once to the GPU. We believe that this simple optimization will greatly reduce the kernel-invocation overhead.

Winograd transformation meta-code

In a 2D Winograd convolution, each transformation step should perform two consecutive matrix multiplications to transform a given tensor into the desired domain. Such multiplications are costly, and depending on the matrix dimensions, they might impose significant runtime overhead. We aim to replace the six matrix multiplications (i.e., $\mathcal{G}.g.\mathcal{G}^T$, $\mathcal{B}^T.d.\mathcal{B}$, and $\mathcal{A}^T.M.\mathcal{A}$) with their corresponding intensively simplified single-level loops. We replace each matrix multiplication code with a template placeholder, such as `%(winograd_filt_transform)`, and later use metaprogramming to fill in the placeholder with the transformation recipes that we introduced in Section 4.4.2. An example of the resulting code is illustrated in Figure 4.7. We further reduce the complexity and improve the performance of the transformation code using two optimization techniques:

- **Adaptive loop-unrolling:** We unroll the Winograd transformation loops to eliminate control instructions and achieve higher speedups. The unrolling factor is a tunable parameter, which we can tune according to available instruction cache size. For those loops in which the iteration count is not dividable by the unrolling factor, we find the closest divisor, or if we cannot find one, we fully unroll the loop.
- **Fused multiply-add (FMA) operations:** Often, the terms involved in computing the elements of transformed matrices contain a multiplication and an addition. Therefore, we can convert those operations into an FMA instruction and perform them all in one step, with a single rounding. We can benefit from FMA operations, provided that the target GPU and the programming interface support such operations. Otherwise, we avoid calling these instructions and simply rely on basic arithmetic instructions, instead.

Winograd code templates

We can implement Winograd convolutions in two different ways: (1) non-fused and (2) fused. A fused implementation is often preferable as it reduces the GPU memory transfer by merging all the Winograd steps into a single kernel. However, for large kernels, its memory requirement might exceed the available GPU shared memory space. In such cases, we use the non-fused implementation as a fallback. As a rule of thumb, fused implementations are often better suited for small convolutions such as 3×3 convolutions with small output tile sizes. We implemented code templates for both versions to demonstrate the applicability of each version under different circumstances.

Non-fused implementation. This version implies that we have a separate kernel for each step of the Winograd algorithm, and each kernel has to write the results back to the global memory. Although it increases the data transfer overhead, the non-fused version is still a viable option for larger Winograd convolutions, where the available shared memory is limited, particularly on mobile GPUs. In general, each kernel assigns a tile of data to each thread, which first loads the data from the global memory to its registers. Then, the actual computation takes place, and ultimately, the output will be written back into the global memory.

Despite the additional arithmetic operations caused by Winograd transformations, a significant portion of the computation takes place in Winograd's matrix multiplication step. Therefore, obtaining higher efficiency also relies on using optimized BLAS routines. The element-wise multiplication of the transformed input with transformed filters can be seen as a dot product of two vectors U and V, as we need to sum up the results across the channels. Therefore, we follow Lavin and Gray's [102] approach to pose this problem as a typical matrix multiplication and benefit from highly efficient SGEMM routines.

To reframe the problem as an SGEMM operation, we vertically stack each element of a transformed filter tile. Then, we group them by their index within the tile and sort them by their filter K in ascending order, such that each group denotes the K-th elements of each filter tile. Consequently, all channels will be horizontally stacked and placed in their corresponding row, such that the first row contains all channels of the first element within the first filter tile. The resulting $(\alpha^2 K, C)$ matrix is $U'_{ij} = U^{i/K}_{i\%K,j}$. The transformed image is reframed in the opposite way, where elements with the same index are horizontally stacked and grouped by the same image tile. Then, the channels are stacked in a column-wise fashion.

Since both matrices U' and V' are grouped by the index within their tile, there will be α^2 groups per matrix. Only the groups in U' and V' sharing the same tile index have to be multiplied, and only α^2 matrix multiplications are needed. Since we need to multiply several small matrices, we avoid invoking different matrix multiplication kernels and, instead, use a batched-SGEMM operation to perform all the multiplications. We use a vendor BLAS library if it exists on the target platform. Otherwise, we use a self-developed SGEMM kernel in the Boda framework.

Fused implementation. Merging all the Winograd steps into a single kernel has the potential advantage of improving the usage of shared memory and registers. Data resides in the shared memory as long as it is needed for computation. Previously, Lavin and Gray [102] proposed this optimization solely for small Winograd configurations, such as $F(3,2)$ and $F(3,4)$, since the shared memory space is very limited [102]. We further extended this optimization and used metaprogramming to support larger configurations, as long as enough shared memory is available. We split the threads within a thread block in half, such that the first half computes filter transformations, and the other half computes input transformations. The matrix multiplication step is divided into equal parts and distributed among the threads. Therefore, we cannot invoke a real matrix-multiplication kernel and, instead, implemented the multiplications within the kernel. Finally, all threads perform the output transformation together.

4.4.4 Variant selection and auto-tuning

Tensor operations have a wide range of possible input sizes and parameters. It is generally difficult, even with metaprogramming, to write code that runs well across more than a limited range of input sizes and hardware platforms. Such tuning parameters might control thread blocking, memory access patterns, or load/store/compute vector widths. Thus, the auto-tuner automatically searches the tuning space to find the right values for the given tuning knobs in the MetaGPU code and even across different implementation variants. This is an important step towards higher performance portability.

The key feature of our auto-tuning method is the automatic per-platform variant selection and automated sweeping over tuning parameters. We have two different variants of Winograd convolutions (i.e., fused and non-fused), which might perform differently on each target platform. Thus, we expect to run the best performing variant on a given platform, based on the convolution and available resources. Furthermore, each variant needs to be optimized prior to execution using several tuning parameters, as described in Table 4.3. By performing a brute-force, guided, or sampled exploration of the space of variants and tuning parameters, we can find the best parameters for a given Winograd convolution operation and provide performance portability among different hardware platforms. Considering the manageable size of the search space, we used the brute-force method. Nevertheless, the tuning process could be further accelerated using more sophisticated search methods.

4.5 Experimental results

Since the main contributions of Boda+ are GPU API abstraction and specialization of Winograd convolutions, we first assess the effect of MetaGPU on improving the productivity and programmability of GPU programming for tensor operations. Then, we provide analytical study results on the Winograd's accuracy and the impact of optimization recipes on the Winograd convolution. Lastly, we demonstrate the effectiveness of our method in generating efficient convolution kernels on various GPU platforms. We evaluate the portability

Table 4.3: Tuning parameters for Winograd convolutions.

Tuning Parameter	Purpose	Values
WV	Winograd variant (fused / non-fused)	[0, 1]
LU	Loop unrolling factor	$[1, 2, 4, 6, \infty]$
MNt	SGEMM Register blocking size	Exponential of two
MNb	SGEMM Thread blocking size	Exponential of two
m	Winograd output tile size	$2 \leq m \leq 10$

of the kernels that we generate using Boda+ and compare the performance with popular deep-learning libraries. We decided to study the results of Winograd convolutions separately, to emphasize the effect of having a specialized and optimized convolution flavor on the runtime performance. In this section, we specifically provide the answers to the following questions:

- What is the impact of Boda+ on programming productivity (i.e., programmability)?
- How precise are the Winograd convolution results, and how does the accuracy change for different Winograd configurations?
- To what extent can we optimize the Winograd transformation code?
- What is the runtime performance of the proposed method, and how does it perform compared with other deep-learning libraries?
- How portable are the generated kernels across different GPU platforms?
- Is there any way to pick the most suitable configuration for a given Winograd convolution?

Experimental setup

To evaluate the proposed method, we chose NVIDIA GTX 1080 Ti and AMD Radeon RX 580, two popular desktop GPUs. We also used a mobile platform based on the Hikey 960 development kit, which contains an ARM Mali-G71 MP8 GPU. Table 4.4 summarizes the configuration details of the target platforms.

4.5.1 Programmability analysis

Our method offers performance portability while easing the burden of rewriting the program for each API. However, to quantitatively evaluate the programming effort required to generate efficient deep-learning kernels, we propose a metric based on total lines of code. Inspired by Memeti et al. [127], we use cloc to determine the lines of MetaGPU code $LOC_{MetaGPU}$ and the total unique lines of code $LOC_{TotalUniqueLines}$ needed to be written for our target APIs to provide code portability. We then define the programming effort as follows.

$$\text{Effort}[\%] = (LOC_{MetaGPU}/LOC_{TotalUniqueLines}) \times 100 \tag{4.2}$$

In most ConvNet frameworks, including Boda, multiple convolution variants exist, each specialized for a specific case. For instance, Boda provides direct, tiled, GEMM, and 1×1 convolution variants. We counted

Table 4.4: Experimental setup.

	Nvidia GTX 1080Ti	AMD RX 580	ARM Mali G71 MP8
OS	Ubuntu 16.04 64-bit		Android 7.0
CPU	Intel Xeon Gold 6126, 12Core @ 2.6GHz		4 Cortex A73 + 4 Cortex A53
Host Memory	64 GB		3GB LPDDR4 SDRAM
GPU Memory	11GB GDDR5X	8GB GDDR5	-
Driver	Linux Display Driver 410.66	AMDGPU-PRO Driver 17.40	Native driver
CUDA	CUDA 10.0	-	-
OpenCL	OpenCL 1.2	OpenCL 2.0	OpenCL 2.0
Vulkan SDK	Vulkan 1.1.97	Vulkan 1.1.97	Vulkan 1.1.97
Libraries	cuDNN 7.3	MIOpen 2.1	ARM Compute Library v20.02.1

Table 4.5: Lines-of-code comparison for different convolution implementations alongside computed effort metric.

	$LOC_{MetaGPU}$	LOC_{CUDA}	LOC_{OpenCL}	LOC_{Vulkan}	$LOC_{TotalUniqueLines}$	Effort [%]
Direct convolution	113	562	631	1137	2330	4.84
Tiled convolution	115	548	618	1119	2285	5.03
GEMM convolution	89	1103	1172	1666	3941	2.25
1x1 convolution	160	1190	1259	1761	4210	3.80

the LOCs for each variant and target API. The results are shown in Table 4.5. For a fair programming effort analysis, we used total unique lines between all the target APIs. The results indicate that using our method requires on average 4% of the total effort needed to implement the code with all of the target APIs.

4.5.2 Winograd accuracy analysis

As mentioned earlier in Section 4.4.2, moving toward a larger internal tile size α, which itself depends on the output tile size m and the filter size r, leads to a higher accuracy loss. However, it is not apparent how much error is tolerable during the inference phase. Thus, we measured the accuracy of Winograd convolutions with various internal tile sizes to find out how significant their error rates are and which one is probably more suitable for a convolutional layer.

Table 4.6 reports the selected polynomial points for different Winograd internal tile sizes, within the range of $[4, 16]$ alongside their relative error. We compute the relative error using the L1 norm $||X||_1$:

$$RelativeError = \frac{||\widehat{X} - X||_1}{||X||_1},$$
$$||X||_1 = max_j \sum_i |a_{i,j}|,$$

where $\widehat{X}$ and X are the Winograd (32-bits) and direct convolution (64-bits) results, respectively. Conventionally, the previous points can be reused for adding a new point to the sequence [103]. However, we noticed that by recomputing the whole sequence of points, more accurate results could be obtained.

We investigated the numerical stability of the generated Winograd convolutions by measuring their error range and error growth rate. Figure 4.8 depicts a box plot of L1-norm errors for Winograd convolutions

Table 4.6: Polynomial points selected by our method alongside their relative error.

α	Points	Relative Error
4	$BP = (0, 1, -1)$	6.11×10^{-8}
5	$BP \cup (2)$	2.65×10^{-7}
6	$BP \cup (1/2, -2)$	5.59×10^{-7}
7	$BP \cup (1/2, -2, 2)$	1.14×10^{-6}
8	$BP \cup (2, -1/2, 1/2, -2)$	1.76×10^{-6}
9	$BP \cup (2, -1/2, 1/2, -2, 4)$	9.93×10^{-6}
10	$BP \cup (1/2, -2, 2, -1/2, 4/3, -3/4)$	1.42×10^{-5}
11	$BP \cup (1/2, -2, 2, -1/2, 4/3, -3/4, -4)$	8.38×10^{-5}
12	$BP \cup (1/2, -2, 2, -1/2, 3/4, -4/3, 9/2, -2/9)$	1.83×10^{-4}
13	$BP \cup (1/2, -2, 2, -1/2, 4/3, -3/4, 1/4, -4, 4)$	5.36×10^{-4}
14	$BP \cup (1/2, -2, 2, -1/2, 9/7, -7/9, 1/4, -4, 7/9, -7/9)$	9.10×10^{-4}
15	$BP \cup (1/2, -2, 2, -1/2, 4/3, -3/4, 1/4, -4, 7/9, -9/7, 4)$	3.45×10^{-3}
16	$BP \cup (1/2, -2, 2, -1/2, 4/3, -3/4, 2/7, -7/2, 4/5, -5/4, 4, -1/4)$	4.66×10^{-3}

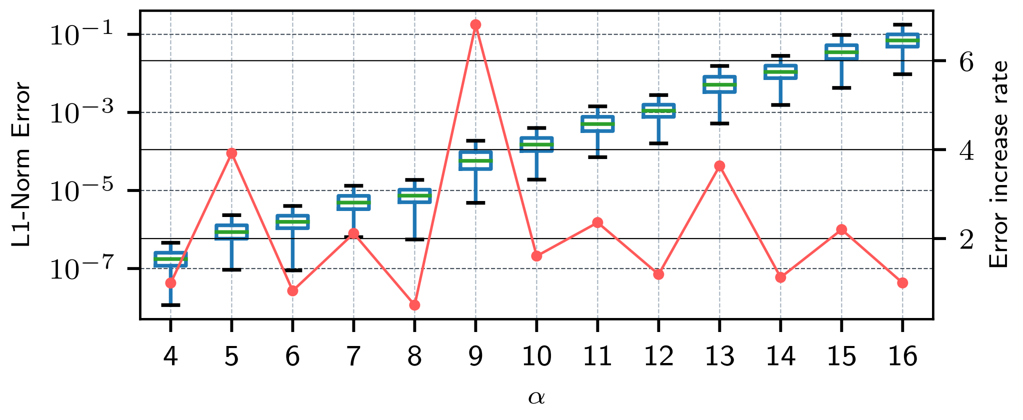


Figure 4.8: L1-norm error analysis for various Winograd internal tile sizes.

with different α. We ran each case 10,000 times with randomly generated input and filters between $(-1, 1)$. We observed that the error rates proliferate with the addition of each new polynomial point. However, it does not precisely follow an exponential trajectory, as opposed to the observation made by Barabasz et al. [103]. Instead, we noticed that Winograd convolutions with even α benefit from a lower error-growth rate. We observed the lowest error growth when $\alpha = 8$.

In comparison with the inference phase, accuracy is a more crucial factor for the training phase, as it affects learning stability. Nonetheless, according to previous studies [128, 129], error rates lower than $1e{-}02$ do not harm the stability, implying that the inference phase is immune to such error rates. Such an observation suggests that our generated Winograd convolutions can be used during inference without experiencing any instability.

4.5.3 Winograd transformation optimization results

In Section 4.4.2, we proposed a method for reducing the computational complexity of Winograd transformation steps using symbolic computation. To demonstrate the effectiveness of our method, we numerically analyze the computations involved in Winograd transformation steps by directly counting the number of additions and multiplications. We selected Winograd convolutions with $m \in \{m \in \mathbb{N} | 2 \leq m \leq 10\}$ and $r \in \{3, 5, 7\}$. The results are given in Figures 4.9a– 4.9c. For each step, we separated the results into three columns, each representing a particular filter size. The baseline is the straightforward implementation of Winograd transformations using typical matrix multiplications. The optimized version represents the actual number of arithmetic operations involved in the generated code. We also demonstrate the amount of FMA operations, which we were able to identify.

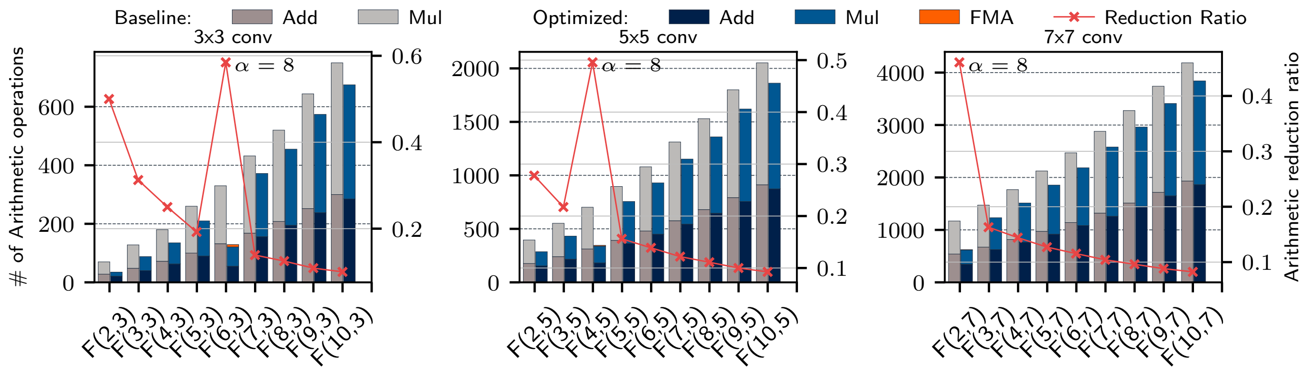
We observed that our method was able to reduce the number of arithmetic operations in transformation steps by up to 62%. We annotated the highest amount of reductions in each diagram. We often obtain the highest amount of reduction when $\alpha = 8$, except for a few cases, where other internal tile sizes yield higher reductions. However, when looking at all the transformation steps, as depicted in Figure 4.9d, we can conclude that transformations are better suited for optimization when $\alpha = 8$. Such a value for α enables our method to factorize more common terms and improve data reuse. Since transformation steps might be considered as a small portion of the total computations involved in Winograd convolution, we also show the total amount of arithmetic reduction in Figure 4.9d. As the blue line indicates, the total reduction of arithmetic operations can reach up to 40%. Overall, our analysis of both accuracy and the number of arithmetic operations in Winograd transformations after optimization confirms that when $\alpha = 8$, Winograd convolutions can be optimized to a greater extent.

To further validate the effectiveness of our method, we generated CUDA kernels for sample convolutions with $r \in \{3, 5, 7\}$ and the same set of Winograd output tile sizes $m \in \{m \in \mathbb{N} | 2 \leq m \leq 10\}$. Figure 4.10 depicts all the runtimes for the convolutions that we ran on our Nvidia GPU. Our results suggest that Winograd convolutions with a filter size larger than five are probably not suitable for deployment, as other types of convolutions perform much faster. We further noticed that larger values of m do not necessarily save more operations during the matrix-multiplication step, as they cause additional computation overhead. Evidence [104] suggests that this happens mainly for two reasons: (1) The dimension of output images has to be divisible by m. Otherwise, the image is zero-padded, leading to a higher amount of operations during both transformation and matrix multiplication. (2) The amount of operations for the image and filter transformations grows quadratically with m.

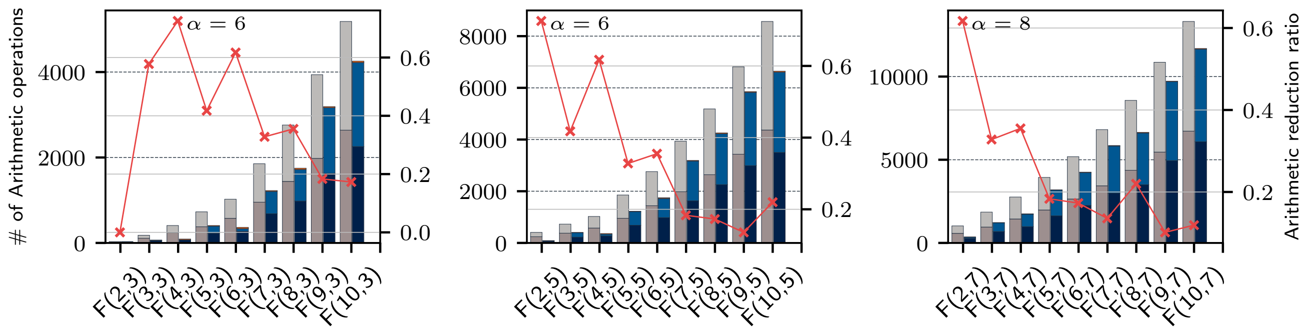
For 3×3 convolutions with small batch sizes, smaller Winograd output tile sizes m offer better runtime. However, when we increase the batch size, larger values of m between (5,7) leads to a better result. In contrast, we observe a different behavior with 5×5 convolutions. For almost every batch size, an output tile size of $m = 4$ offers lower runtime, provided that we enable our optimization. We notice that our method can speed up convolutions by up to 1.65×, particularly when $\alpha = 8$.

4.5.4 Performance portability analysis

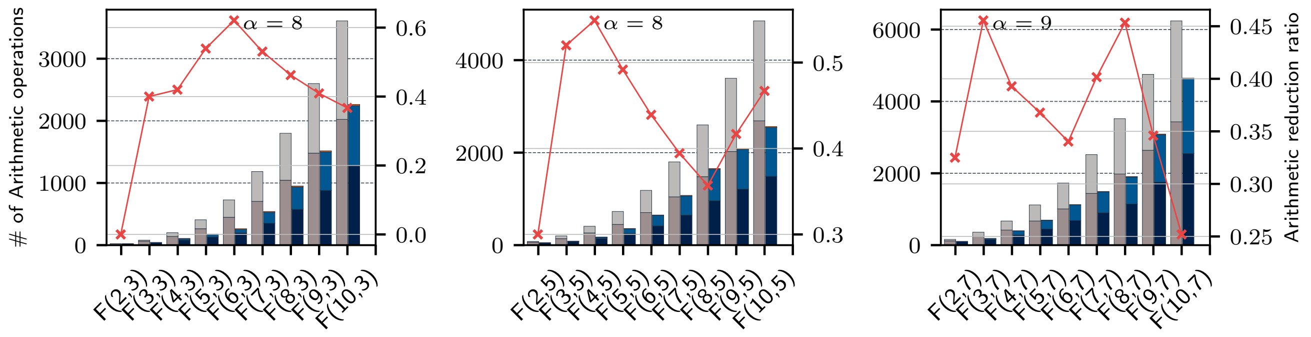
Instead of showing end-to-end runtime results for whole deep neural networks, we follow a more fine-grained approach and report per-convolution runtimes because it better highlights the impact of our optimizations. Successfully speeding up even a single convolutional layer implies shorter inference runtime for the whole network. Thus, to evaluate the performance portability of our approach, we selected a range of convolution operations and generated the corresponding GPU code for each of the target APIs. Particularly, we extracted 43 unique convolutions from real ConvNets, such as AlexNet [84], Network-in-Network [130], and the InceptionV1 [131] networks, which have (1) a batch size of five, and (2) more than $1e8$ FLOPS. To evaluate Winograd convolutions, however, we narrowed down our list to another 31 unique convolutions, as Winograd



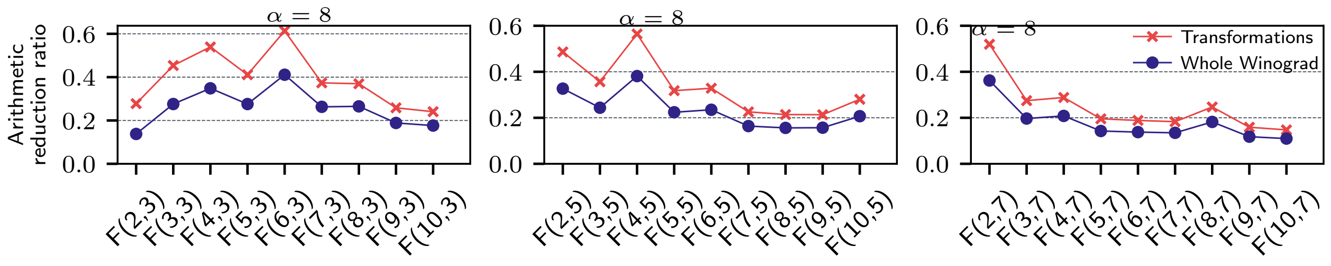
(a) Filter transformation



(b) Input transformation

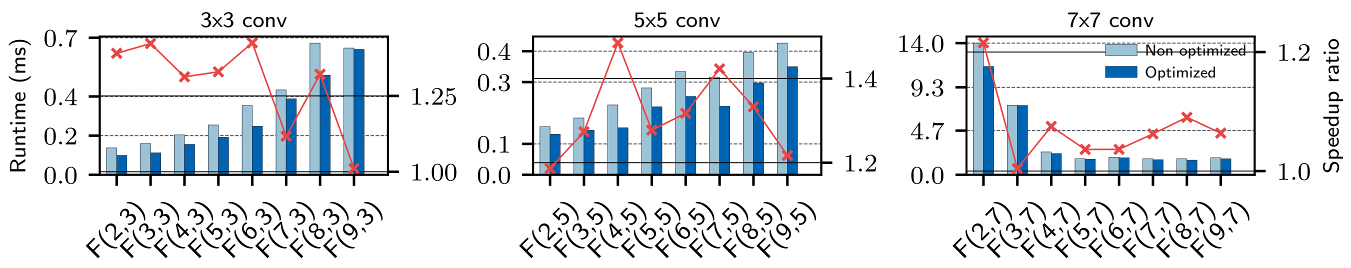


(c) Output transformation

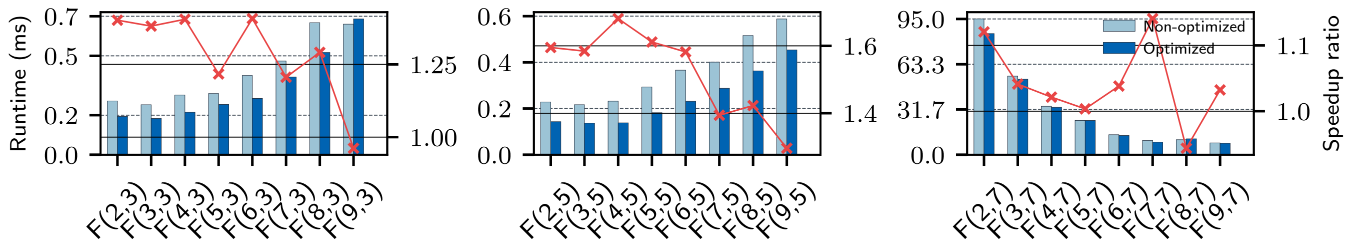


(d) Overall arithmetic reduction ratios related to transformation steps and the whole Winograd algorithm for a single tile.

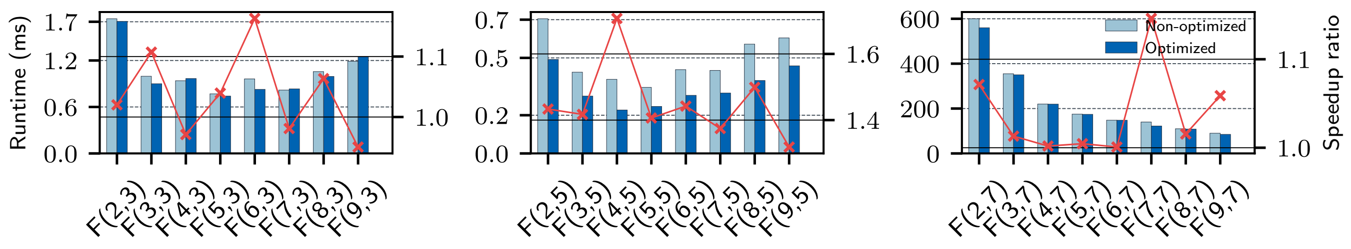
Figure 4.9: Comparing the number of arithmetic operations of each Winograd transformation, before and after optimization, where $r \in \{3, 5, 7\}, m \in [2, 10]$. Our analysis indicates that the highest arithmetic reduction can be achieved when $\alpha = 8$.



(a) $Batch\ size = 1$



(b) $Batch\ size = 5$



(c) $Batch\ size = 20$

Figure 4.10: Comparing the runtimes of optimized and non-optimized Winograd convolutions $F(m^2, r^2)$, where $r \in \{3, 5, 7\}, m \in [2, 9]$, and the batch size $B \in \{1, 5, 20\}$.

convolution operations only makes sense for certain convolutions. The rationale behind this selection is that we wanted these convolutions to model a streaming deployment scenario with high computational load but some latency tolerance. The exact specifications for both sets of convolutions can be found in Table 4.7 and Table 4.8.

For the sake of precision, we measured the execution times using GPU timers. Furthermore, to counter run-to-run variation, we executed each kernel ten times and reported the average of the runtimes we obtained. Because Vulkan GPU timers were not supported on our mobile platform, we had to use its CPU timers instead. All the average speedups reported across the convolutions are computed using the geometric mean.

We now present per-convolution-operation runtime results across three different hardware platforms to illustrate the efficiency and performance portability of our method. We sorted the operations by FLOP count, a reasonable proxy for their difficulty.

Nvidia GPU. A runtime comparison of CUDA, OpenCL, and Vulkan on our benchmark set of operations is given in Figure 4.11. All runtimes are for running each operation using the best function generated

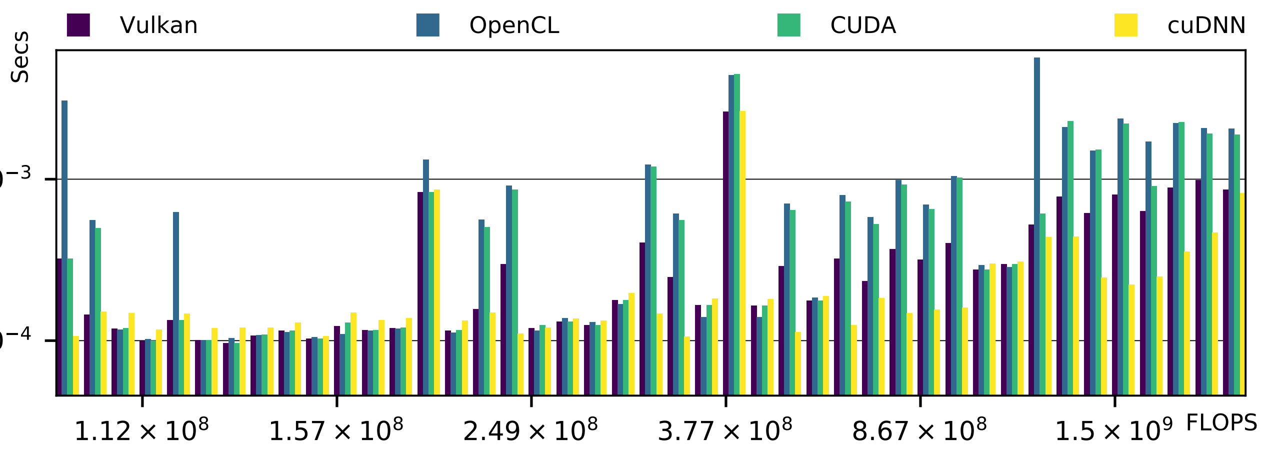


Figure 4.11: The runtime comparison of kernels generated by our method and cuDNN vendor library on Nvidia GTX 1080 Ti.

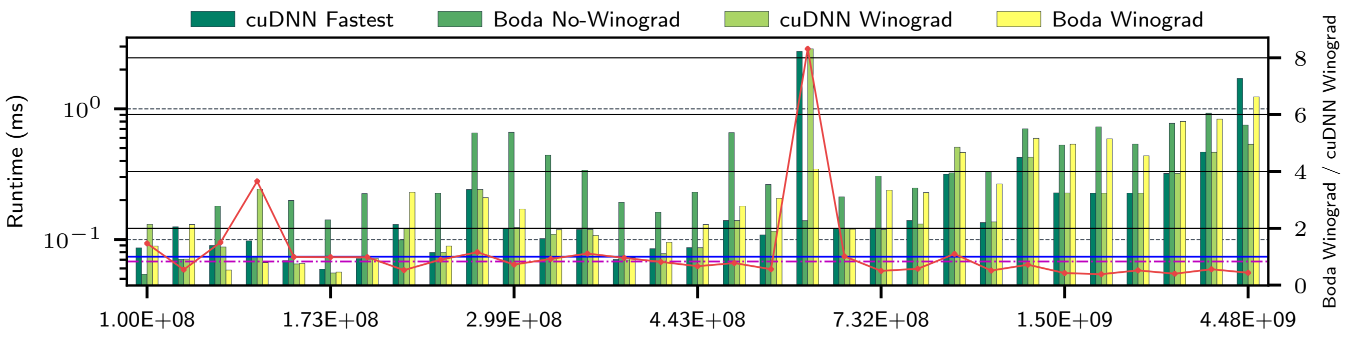


Figure 4.12: The runtime comparison of Winograd convolutions generated using our method with cuDNN on Nvidia GTX 1080 Ti.

by our method for that operation, selected by auto-tuning. The implementations are the same and only the backend API is different. We also added cuDNN runtimes as the baseline to show the performance of our method relative to the highly-tuned vendor ConvNet library. The results clearly show that our Vulkan backend often yields lower runtime in comparison to the other two, and closer to cuDNN's performance. We believe that this is owed to kernel batching and the optimizations provided by Vulkan. On average, Vulkan outperformed CUDA and OpenCL kernels by a factor of 1.54 and 1.86, respectively. Although cuDNN was able to operate 1.38$\times$ faster than Vulkan, we noticed that in some cases, Vulkan can be up to 1.46$\times$ faster than cuDNN.

Note that our results are slower especially in cases with 3$\times$3 kernel sizes, where cuDNN is using Winograd convolution. In Figure 4.12, we show that equipping our method with the Winograd convolution can decrease the performance gap and get us closer to the performance of the highly-tuned cuDNN. We also included Boda+'s runtime in the absence of the Winograd convolution to display its impact on the performance of an inference engine. We observed that cuDNN's fused Winograd implementation only supports 3×3 convolutions. Our method, on the other hand, is more versatile and can generate efficient fused Winograd kernels for larger convolutions as well. The striped horizontal line in Figure 4.12 indicates the average speedup over cuDNN's Winograd convolutions. The results reveal that not only our method can often yield

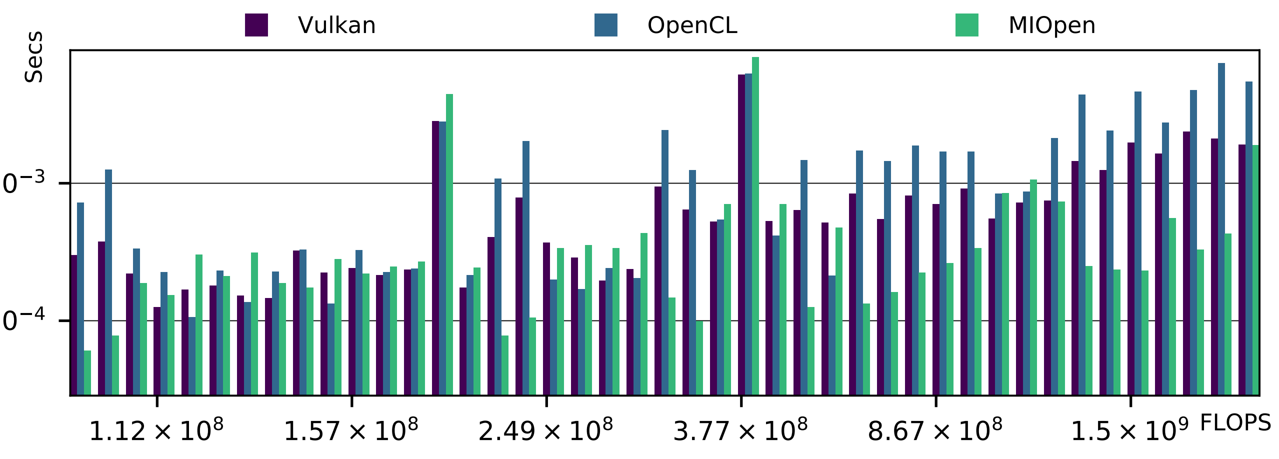


Figure 4.13: The runtime comparison of kernels generated by our method and the MIOpen vendor library on AMD Radeon RX 580.

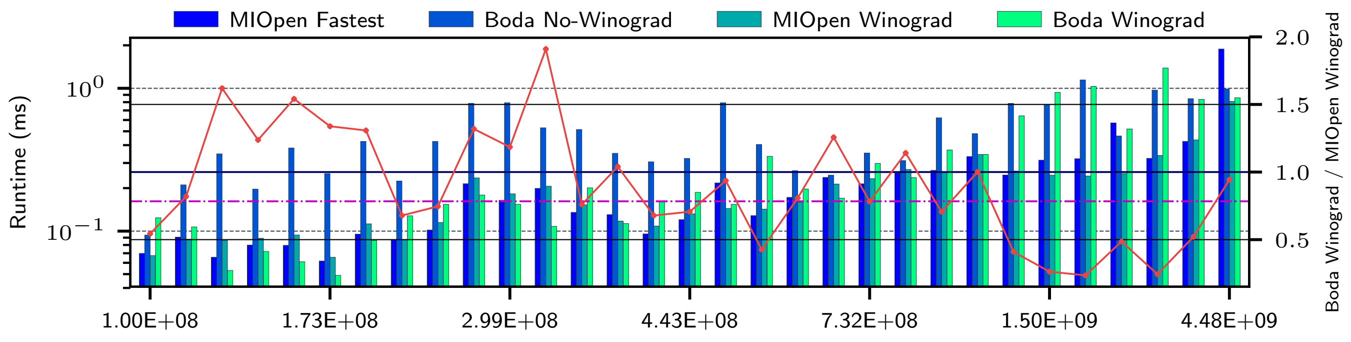


Figure 4.14: The runtime comparison of Winograd convolutions with MIOpen library on AMD Radeon RX 580.

runtimes close to cuDNN's performance, but also can perform better than cuDNN, by up to 8.1× in some cases. However, cuDNN can achieve better runtimes for larger convolutions. We believe that this can be mainly attributed to more efficient matrix-multiplication routines.

AMD GPU. Figure 4.13 compares the runtimes of our benchmark using OpenCL and Vulkan on the AMD GPU. We also show MIOpen runtimes as the baseline to show the performance of our method relative to the optimized AMD ConvNet library. Again, we notice that Vulkan outperforms OpenCL by a factor of 1.51 on average. Presumably benefiting from the highly-optimized MIOpenGEMM, MIOpen performs better than our Vulkan implementation for 25 out of 43 operations. For the 18 remaining operations, however our Vulkan version runs up to 2.28× faster than MIOpen.

We also evaluated the effect of adding Winograd convolution and compared the runtimes on the AMD paltform. Figure 4.14 demonstrates the results. The magenta striped line indicates the average speedup. Although, MIOpen still performs better than our method for larger convolutions, we speculate that this is presumably due to benefiting from the highly-optimized MIOpenGEMM library. However, in specific cases, we were able to outperform MIOpen Winograd implementation by up to a factor of 1.9.

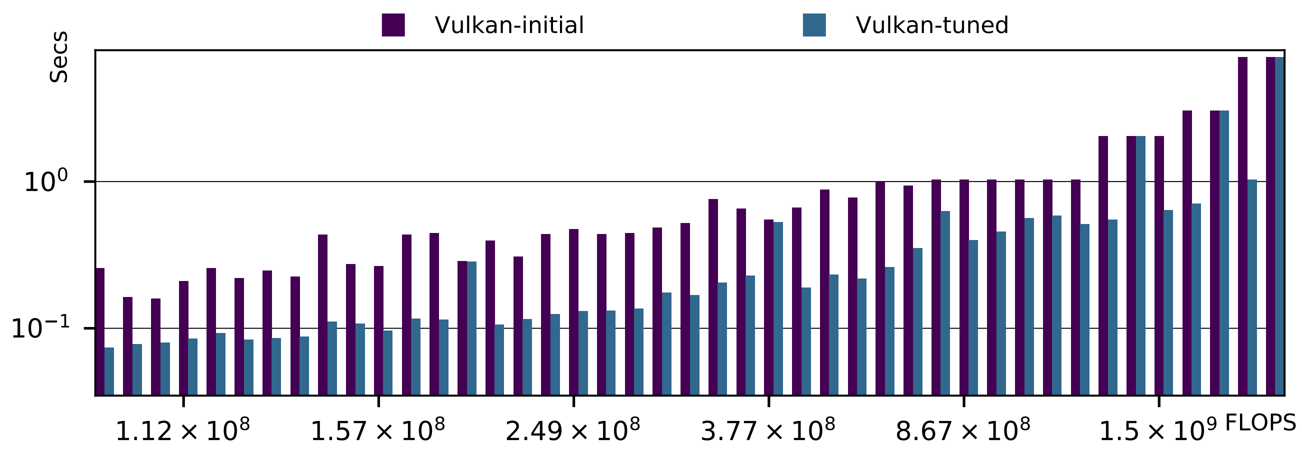


Figure 4.15: Vulkan performance with and without auto-tuning on Mali G71.

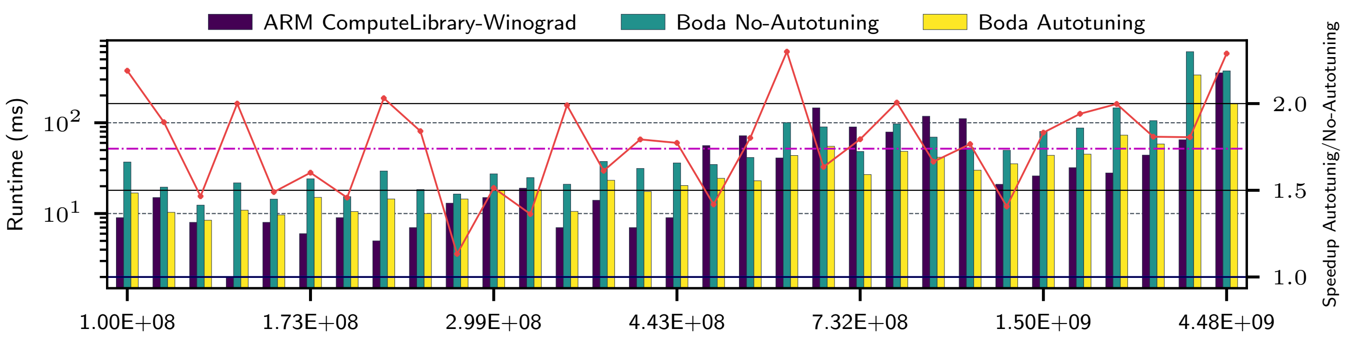


Figure 4.16: Winograd convolution performance with and without auto-tuning on Mali G71.

Hikey Mali-G71 GPU. Until now, Figures 4.11 to 4.14 illustrated that we were able to achieve competitive performance compared to the vendor libraries on two different platforms. This observation confirms that our method achieves good performance portability. To further validate the impact of auto-tuning on performance portability, we executed the code generated by our method with and without auto-tuning on our Mali G71 mobile GPU. This platform is entirely different from the previous two GPUs and usually requires an enormous amount of effort to achieve reasonable performance.

The final results after selecting the right variant and tuning parameters are shown in Figure 4.15. Note that runtimes are reported using CPU timers, because Vulkan GPU timestamps are not supported on Mali G71. Auto-tuning requires much less effort than manual tuning and improves performance significantly—on average by a factor of 3.11.

We also evaluated how Winograd convolution performs on our mobile GPU platform. Figure 4.16 illustrates the results of using the auto-tuner to select the right Winograd implementation and tuning parameters. We always used a non-fused implementation with $m = 2$, when auto-tuning is disabled. When auto-tuning is enabled, we can achieve a considerable speedup—on average, by a factor of 1.74. The red line shows the achieved speedup using auto-tuning for each convolution operation.

To compare the performance of our method with a well-known deep-learning library, we also added the Winograd convolution runtimes of the ARM compute library. The results in Figure 4.16 verify the importance of auto-tuning to achieve competitive results compared with other frameworks. Auto-tuning enabled us to find more efficient implementations and even surpass the performance of the ARM compute library for several convolution operations. We also noticed that the ARM compute library uses half-precision floating-point operations in matrix multiplications, which explains the reason for higher performance in other convolutions.

4.6 Related work

With the increasing popularity of GPUs, several authors compared CUDA and OpenCL programming models [132, 119, 133, 116, 134, 120, 135, 127], but none of them studied Vulkan. Karimi et al. [132] and Fang et al. [119] compared CUDA with OpenCL, focusing on their performance on conventional desktop GPUs. Du et al. [116] were among the first who studied OpenCL performance portability and showed that performance is not necessarily portable across different architectures. In contrast to these studies, we carried out our experiments on recent architectures and included mobile GPUs to augment the performance portability analysis. Kim et al. [120] proposed a one-to-one translation mechanism for converting CUDA to OpenCL kernels, but they do not employ any metaprogramming and code generation to achieve higher efficiency as we do. To the best of our knowledge, VComputeBench [122] is the only work which investigates Vulkan from the compute perspective and proposes it as a viable cross-platform GPGPU programming model. However, the authors concentrated more on creating a benchmark suite and did not provide a method for code translation and enhancing performance portability.

Several studies have been conducted to reduce the arithmetic complexity of convolution operations [136]. Cong et al. [137] reduced the convolution runtime by up to 47% using the Strassen algorithm. Vasilache et al. [138] further reduced the computational complexity of convolutions using an FFT-based method. However, such a method is only practical in cases where the compute-to-memory ratio is high, and the cache size is limited. Thus, FFT convolutions are mostly used for convolutions with large image/filter sizes, and when the number of input/output channels is relatively small [139].

Soon after the seminal paper on Winograd convolution [102] appeared, the algorithm was integrated in popular deep-learning libraries such as Nvidia cuDNN, AMD MIOpen, and Intel MKL. Subsequently, several researchers attempted to make Winograd convolutions more accurate for larger kernel and input sizes [125, 103]. Our experiments show that the polynomial points selected by our method can produce slightly more accurate results. Further studies on the Winograd algorithm are mostly aimed at improving its performance on various hardware platforms, such as GPUs, CPUs, edge devices, and artificial-intelligence accelerators [140, 104, 141], reducing its computational complexity by leveraging sparse computations and parameter pruning [142, 143, 144].

To the best of our knowledge, existing methods for implementing efficient Winograd convolutions are geared toward a limited set of configurations (e.g., $F(2,3)$ and $F(4,3)$) and computing devices. Each study recommends a new set of optimizations, which are often bound to a specific hardware platform. For example, Xygkis et al. [141] attempted to optimize the Winograd convolution on an Intel Movidius Myriad2 device. Such edge devices have limited power and memory capacity, and due to the high memory consumption of Winograd kernels, efficient memory management is essential. Therefore, the authors introduced optimization methods like data transfer management and data-representation optimization, which seems to be highly rewarding on Intel Myriad 2. However, they only evaluated their method for a single Winograd convolution.

In contrast to GPU devices, CPUs have access to a larger amount of memory. Such a feature enables inference frameworks to execute Winograd convolutions with higher dimensions and larger filters and output tile sizes. Three methods have been introduced to implement efficient Winograd kernels on CPUs [140, 104, 105]. Each of them suggests a different mixture of optimizations based on the target CPU. For instance, Jia et al. [104] demonstrated that their Winograd implementation can support n-dimensional convolutions.

They employed various optimization techniques, including data layout optimization, an efficient SGEMM implementation, and transformation codelets for the efficient execution of Winograd operation on CPUs. Despite their successful attempt in accelerating Winograd, their method has been tested only on CPUs. Moreover, Jia et al. [104] claim that when the Winograd's internal tile size (i.e., α) is even, only input and filter transformations have a specific pattern that allows for further reduction in computational complexity. In contrast, we demonstrated that all three Winograd transformation matrices often contain a regular pattern, even when α is odd. Furthermore, none of the above-mentioned studies proposed a solution for making Winograd convolutions performance portable. Such a feature is crucial for inference frameworks, which aim to operate on various platforms.

The amount of work published on the portable execution of ConvNets as well as the use of Vulkan in this context is very limited. To address the performance portability issue, inference engines and tensor compilers such as TVM [92], PlaidML [145], TensorFlow's XLA [93], Tensor Comprehensions [146], Glow [94], DeepMon [147], and Boda [95, 115] provide a platform to facilitate code generation and performance optimization. Important steps in that regard were the use of compiler techniques [145, 146] as well as device-specialized kernels written in shader assembly instead of high-level programming languages [123] or platform-independent intermediate representations. However, implementations that work on multiple platforms are often optimized for certain architectures or vendors. This reduces the portability and performance predictability of ConvNet execution on server-/desktop-grade GPUs and mobile GPUs alike. Furthermore, none of the works mentioned above are able to generate code for our target APIs using a single-source approach for the kernel definition. PlaidML and Tensor Comprehension do not support Vulkan at all. TVM and DeepMon are able to generate Vulkan code, but they require different input code for each programming model, demanding extra programming effort to introduce new tensor operations. Boda, on the other hand, has a compatibility layer on top of OpenCL and CUDA. Its approach is based on writing lowest-common-denominator code that is compatible between the two and uses macro definitions to abstract away syntactic differences. However, because of its larger code divergence, such an approach is definitely not extendable to include Vulkan as well.

Among the frameworks mentioned above, TVM framework [92] is the most comprehensive solution to run deep neural networks on a wide variety of hardware backends. TVM adopts the decoupled compute/schedule paradigm introduced in the Halide framework [148] and provides a domain-specific language for defining tensor operations and their optimization routines. Winograd convolution is also available in the TVM codebase. However, it is a non-fused implementation and uses predefined and hard-coded transformation matrices. We believe that integrating our symbolic analysis approach into the TVM's Winograd implementation can improve its versatility and runtime performance to a great extent.

4.7 Discussion

We attempted to show the positive impact of Boda+ on programming productivity and performance portability using the experimental results and analytical studies presented in Section 4.5. Here, we would like to discuss further qualitative insights and describe where we stand in comparison with other existing classes of methods. For a fair comparison, we chose Boda [95] as the baseline, TVM [92] as an end-to-end machine-learning compiler, and vendor libraries, such as cuDNN, as they are widely used by AI programmers. We qualitatively assessed these tools with respect to various aspects, such as productivity, efficiency, performance portability, extensibility, and ease of use. The results are given as a radar chart in Figure 4.17. From our point of view, TVM does a good job in striking a balance between the qualitative metrics. Most of its success is owed to its big established userbase, advanced architecture, and comprehensive documentation. However, our framework is more versatile, giving the user the highest degree of freedom to implement custom optimizations.

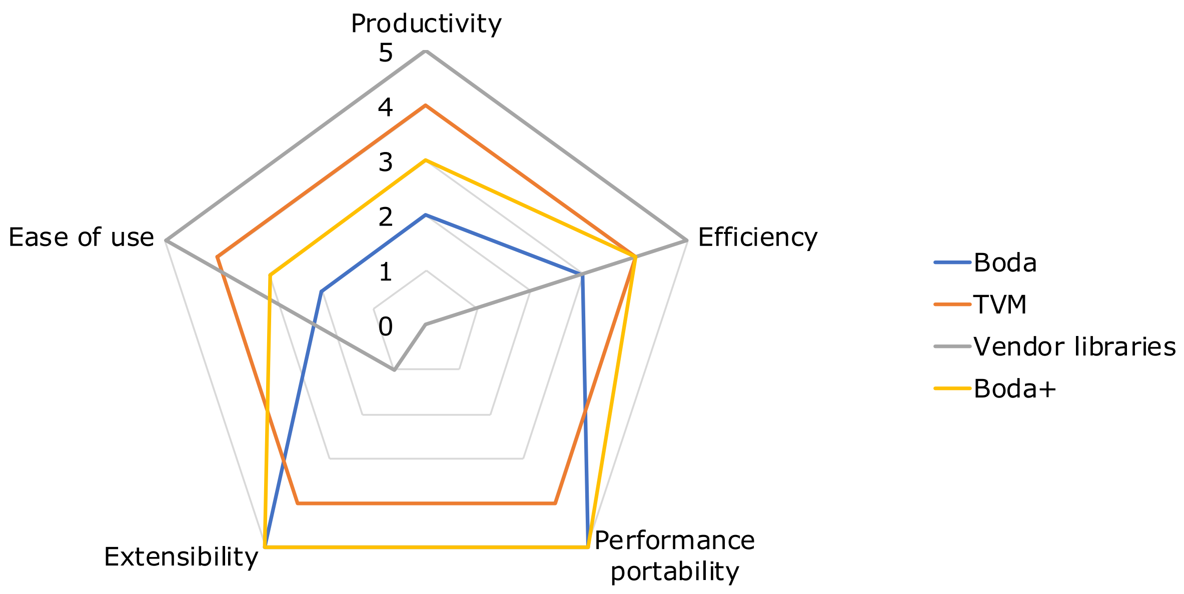


Figure 4.17: The qualitative analysis of various tensor-based libraries.

Figure 4.17 depicts two distinct spectrums, where one spectrum is the vendor libraries, and the other is Boda+. Vendor libraries are mostly suited for achieving a highly efficient implementation with minimal programming effort, where only accuracy and proof-of-concept is important. On the other hand, our framework provides performance portability and a high degree of extensibility with the cost of negligible performance loss and increased programming difficulty due to the need to support various GPU APIs. Overall, we believe that such a sacrifice is inevitable to gain the additional performance benefits. Lastly, we argue that Boda+ provides a higher degree of productivity to the users due to our MetaGPU API. Moreover, the addition of Winograd convolution made Boda+ a more efficient library than Boda.

All in all, we noticed a trade-off between the studied design aspects. In the following, we provide more detailed discussions on this trade-off and the extensibility of our framework.

4.7.1 Design trade-offs

Well-known vendor and proprietary libraries, such as cuDNN, MIOpen, and ARM Compute library, offer striking performance with minimal programming effort from the end user on their own hardware platform. Although the details of their development history are lacking, given the long development history and continuous performance improvement, we can merely assume that extensive working hours were spent on achieving the current level of performance. A programmer might readily decide to opt for one of these vendor libraries. Nonetheless, expanding the use case of the target application to various platforms using an alternative framework is often widely desired. When it comes to a custom-designed framework for performance portable deployment, the five-fold trade-off between efficiency, portability, productivity, ease of use, and extensibility emerges. Our investigations show that TVM is considerably successful in addressing all the criteria. However, when it comes to extra extensibility and portability, we might need to sacrifice one aspect for another. With Boda+, we acknowledged these issues and aimed to strike a balance between the design trade-offs, but with more attention to performance portability.

We are well aware that measuring development productivity is a challenging task, as it depends on various factors and mindsets [149]. In Section 4.5.1, we contemplated the possibility of improving this situation and reducing the development time from years to months. When analyzing the development of Boda+, we must decouple the effort spent on the framework itself from the actual time spent on implementing ConvNet operations. Given the current implementation of Boda+, these two activities are, by design, closely coupled in the source code, and it might not be accurately possible or meaningful to make a precise judgment. Nevertheless, we compared the effort involved in using our method with the effort in programming in each of the target APIs. Thus, we still believe that the programming effort estimations provided in Section 4.5.1 give at least a ballpark figure on the effectiveness of our method.

Benefiting from the novel MetaGPU abstract programming interface, metaprogramming, and auto-tuning enabled us to provide a high degree of performance portability in addition to the programmability, as we showed earlier in Section 4.5. We showed that the specialized kernels generated by Boda+ could compete with vendor libraries with reasonable programming effort when various GPU platforms are involved. It is worth mentioning that such an effort is only necessary if new tensor operations are necessary, such as the Winograd operation that we added to Boda+. Thus, efficiency is also maintained together with portability and productivity.

4.7.2 Generalization and extensibility

Although Boda+ is merely designed to accelerate ConvNets, we believe that it is generic enough and can be extended to support other application domains with reasonably low effort. Thus, here we list the classes of important features and development activities involved in Boda+, ordered from general framework support to operation-/target-specific optimizations.

Tensors processing. Tensors are first-class citizens in Boda+. Although the support for tensors and compute graph handling is more targeted towards ConvNets, they are generic enough to be used for the implementation of a wider range of operations over tensors. Thus, with minimal effort, this part of the framework can be used for other application domains as well.

Performance auto-tuning. Auto-tuning is one of the main pillars of our framework, enabling the performance portability. Boda+'s support for auto-tuning is also general, and the required development effort can be readily amortized across all hardware targets and operations. Furthermore, with the addition of more advanced auto-tuning algorithms, more complex tuning search spaces can be analyzed with less overhead.

Diverse API support. Boda+'s backends for CUDA, OpenCL, and Vulkan are generally useful for running any compute graph over tensors on the respective programming APIs. Furthermore, the MetaGPU abstraction layer immensely simplifies the transition between the target APIs. This part of the framework is also generic and the effort can be amortized across all APIs and operations.

Specialization and metaprogramming. We extensively used metaprogramming and code generation for specializing ConvNet operations on the target GPUs. In general, adding new operations or new variants of existing operations requires some effort. However, the framework is well structured to reduce the development time, and the effort can still be partially amortized across different operations.

New operations. Adding a new operation in our framework might require some effort at the framework level as well. Such an effort involves registering the operation in the framework, specifying the required optimizations, optional heuristics on when to use this operation, and fallback implementations. In the case of having a similar operation interface, this effort can be amortized across all variants.

Operation-specific optimizations. To achieve utmost performance, each operation may require some amount of tuning and optimization. We demonstrated one example of this for optimizing Winograd convolutions. Boda+ fully supports calling external libraries and scripts before generating the final code. Principally, such optimizations can be written as generic as possible to be shared among different operations, but we assume that this level contains the least amount of effort amortization.

4.8 Conclusion

In this chapter, we explored the possibility of providing performance portability with a high degree of freedom in choosing the GPU programming interface. We presented a comparative analysis of the GPU programming interfaces CUDA, OpenCL, and Vulkan. We let this comparison guide us in developing a method for generating tensor GPU kernels coded in any of those APIs from a single source that abstracts away the syntactic differences between these APIs. We implemented our approach in a state-of-the-art CNN inference framework and analyzed the programmability and performance portability of the generated kernels. Based on our experiments, our method reduces the programming effort by 98% when code portability between different APIs is demanded. Furthermore, we showed that Vulkan offers better performance compared with other APIs on our convolution benchmarks and sometimes performs better than CNN vendor libraries.

We further experimented with a hard-to-implement convolution algorithm, namely Winograd convolution. Such an algorithm is a promising method for reducing the computational complexity of convolution operations. However, if not appropriately implemented, the performance may be lower than expected. The overhead of the Winograd transformation steps can make it even inferior to direct convolution. In this chapter, we showed the flexibility of our framework to be extended with a method based on symbolic computation to create minimal yet efficient recipes that replace the straightforward matrix multiplication method within Winograd transformations. Our empirical evaluation illuminated that choosing the right output tile size m, depending on the filter size, can significantly reduce the number of arithmetic operations while offering acceptable accuracy. To the best of our knowledge, this critical observation went unnoticed so far. Furthermore, we were able to generate performance-portable Winograd convolutions with the help of template meta-programming. Our runtime analysis shows that we can not only use the same Winograd meta-code to run on a multitude of GPU platforms, including a mobile GPU, but also compete with vendor ConvNet libraries, such as cuDNN, MIOpen, and the ARM compute library.

Finally, we believe that the proposed method can be extended to support other platforms, such as CPUs, deep-learning accelerators, and dedicated inference engines (e.g., TVM [92]). To achieve higher speedups and better compatibility on new hardware, we plan to implement tunable BLAS routines tailored to Winograd multiplication steps.

Table 4.7: List of sample convolutions for testing normal convolution kernels.

KSZ	S	OC	B	input	FLOPs
5	1	32	5	28×28×16	1e+08
5	1	64	5	14×14×32	1e+08
1	1	256	5	7×7×832	1.04e+08
1	1	112	5	14×14×512	1.12e+08
1	1	128	5	14×14×512	1.28e+08
1	1	64	5	28×28×256	1.28e+08
1	1	64	5	56×56×64	1.28e+08
1	1	128	5	14×14×528	1.32e+08
1	1	144	5	14×14×512	1.44e+08
1	1	96	5	28×28×192	1.44e+08
1	1	384	5	7×7×832	1.56e+08
1	1	160	5	14×14×512	1.60e+08
1	1	160	5	14×14×528	1.65e+08
1	1	4096	5	1×1×4096	1.67e+08
1	1	192	5	14×14×480	1.80e+08
5	1	128	5	14×14×32	2e+08
3	1	320	5	7×7×160	2.25e+08
1	1	384	5	13×13×384	2.49e+08
1	1	128	5	28×28×256	2.56e+08
1	1	256	5	14×14×528	2.64e+08
1	1	96	5	54×54×96	2.68e+08
3	1	384	5	7×7×192	3.25e+08
3	1	208	5	14×14×96	3.52e+08
1	1	1000	5	6×6×1024	3.68e+08
1	1	1024	5	6×6×1024	3.77e+08
6	1	4096	5	6×6×256	3.77e+08
3	1	224	5	14×14×112	4.42e+08
1	1	256	5	27×27×256	4.77e+08
3	1	256	5	14×14×128	5.78e+08
5	1	96	5	28×28×32	6.02e+08
3	1	288	5	14×14×144	7.31e+08
3	1	128	5	28×28×96	8.67e+08
3	1	320	5	14×14×160	9.03e+08
11	4	96	5	224×224×3	1.01e+09
11	4	96	5	227×227×3	1.05e+09
7	2	64	5	224×224×3	1.18e+09
3	1	1024	5	6×6×384	1.27e+09
3	1	256	5	13×13×384	1.49e+09
3	1	384	5	13×13×256	1.49e+09
3	1	192	5	28×28×128	1.73e+09
3	1	384	5	13×13×384	2.24e+09
3	1	192	5	56×56×64	3.46e+09
5	1	256	5	27×27×96	4.47e+09

Table 4.8: List of sample convolutions for testing Winograd kernels.

KSZ	S	OC	B	input	FLOPs
5	1	32	5	28×28×16	1e+08
5	1	64	5	14×14×32	1e+08
3	1	256	1	14×14×128	1.16e+08
5	1	96	1	28×28×32	1.2e+08
3	1	288	1	14×14×144	1.46e+08
3	1	128	1	28×28×96	1.73e+08
3	1	320	1	14×14×160	1.81e+08
5	1	128	5	14×14×32	2.01e+08
3	1	320	5	7×7×160	2.26e+08
3	1	1024	1	6×6×384	2.55e+08
3	1	256	1	13×13×384	2.99e+08
3	1	384	1	13×13×256	2.99e+08
3	1	384	5	7×7×192	3.25e+08
3	1	192	1	28×28×128	3.47e+08
3	1	208	5	14×14×96	3.52e+08
3	1	224	5	14×14×112	4.43e+08
3	1	384	1	13×13×384	4.49e+08
3	1	256	5	14×14×128	5.78e+08
5	1	96	5	28×28×32	6.02e+08
3	1	192	1	56×56×64	6.94e+08
3	1	288	5	14×14×144	7.32e+08
3	1	128	5	28×28×96	8.67e+08
5	1	256	1	27×27×96	8.96e+08
3	1	320	5	14×14×160	9.03e+08
3	1	1024	5	6×6×384	1.27e+09
3	1	384	5	13×13×256	1.5e+09
3	1	256	5	13×13×384	1.5e+09
3	1	192	5	28×28×128	1.73e+09
3	1	384	5	13×13×384	2.24e+09
3	1	192	5	56×56×64	3.47e+09
5	1	256	5	27×27×96	4.48e+09

Note: KSZ, S, OC and B are the kernel size, stride, number of output channels, and batch size of each convolution operation. $input$ is sizes of input tensor, specified as $y{\times}x{\times}chan$. FLOPs is the per-operation FLOP count.

5 Performance scalability: An adaptive and scalable neural architecture search platform

Nowadays, the prevalence of employing AI models on edge devices with limited resources is of no surprise. MobileNet [150, 151], ShuffleNet [152, 153], DiCENet [154], and CondenseNet [155] are among the notable mobile-friendly ConvNet models that were designed with custom convolutional blocks to improve the overall efficiency. However, such models do not promise to yield the utmost efficiency on every hardware platform or AI task. A typical solution is to design a customized network according to the limitations and specifications of the target hardware or a given AI task. Nevertheless, the design and training of such models are not straightforward, as it is often necessary to perform a multitude of different training experiments to find the most suitable AI model. Such a variety of experiments could result from manual hyperparameter tuning, automatic neural architecture search (NAS), or iterative model compression/pruning. Therefore, we argue that an efficient and scalable training platform is inevitable for obtaining the desired model in a reasonable amount of time.

Given that state-of-the-art deep-learning (DL) models are considerably deep and require an extensive amount of time to be fully trained, designing a robust DL model is an arduous and time-consuming task. As a result, DL training has become one of the most common workloads in HPC and cloud data centers. As a prominent data-intensive and long-running workload, the training phase is primarily executed on distributed systems with access to a multitude of high-end accelerator cards (e.g., GPUs and TPUs). Given the large pool of computing nodes available in a distributed infrastructure, scalability plays an essential role in benefiting from such resources. Moreover, the efficient mapping of DL training processes to the processing nodes is a challenging task. Particularly, distributed DL training is often realized via parameter-server or all-reduce architectures. In parameter-server architecture, a set of nodes are dealing with collecting and distributing DL gradients among the workers. Thus, in case of a large model, communication bottleneck issues might arise. However, all-reduce architecture does not require any specific node for collecting gradients and all the compute nodes participate in performing reduction over the gradients computed on their neighboring nodes.

Shared HPC clusters are often equipped with a scheduler (e.g., SLURM) that mediates and maps the submitted jobs to the available resources. A common disadvantage is that such schedulers often require the user to define the required resources manually. As a result, the scheduler procures the resources only if they are available. Naturally, improper resource allocation leads to inefficient training performance and also underutilization of the cluster. Furthermore, DL jobs are often checkpointed periodically, providing the opportunity to be adjusted given their progress and performance. For instance, we can scale up or down the resources allocated to a DL job during execution to increase the utilization and make resources accessible again for other concurrent jobs. Nonetheless, existing DL-agnostic schedulers do not consider these properties. Alternatively, a DL-aware job scheduler can significantly enhance the scheduling process and make a more efficient and scalable training platform.

This chapter proposes a specialized training platform dedicated to DL model training and compression. Existing similar platforms such as Microsoft NNI [156], Gandiva [157], Pollux [158], Optimus [159], and DL^2 [160] primarily focus on ordinary DL training and neglect their unique characteristic. Since the models with fewer parameters are more scalable in data-parallel training [161], we believe that scalable DL training is achieved via a combination of network pruning and DL-aware job scheduling. Our proposed platform streamlines the resource provisioning on HPC/cloud clusters via micro-services, accelerates DL-training/NAS jobs, optimally scales up/down the training phase, and adapts the trained model to save memory and computation.

5.1 Motivation and background

This section aims to describe the motivations and necessities behind our approach for scaling the performance of deep-learning applications. We argue that such applications have specific characteristics that DL-specific schedulers can exploit to expedite their training time and also increase resource utilization. Furthermore, scalability is not only provided by external scheduling platforms, but also from adjusting the deep-learning model itself. In the following, we explain the challenges and ideas that motivated the design of our solution.

5.1.1 Quantitative analysis of distributed DL training

The quality of a training algorithm can be analyzed quantitatively via training throughput and statistical efficiency. In the following, we briefly describe these efficiency metrics.

Training throughput. The throughput of deep-learning training is defined as the number of training samples processed per a unit of time. Each training iteration is composed of the time spent on computing gradients T_{grad} and the synchronization time T_{sync}. The latter is governed by the size of gradients and network performance, which itself depends on the distance of nodes from each other. For instance, the GPUs that are co-located on the same node will benefit from lower synchronization time.

Given a distributed data-parallel training job, the throughput depends on various critical factors [158], such as resource allocation and node interconnections, gradient synchronization method (e.g., synchronous or asynchronous), and the selected batch size. The first two factors are almost fixed and unmodifiable. However, the batch size is easily adjustable. Typically, a larger batch size causes a higher throughput and is beneficial to offset the T_{sync} time.

Statistical efficiency. The efficiency of a training job can be described as the amount of learning progress made per a unit of training data. This metric has a tight correlation with batch size and learning rate parameters, where a larger batch size typically decreases the statistical efficiency [158], as it adversely affects the training stability and generalization performance. Estimating this metric is often based on the gradient noise scale (GNS), which analyzes the noise-to-signal ratio of the stochastic gradient. Simply put, a larger GNS translates to the possibility of increasing batch size and learning rate to higher values without noticing a significant loss in statistical efficiency. However, this metric is not constant throughout the training phase and tends to increase by up to $10\times$ near convergence [162]. Therefore, increasing the batch size in the later stages of the training can enhance the statistical efficiency and expedite the training process by achieving higher throughput.

5.1.2 The necessity of DL-aware scheduling

Shared computing clusters often provide resources through a job scheduler that manages the available compute nodes and resources. The main goal of a scheduler is to provide users with fair access given the user's requirements. For this purpose, a wide variety of general-purpose schedulers have been developed. SLURM [163], Mesos [164], YARN [165], and Borg [166] are among the well-known generic schedulers. A typical workflow for submitting a training job to such schedulers is to define a job script containing the path to the training code and additional scheduler-specific parameters, such as the number of required resources (CPU cores, memory, and GPUs) and time duration. After submitting the job, the scheduler attempts to allocate the required resources based on its scheduling algorithm and runs the user's code. The number of allocated resources remains constant throughout the execution, and the job might even terminate if the duration exceeds beyond the defined runtime, requiring the user to resubmit the job.

We argue that using workload-agnostic schedulers is detrimental to the process of training deep-learning models, as it suffers from the following limitations and challenges.

Selecting the right number of worker nodes. Traditional DL-agnostic schedulers demand the user to define the number of nodes involved in the training. All-reduce distributed architectures only need the number of workers, whereas, within a parameter-server architecture, the number of workers and parameter servers (PS) have to be carefully selected. In this regard, former studies observed that a disproportionate quantity of PS and workers leads to communication bottlenecks. Therefore, a proper resource allocation scheme is one of the main challenges in scaling the training performance of a given deep-learning model. Without any adjustment in the training hyper-parameters, increasing the number of resources will lead to a decreasing speedup [160], preventing a linear speedup. The underlying reason is the increasing amount of incurred communications. Therefore, the best course of action is to let the scheduler select the right number of resources for a given job. The scheduler can observe the training progress and scale the job to the maximum number of possible worker nodes, provided that no other job is competing for resources. DL-agnostic schedulers fail to assist the user in selecting the right number of computing resources and follow a fixed scheduling strategy.

Static resource allocation. Existing DL-agnostic schedulers follow a fixed scheduling strategy, in which, once all the resources are assigned, other jobs should wait in the queue. Furthermore, if a training job finishes and no other job is waiting in the queue, other running jobs cannot benefit from the recently freed resources, making the cluster underutilized. Above all, the convergence and progress speed of DL training jobs varies over time and different deep models [167], as they make moderately large improvements at the beginning and begin to slow down at the end of the training, demanding additional computing power to expedite the convergence speed. Hence, depending on the progress speed of the jobs, it is necessary to scale up/down their resources to adapt the computational requirements of all jobs in the cluster.

All the situations mentioned earlier necessitate a dynamic resource allocation system to maximize resource utilization and expedite the running jobs. Currently, deep-learning training algorithms are checkpointable and highly scalable. Thus, it is relatively straightforward to save a checkpoint, seize the training, adjust resource allocations, and resume the job with new computing resources. Despite the incurring scaling overhead, the benefits outweigh the cost, as we can achieve higher resources utilization, faster DL training, and more fair job scheduling. Making the scheduler fully in charge of setting and adjusting the number of worker nodes will not only cut back on the operating costs, but also accelerate the design and validation of AI applications. DL^2 [160] and Optimus [159] mostly inspired our solution on designing cluster-level scheduling by following their methodology on allocating compute nodes.

Tuning training hyper-parameters. Although a DL-aware and dynamic resource allocation can significantly enhance training jobs, neglecting the inter-dependent training hyper-parameters can prevent us from achieving maximum efficiency [158]. Particularly, batch size and learning rate of a training job largely depend on the model and the allocated resources, as they influence the training progress and computation volume.

Conventionally, the user is responsible for fine-tuning such training hyper-parameters and the number of resources. Typically, such a task is done by trial and error or based on former experience. However, former studies [158] have shown that the optimal value for such parameters is not constant throughout the training and might need to be adjusted. Hence, a fully DL-aware scheduler should consider both the optimal resource allocation and hyper-parameter tuning. Notable works in this area are Pollux [158] and Kungfu [168]. Particularly, we used the goodput metric introduced in Pollux to find the best hyper-parameters.

5.1.3 Dealing with large deep-learning models

Given the ever-increasing complexity of AI tasks, deep-learning models tend to have a deeper and wider structure to provide more accurate results. For instance, the ResNet family has a wide range of architectures, from 18 to 110 layers. Opting for deeper models dramatically increases the number of parameters, incurring a higher volume of computation and memory. Evidently, low-budget edge devices are not able to run such deep models. Furthermore, former studies [161] have shown that smaller models with fewer parameters scale better in data-parallel training jobs. A typical solution is to shrink and adjust large models to fit the capabilities of the target hardware and also expedite the training time. A well-known method is network pruning that removes unnecessary feature-map channels to reduce the computational complexity of a model. Since network pruning often requires a multitude of additional training processes to fine-tune the model and increase the accuracy, this process can also benefit from a DL-aware training platform.

5.1.4 A holistic approach for network design and training

As we have learned so far, the scalability of deep-learning models is not only tied to the proper scheduling of the training phase. Additional steps such as network pruning are required to accelerate the training phase and make the model smaller to fit the limitations of the target hardware. Thus, we argue that a holistic and comprehensive solution is necessary to achieve such a goal. As depicted in Figure 5.1, a potential solution is a synergy of network pruning together with a DL-aware scheduler that can efficiently train a model suitable for execution on a device with constrained FLOPs. Ideally the platform is built on top of well-developed distributed systems for training, such as Horovod [169]. Furthermore, using containers (e.g., Docker or Podman) and Kubernetes streamlines the process of scaling up and down the resources in a fraction of seconds.

Moreover, we believe that neural architecture search can greatly benefit from a robust and efficient training platform, as NAS jobs are extremely time-/resource-consuming. NAS jobs are slightly different from normal DL training jobs, demanding special mechanisms to achieve scalable training experience. For instance, one-shot NAS methods often have two different gradient optimizers—one for the actual weights and another for architecture weights. To the best of our knowledge, existing DL-aware schedulers, such as Pollux, do not support NAS methods. In this work, we aim to address this shortcoming and provide scalable training for NAS jobs.

5.2 Scalability via DL-aware scheduling

We introduce an elastic multi-tenant DL-aware scheduling platform based on the all-reduce architecture to deal with the increasing cost and complexity of DL training and neural architecture design. The multi-tenancy feature of our scheduler enables the efficient training of several deep-learning jobs concurrently. Furthermore, our proposed scheduler is equipped with a progress monitoring component for retrieving status updates, a cluster of docker containers managed by Kubernetes responsible for performing the actual training work, and a command-line client through which a user can interact with the system.

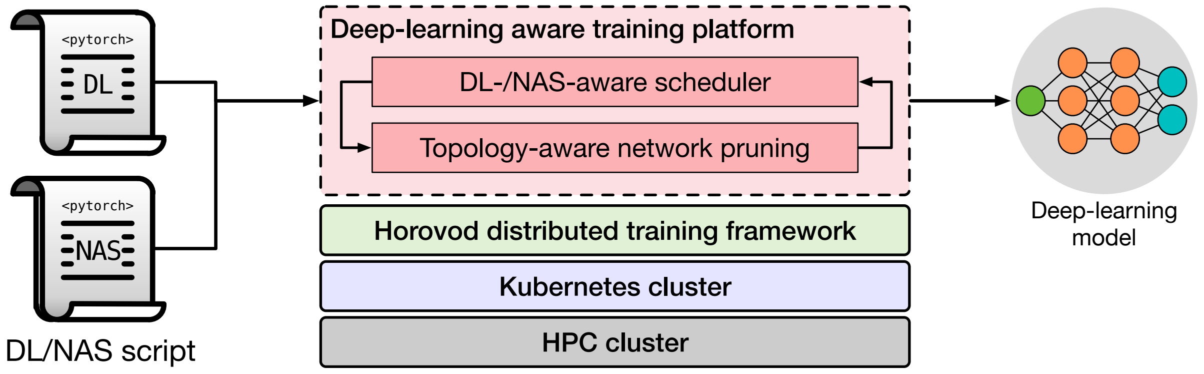


Figure 5.1: Adaptive DL-aware deep-learning training platform.

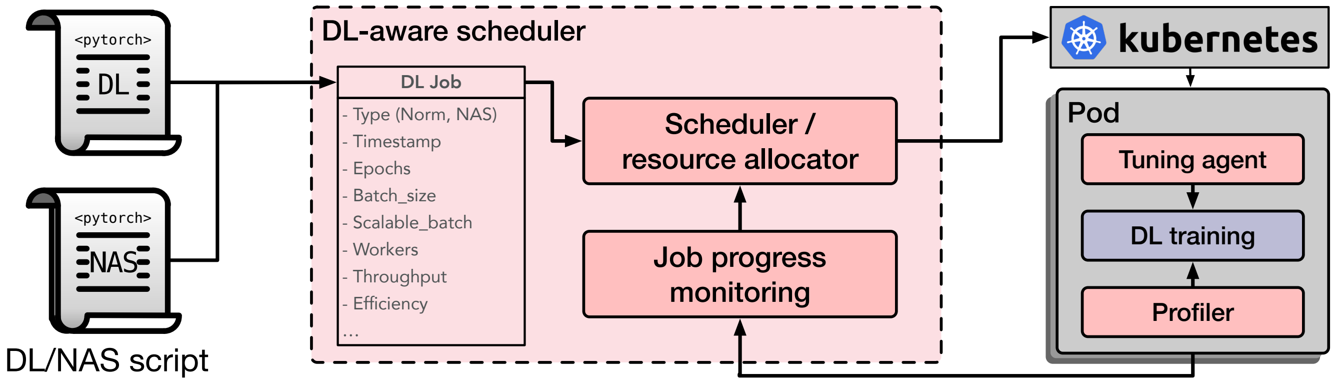


Figure 5.2: An overview of our deep-learning training scheduler.

Our scheduler operates in two separate but interdependent levels to provide scalable and elastic job scheduling, namely cluster-level and job-level scheduling. The former deals with efficient resource allocation given the available nodes on the cluster, while the latter attempts to adjust the training hyper-parameters of the individual training jobs to enhance their throughput and efficiency. Figure 5.2 illustrates a high-level overview of our scheduler's architecture. The workflow starts with the user submitting a DL or NAS job to the scheduler. The scheduler extracts various principal information, such as job type, epochs, and batch size. Using the extracted information, the cluster-level scheduler allocates resources based on the job's specification. Then, during the training time, both the job and cluster-level schedulers constantly monitor the progress and adjust the job's parameters along with the allocated resources. In the following, we will explain the cluster-level and job-level scheduling.

5.2.1 Job-level scheduling

As depicted in Figure 5.2, a profiler is running along with the training script inside the docker container on the worker node. It periodically checks the training job's status and measures the gradient noise scale and throughput. Inspired by Pollux [158], we compute a metric called goodput at each iteration, which is the product of training throughput and statistical efficiency, designed to strike a balance between the two

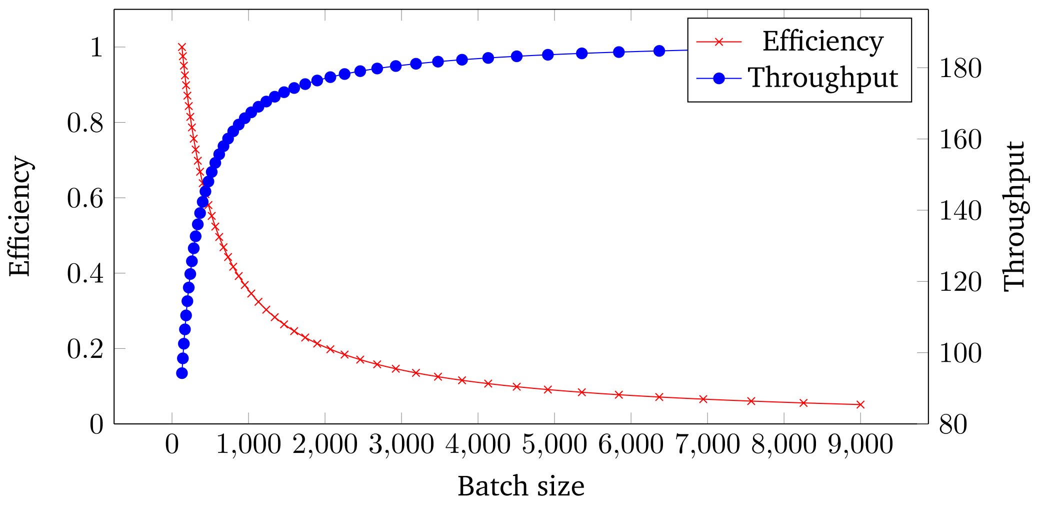


Figure 5.3: The tradeoff between statistical efficiency and throughput given different batch sizes.

metrics. Figure 5.3 illustrates the statistical efficiency and throughput given different batch sizes. As it is shown, increasing the batch size improves the throughput but with the cost of decreasing the efficiency. The product of these two metrics in the goodput calculation guarantees to find the optimal batch size.

Statistical efficiency $E_t \in (0, 1]$ is modeled using pre-conditioned gradient noise scale φ_t and is expressed as:

$$E_t(M) = \frac{\varphi_t + M_0}{\varphi_t + M}, \tag{5.1}$$

$$\varphi_t = \frac{tr(P\Sigma P^T)}{|Pg|^2}, \tag{5.2}$$

where M_0 and M are the initial and new batch sizes, respectively. g is the gradient, P is the preconditioning matrix of the SGD algorithm, and Σ is the covariance matrix of per-example stochastic gradients. Intuitively, the statistical efficiency measures the contribution of each training sample to the overall training progress. As NAS jobs have two different gradient optimizers, one for network and another for architecture weights, we compute the statistical efficiency for both of them and compute the average.

The training throughput is computed as $T = m/T_{iter}$, where T_{iter} is the time spent per iteration. It is evident that larger batch size increases the throughput, but it affects the overall accuracy. Thus, the learning rate has to be adjusted. We use AdaScale [170] to adjust the learning rate, but any other scaling rule can also be used. Iteration time also plays a vital role in the overall throughput. It involves the time required for computing the gradients plus the synchronization time. Thus, keeping the training processes close together on nearby nodes is essential to avoid expensive communication costs.

5.2.2 Cluster-level scheduling

For proper resource allocation and scheduling, we opted for a reinforcement-learning method, as it can adapt to the dynamic behavior of DL training jobs following a black-box end-to-end model. Additionally, former studies demonstrated the effectiveness of such a setup in comparison with expert heuristics and explicit modeling [160]. Thus, similar to DL^2 [160], we designed our reinforcement-learning method to take system state as the input and produce resource allocation policy for each job.

Environment states. We defined a matrix $s = (x, d, e, w, t, s)$ as the input state for the RL agent, where:

- x is a matrix, where each row is a one-hot vector for describing the type of the job. We categorize the jobs based on the model that they train, as similar jobs might benefit from the same scheduling strategy.
- d is a vector that contains the number of time slots spent for running each job.
- e is a vector that represents the number of remaining epochs for each running job.
- w is a vector that stores the number of workers allocated to each job.
- t and s are also vectors representing the throughput and statistical efficiency values for each job, respectively.

Action space. Unlike DL^2 that operates in a parameter-server architecture, our scheduler works in an all-reduce architecture. Thus, we only need to predict the right number of workers. To achieve this, we configured the actions made by the RL agent to produce discrete values for the number of workers that should be assigned to each job.

Reward function. The overall goal of scheduling is to reduce the total completion time. However, since an RL agent cannot wait until the job completion time is computed, we used the goodput predictive metric as the reward function. The rationale behind selecting such a metric is that an allocation policy that increases the throughput and efficiency of a job can potentially decrease the overall job completion time, as higher throughput correlates with processing more training samples.

5.3 Scalability via network pruning

As mentioned earlier, training small models with fewer parameters scales better. Thus, we equipped our training platform with a model compression technique that prunes redundant channels. The core of the model compression technique is to identify the appropriate compression policy for each layer as each layer has a different redundancy level. Conventionally, network pruning was done using handcrafted policies, often failing to deliver an optimal compression ratio with minimal accuracy loss. Our proposed method is, however, a topology-aware network pruning called AGMC (auto graph encoder-decoder model compression) that finds optimal pruning policies automatically.

To prune a given DNN, we first modeled the DNN as a computational graph and introduced a GCN-based graph encoder to learn the DNN's representation g. Then, the decoder decodes g into layer embeddings $s_i \in S, i = 1, 2, .., T$, where T is the number of hidden layers. Since we aim to compress the DNN by predicting the pruning ratio for each hidden layer, the RL agent takes the layer state S as the environment state to search for the hidden layer's pruning policy $a_i \in A, i = 1, 2, .., T$. The pruned DNN's performance is then used as a reward for the current actions A taken by the RL agent. Figure 5.4 depicts an overview of our method. In the following, we will explain the details of the simplified computational graph, graph encoder-decoder, and RL agent within our approach.

5.3.1 Computational graph coarsening

Computational graphs with their rich topological information are the conventional representation for defining deep neural networks. Typically, they are composed of thousands if not billions of primitive operations

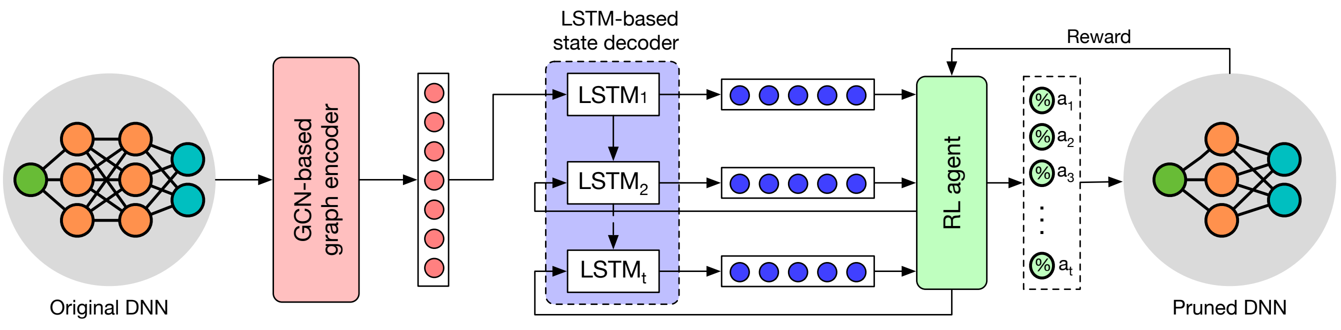


Figure 5.4: The workflow of auto graph encoder-decoder model compression (AGMC).

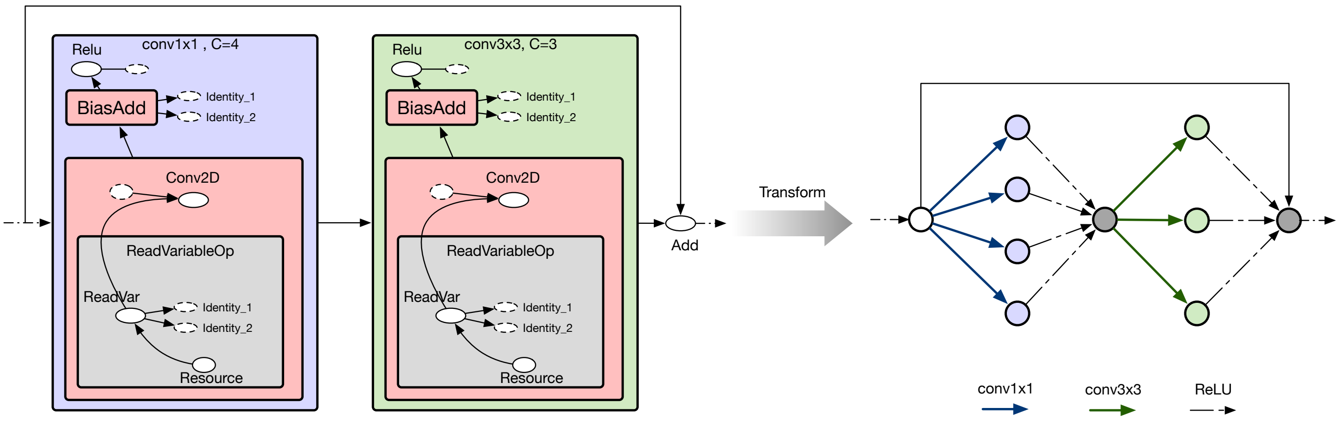


Figure 5.5: A sample computational graph coarsening applied on a ResNet block.

(e.g., add, minus, and multiply) [85], where edges are operations and nodes are intermediate results (i.e., feature maps). Thus, they are often bloated, hard to interpret, and infeasible to be directly used for any further analysis. Nonetheless, thanks to the frequently used high-level machine-learning operations $\mathcal{O}=\{n \times n$ conv, ReLU, BatchNorm, (Max/Average) Pooling, Padding, Splitting$\}$ or custom blocks (i.e., motifs) in state-of-the-art networks, we can simplify such graphs by replacing primitive operations such as *add*, *multiply*, and *minus* with high-level operations. Such a simplification can significantly reduce the complexity while preserving important structural information. As depicted in Figure 5.5, we transform a given computational graph represented in popular DL frameworks to a simplified version. Formally, we construct a graph $G=(V,E,\mathcal{O})$, where V is the feature maps, E is the operations, and $\mathcal{O}$ is the high-level operations. Each directed edge with an edge type is associated with an operation in $\mathcal{O}$.

5.3.2 Automated graph encoder-decoder

To enable further analysis of the graph inputs, we need to extract meaningful information, preferably in a lower dimension. Embedding vectors are particularly designed for this purpose, as they are a relatively low-dimensional learned space into which we can translate high-dimensional vectors. Furthermore, such a representation facilitates the analysis with machine-learning models. An embedding can be learned and then reused across different models.

To extract and learn the embeddings from the hidden layers in computational graphs, we introduce a two phase encoder-decoder. Graph convolutional networks (GCN) and their variants [171, 172] have been successfully applied to learn the topology information from graphs. For instance, they have been successfully applied to node classification, link prediction, and graph classification. The encoder part analyzes the computational graph using GCN and generates an embedding for the entire DNN's structure $g \in \mathbb{R}^{1\times d}$, where d is the embedding vector size. To obtain the embedding vector for each hidden layer, we designed a decoder based on LSTM [173] networks. We denote layer embeddings as $S \in \mathbb{R}^{T\times d}$, where T is the number of hidden layers, and each contains an embedding of size d. In the following, we provide a detailed explanation of both encoder and decoder parts.

GCN-based graph encoder

As already explained in Section 3.7, we can employ GCNs to embed graphs by gathering node features from neighboring nodes. Hence, we apply the graph encoder on the target computational graph G to obtain node-embedding matrix $\mathbf{H} = \text{GCN}_{encoder}(G) \in \mathbb{R}^{N\times d}$. The graph encoder is a two-layer GCN with a hidden feature size of 50 units and a DNN embedding size of 11 units. The message passing function is also formulated as Equation 5.3.

$$h_i^{l+1} = \sum_{j\in N_i} \frac{1}{c_i} W^l h_j^l, \tag{5.3}$$

where h_i^l refers to the hidden state of node i in the l^{th} GCN convolution, c_i is a constant coefficient, N_i is the set of node i neighbors, and W^l is the weight matrix that needs to be learned during training.

Similar to standard GCNs, our GCN also attempts to learn node embeddings, yet we need to learn the embedding for the entire graph structure. A commonly used mechanism to compute graph-level embedding is to utilize the graph mean pool as formulated in Equation 5.4. Principally, it computes an average of the node embeddings and provides a graph embedding **g**.

$$\mathbf{g} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{h}_i, \tag{5.4}$$

where $\mathbf{H} = \mathbf{h}_i, i = 1, 2, ..., N$ is the node-embedding matrix, $\mathbf{h}_i$ is the embedding of i^{th} node , N is the total number of nodes in the graph, and d is the embedding size.

LSTM-based decoder

The decoder component is designed to learn the environment states of the target DNN hidden layers required for the RL agent. Since the state vectors in the reinforcement-learning (RL) environment are determined by the previous state and the action (the pruning ratio), the decoder takes the previous layer's state vectors and RL agent's action as input as well. To deal with such an input space, we used long short-term memory, as it uses feedback connections to process the sequence of data, instead of a single data point. Thus, we define our decoder as in the following formulas:

$$s_1 = \text{LSTM}_{decoder}\left(\mathbf{g}\right) \tag{5.5}$$

$$s_t = \text{LSTM}_{decoder}\left(s_{t-1}, a_{t-1}\right) \tag{5.6}$$

For the $t-$th hidden layer, we use the feature s_{t-1} of the previous hidden layer and the compression policy a_{t-1} (the action selected by the RL agent) to calculate the environment states.

5.3.3 Network pruning using reinforcement learning

We leveraged reinforcement learning to find the optimal pruning ratios efficiently. We briefly explained the concept of RL in Section 3.6. In the following, we describe the details of the RL setup for network pruning.

Environment states. In contrast to existing RL-based model compression methods that use fixed hand-crafted layer embeddings as environment states, we use DNN layer embeddings $S \in \mathbb{R}^{T \times d}$ generated by the graph encoder-decoder as environment states.

Action space. The actions made by the RL agent are pruning ratios within a continuous space. Specifically, the RL agent takes the layer embeddings $S \in \mathbb{R}^{T \times d}$ as environment states and predicts corresponding pruning ratios $a_i \in A, i = 1, 2, .., T$, where $a_i \in [0, 1)$.

Reward function. Since the RL agent is responsible for pruning the network, the actions made by the RL agent should be evaluated and rewarded with positive feedback or penalized with a negative value. Thus, we formulated the reward function to evaluate the performance of the pruned network as $R_{err} = -Error$, where $Error$ is the compressed DNN's top-1 error on the validation set. However, one drawback of such a reward function is that it offers little or no incentive for model size and FLOPs reduction. Without including any constraint (e.g., FLOPs or #parameters), the RL agent tends to look for a tiny compression ratio. To solve this issue, we devised an action rescaling mechanism. As described in Algorithm 1, we compute the size that we still need to reduce according to the original scale. Lines 1-2 compute the total model size (e.g., FLOPs and #parameters) W_{all} and reduced size $W_{reduced}$. If the reduced size is less than the desired model size reduction d, the algorithm will rescale the pruning ratios to compensate for the difference $d - W_{reduced}$. Lines 4-7 relate to the rescaling process, and the for-loop in lines 5-7 adjusts the pruning ratio for each layer according to the difference to the desired model size reduction. Finally, in line 7, we truncate the pruning ratio with the upper bound a_{max}.

Algorithm 1: RL action rescaling algorithm to achieve the desired model size reduction.

```
Input: The actions $a = \{a_0, ..., a_T\}$, the upper bound of actions $a_{max}$, the model size
       (#FLOPs/#Parameters etc.) of each hidden layer $W = \{W_0, ..., W_T\}$, and the desired model
       size reduction $d$
Output: The actions $a'$ after re-scaling
$W_{all} = \sum_t W_t$
$W_{reduced} = \sum_t W_t a_t$
if $W_{reduced} < d$ then
    $d_{rest} = d - W_{reduced}$
    for $i = 1, 2, ..., T$ do
        $a_i += (d_{rest} * (a_i / \sum_t a_t))/W_i$
        $a'_i = min(a_{max}, a_i)$
return $a'$
```

RL agent policy. There are various RL search policies, such as proximal policy optimization (PPO) [174] and deep deterministic policy gradient (DDPG) [175]. However, to make a fair comparison with our baseline method AMC [176], we chose DDPG as our RL policy to exclude the effect of the RL policy on the evaluation results. We believe that this isolation design choice can help show the benefit of our learning-based network pruning compared to handcrafted methods.

We configured the DDPG agent to receive the environment states $s_i \in S, i = 1, 2, .., T$ produced by our decoder and produce actions (i.e., pruning ratios) $a_i \in A, i = 1, 2, .., T$ by using a multi-layer perceptron (MLP). The actor-network μ and the critic network Q have two hidden layers, each with 300 units. The μ's output layer applies the sigmoid function to bound the actions within $(0, 1)$. We use $\tau = 0.01$ for the soft target updates. In the first 25 episodes, our agent searches with random action. Then, it continues searching for 300 episodes with exponential decayed noise.

5.4 Experimental results

We individually evaluate our training platform according to the two components that build our system. First, we analyzed our DL-aware scheduler for normal training and neural architecture search. Then, we conducted a detailed study on the efficiency of our network pruning method.

5.4.1 Evaluating the DL-aware job scheduler

We evaluated our scheduler with respect to the job completion time using a simulation platform to schedule 40 jobs on 16 nodes. The jobs are submitted based on a job trace, which instructs how and when the jobs arrive. The jobs resemble the training progress for various models such as Resnet-50, VGG-16, and ResNeXt. Figure 5.6 shows the average job completion time (i.e., time from job submission to its completion) of the Dominant Resource Fairness (DRF) scheduler, DL^2, and our elastic DL-aware scheduler. We chose DRF as it is a generic, DL-agnostic scheduler often used in typical schedulers, in which jobs are prioritized based on their dominant resource share (e.g., CPU, GPU). As the results show, we achieved 51% reduction in the average job completion time compared with DL-agnostic schedulers. Clearly, exploiting DL training characteristics can yield such a considerable improvement. Moreover, we outperformed the DL-aware scheduler DL^2 and alleviated the training time by up to 36%. Such a comparison shows that the combination of adaptive hyperparameter tuning and DL-aware job scheduling has a significant effect on accelerating DL training performance.

Next, we studied the effect of adaptive hyperparameter tuning on the DL training process. In contrast to Pollux that only studied normal DL training jobs, we also included NAS jobs and attempted to accelerate them using this technique. We ran our experiments on Nvidia's A100 GPUs available on the Lichtenberg II cluster at TU Darmstadt. Figure 5.7 depicts the average job completion times in log-scale for various DL and NAS jobs, each configured with different settings. We used MobileNet-v2 and DARTS trained on CIFAR-10 as our normal DL training and NAS workload, respectively. Regarding the training settings, we primarily altered the number of GPU workers $w \in \{1, 2, 4\}$ while keeping the batch size constant for four cases and let the remaining two cases to be adaptively adjusted. The remaining training parameters were left untouched, except for the learning rate, which is also adjusted to maintain a high level of accuracy.

As the results show, adding another GPU worker without adapting the batch size slows down the overall training process since per-GPU batch size becomes smaller (i.e., $B = 32$) and more gradient synchronizations are necessary. Manually increasing the overall batch size to 128 implies that each GPU uses the initial batch size (i.e., $B = 64$) and alleviates the issue by offering a noticeable speedup of 1.14×. However, with the same number of GPU workers, we achieved the speedup of 2× using adaptive batch size tuning. We noticed a similar pattern within NAS training, however, with less speedup. Nevertheless, the speedup compared to manual batch size selection is clearly noticeable.

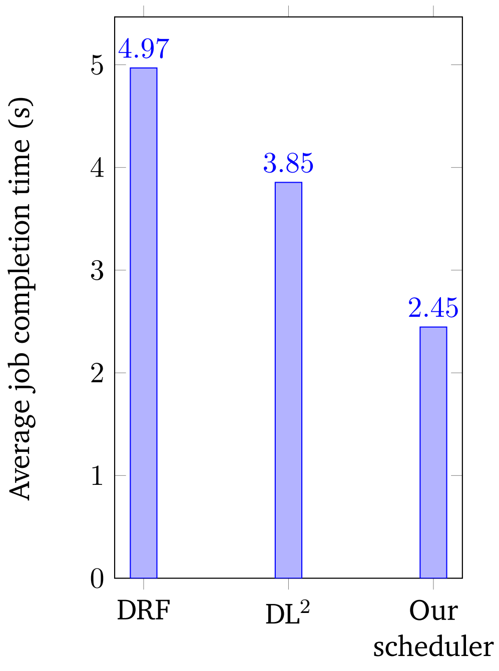


Figure 5.6: Average job completion time of our scheduler compared with DRF and DL^2.

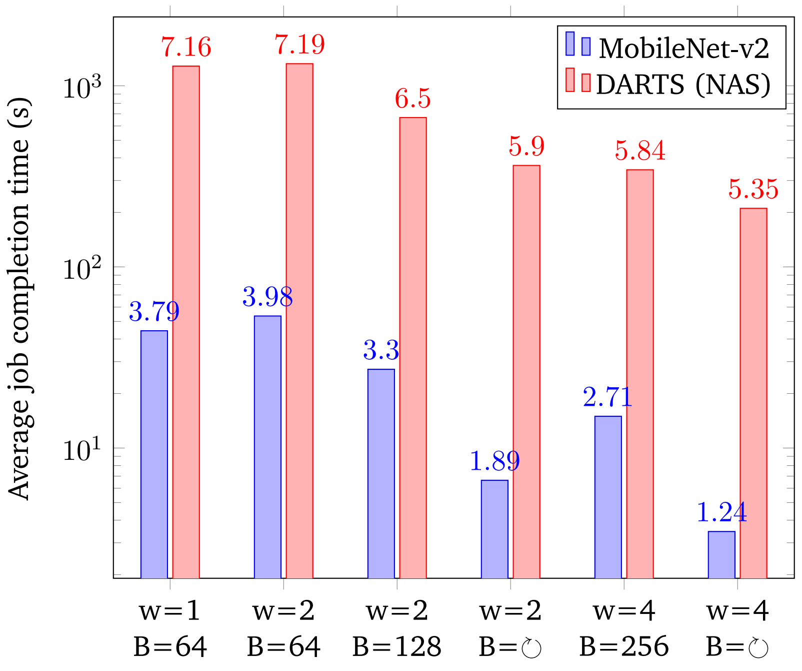


Figure 5.7: Average job completion time for different number of GPU workers with and without adaptive batch size.

A noteworthy benefit of adaptive batch size tuning is the improvement of resource utilization. Figure 5.8 depicts the GPU utilization of two GPUs during the first few training epochs of ResNeXt. The first epoch used the initial batch size of 128 per GPU, marking the total GPU utilization of $\sim$50% per GPU. Our adaptive batch size tuning increases the batch size to 253 after the first epoch, increasing the utilization to above 75% per GPU. Having access to more training data and less inter-GPU synchronization can explain such an improvement.

Looking again at Figure 5.8, we notice several sudden utilization drops throughout the training that we annotated with gray rectangles. After further investigations, we found out that these drops are related to the end of each epoch, where all the batches are processed. These intervals can be effectively used for checkpointing, task migration, and further training adjustments since less interruptions will be caused.

5.4.2 The impact of adaptive batch size tuning on NAS

Since one of the goals behind our scheduler is to enhance the performance of NAS training, we conducted experiments on DARTS [107], which is a popular NAS method. We used four A100 Nvidia GPUs and trained DARTS with default settings on the CIFAR-10 dataset. Compared to the fixed batch size set at the beginning of the training, adaptive batch size tuning led to 45% faster execution while offering the same accuracy. Such an observation validates that selecting the right batch size given the available hardware is important to fully utilize the training hardware and produce the trained model in the minimum amount of time.

Although Pollux does not support multiple data loaders that are common in NAS training, our scheduler resolved this issue and effectively tuned NAS jobs. Moreover, we noticed that Pollux sets a threshold of 5% for updating the batch size at each training iteration. In other words, Pollux adapts the batch size only if the expected goodput exceeds more than 5%. In our experiments, we noticed that such a design

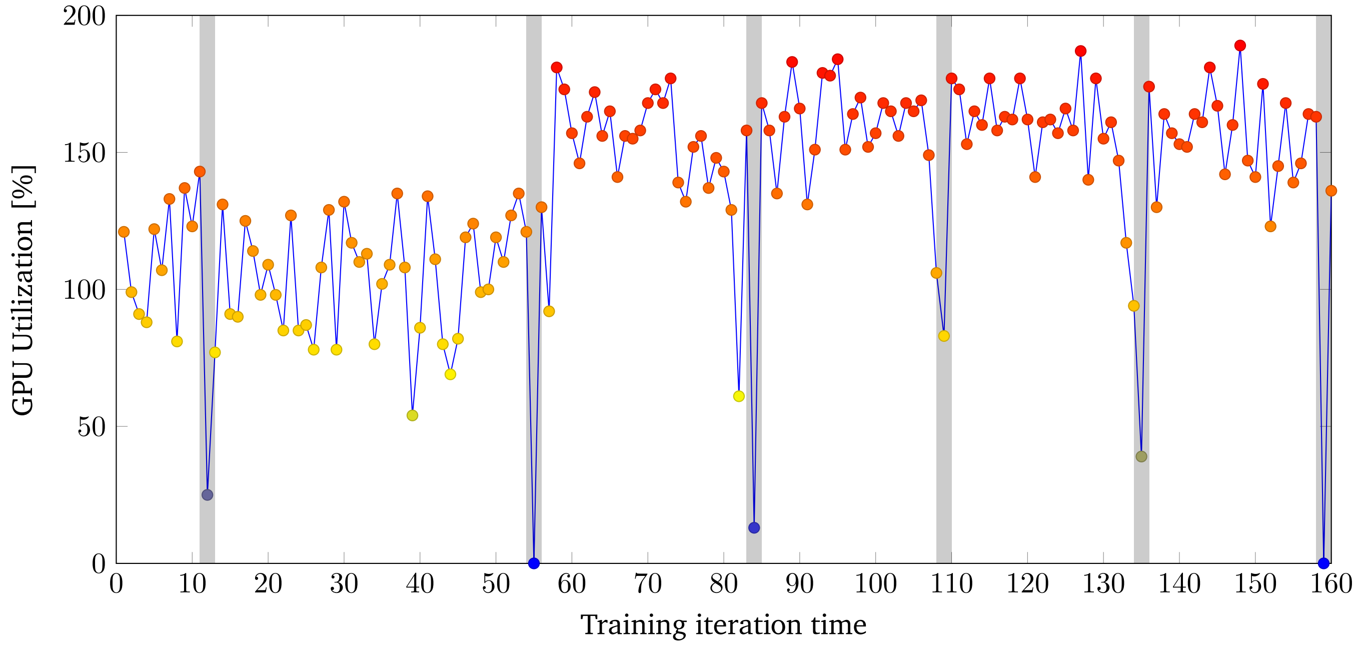
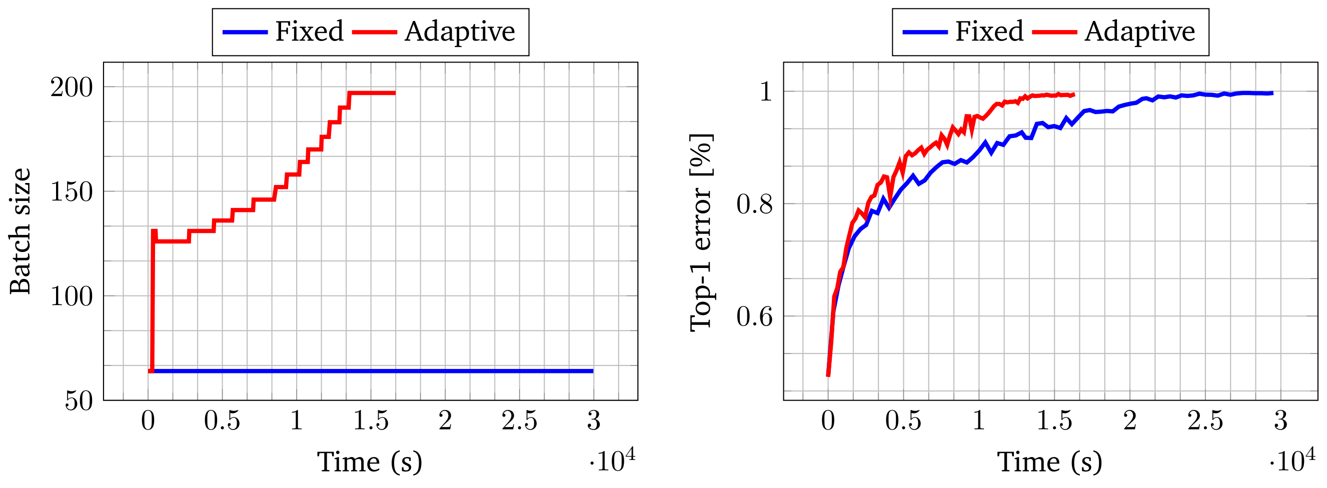


Figure 5.8: GPU utilization of two NVIDIA GeForce GTX 1080 Ti's. The first epoch uses the initial batch size of 128 and lasts for 58 seconds.

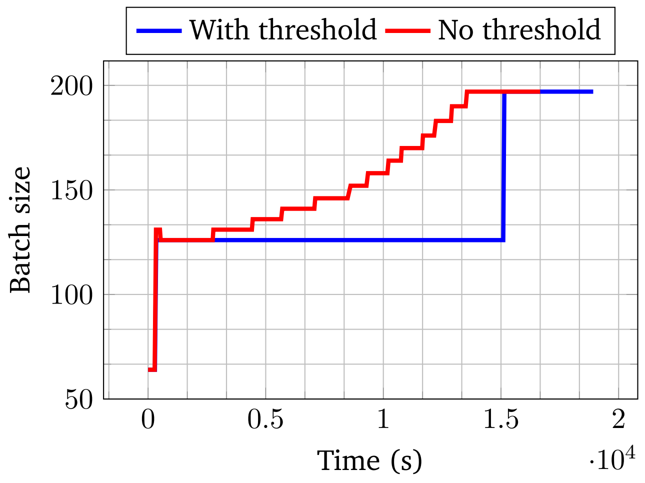


(a) Batch size

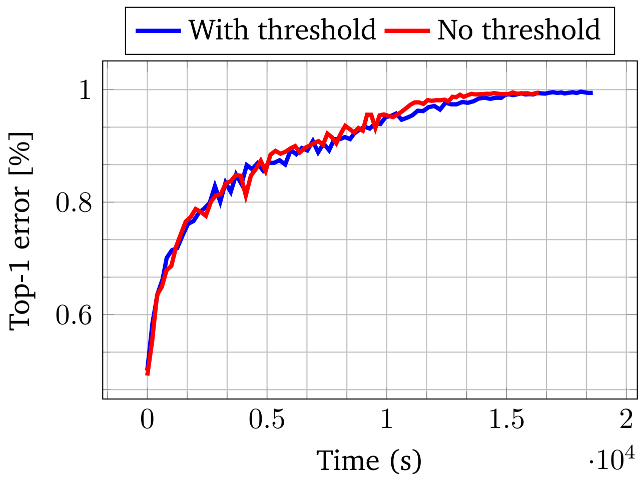
(b) Accuracy

Figure 5.9: Adaptive batch size tuning versus fixed batch size.

choice is not suitable for NAS training, as each epoch takes a longer amount of time to finish. Eliminating the threshold provided us with lower job completion time and, therefore, higher speedups. As depicted in Figure 5.10, we noticed an increase in the batch size at each iteration while Pollux waits until the threshold is met. Given that both configurations attain similar accuracy, but ours managed to finish earlier, we argue that the no-threshold mechanism works better for NAS jobs.



(a) Batch size



(b) Accuracy

Figure 5.10: Adaptive batch size tuning of our method (no-threshold) versus Pollux (with threshold).

5.4.3 Evaluating the network pruning

We evaluated our network pruning method by performing FLOPs-constrained structured pruning on several bulky and mobile-friendly ConvNets, including ResNet-20/56 [85], VGG-16 [177]), and MobileNet-v1/v2 [150, 151]. To show the superiority of our approach and validate the effectiveness of the learning-based embedding, we compared our approach with three existing methods:

- Uniform, shallow, and deep empirical policies [178, 179].
- Handcrafted channel reduction methods, such as SPP[180], FP [179], and RNP [181].
- Regularization-based methods, such as MorphNet [182] and SSL [183].
- RL-Based AutoML methods, such as automatic model compression method AMC [176], which manually defines DNN layer embeddings, and random search with reinforcement learning (RS), which does not leverage any layer embeddings.
- Other pruning methods, such as DSA [184] and Rethink [185].

Datasets. We conducted our experiments using the datasets CIFAR-10 [186] and ImageNet [187]. To accelerate the search process on CIFAR-10, we split the training set into two partitions of $15k$ and $5k$ images. We used the $15k$ training set to rapidly fine-tune the candidate model and the remaining $5K$ images as the validation set to calculate the reward function. To deal with the ImageNet dataset, we split $5k$ images from the training set as the validation set to calculate the reward. However, the validation accuracy of the ImageNet dataset is sensitive to the compression, as with high compression ratios, the accuracy drops considerably without fine-tuning. Thus, the RL agent can not get a valuable reward. As a remedy, we decompose the pruning on the ImageNet dataset into several stages and add one epoch of fine-tuning for each search episode. For instance, to obtain a 49% FLOPs model compared to the original network, instead of performing a single step 49 % FLOPs pruning, we prune the target DNN two times, each with 70% FLOPs constraint (i.e., 70%FLOPs $\times$ 70%FLOPs = 49 % FLOPs).

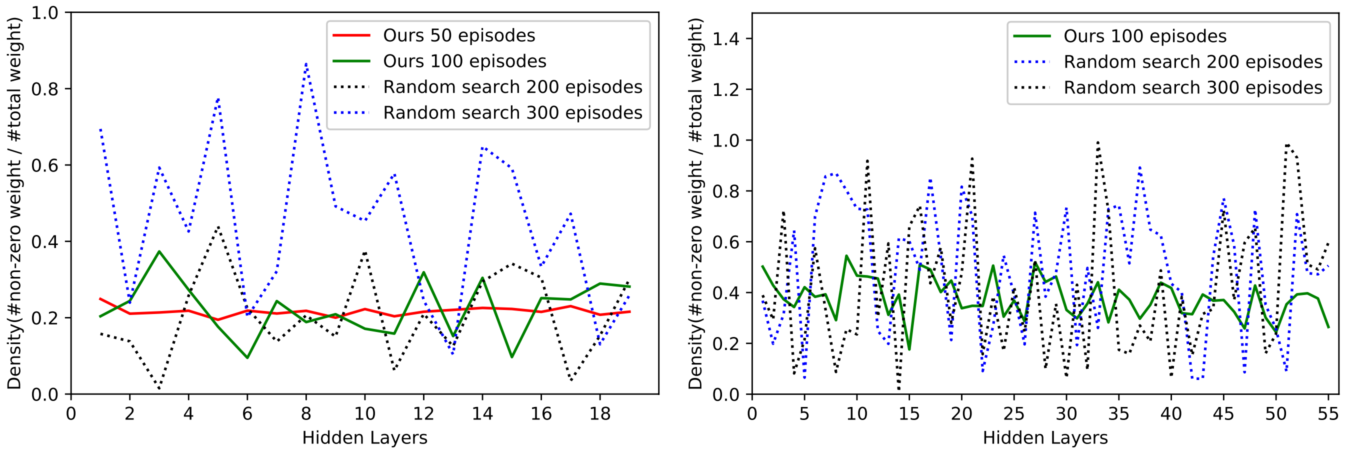


Figure 5.11: Comparing the pruning stability of our method across different layers with random search on ResNet-20/56.

The effectiveness of graph encoder-decoder for network pruning

We argued that the automatic extraction of layer embeddings from graph topology is more effective than handcrafted pre-defined features. Thus, to validate our claim, we analyzed our layer embeddings with existing methods. First, we demonstrate the necessity of layer embeddings by comparing the accuracy with a method that does not use any feature. Then, we compare the accuracy achieved without layer embedding with the ones that AMC [176] uses.

Learning-based vs. no embedding. To prove the benefit of using layer embeddings for network pruning, we compared our method with a simple random search RL agent that does not use any layer embedding. We configured all the hidden layers to a fixed one-hot vector as the RL agent's environment state and set the remaining parameters similar to our RL setup. Figure 5.11 shows that our method achieves better results compared with the blind random search on ResNet-20 and ResNet-56. Particularly, for ResNet-20, random search uses 200 and 300 episodes, achieving a compressed network with 71% and 88.41% validation accuracy, respectively. On the other hand, AGMC only searches for 50 episodes and 100 episodes with a validation accuracy of 93.8% and 94.6%, respectively. Hence, AGMC achieved a higher compression ratio and accuracy with considerably fewer search episodes.

Moreover, looking at the compression ratio of each individual hidden layer, we observed that AGMC tends to apply a more uniform pruning policy. Such an observation is similar to the uniform pruning policy[179], which shows that the uniform policy can be more beneficial for network pruning.

We also compared the performance of our method with the random search on ResNet-56 under different FLOPs-constraints. ResNet-56, with its deeper network structure, is more challenging to prune properly. Figure 5.12 illustrates the validation accuracy with respect to different pruning ratios. We observed that AGMC outperforms the random-search method with a higher accuracy given the same pruning ratio. With 10% FLOPs reduction, the validation accuracy of both methods are almost similar. However, with 90% FLOPs reduction, AGMC excels by a large margin.

Learning-based vs. manually-defined layer embeddings. AMC utilizes a pre-defined set of layer embeddings for its RL agent to search for the best pruning policy. Such an embedding vector consists of 11

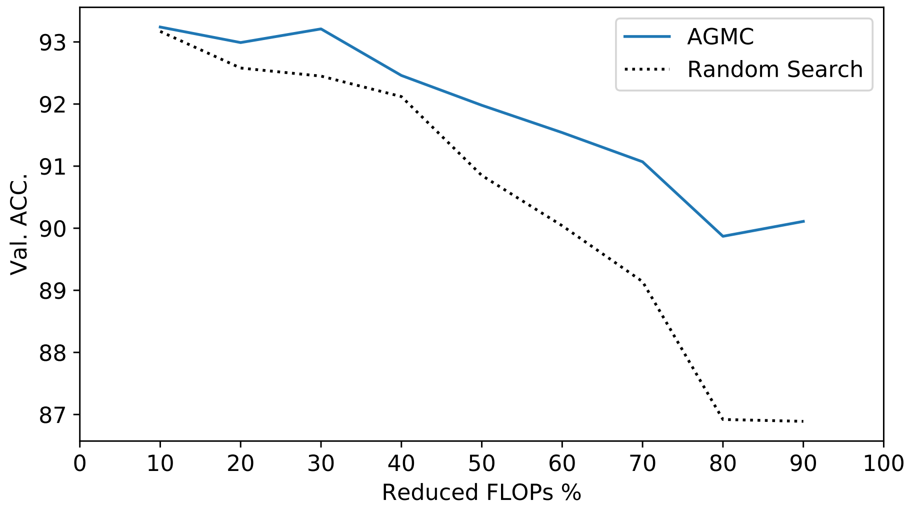


Figure 5.12: Validation accuracy comparison of random search and AGMC on ResNet-56 under different FLOPs.

features denoted as $s_t = (t, n, c, h, w, stride, k, FLOPs(t), reduced, rest, a_{t-1})$, where t is the layer id, the dimension of the kernel is $n \times c \times k \times k$, and the input is $c \times h \times w$. $FLOPs(t)$ is required FLOPs to execute the layer. $Reduced$ is the total number of FLOPs reduced in previous layers and $rest$ is the number of remaining FLOPs in the upcoming layers. We argue that such a rigid design cannot capture all the essential features such as topological information or the number of parameters available in each layer.

To make a fair comparison, we also set our learning-based embedding vector size to 11 (i.e., $S \in \mathbb{R}^{T\times 11}$) and evaluated it with AMC's embeddings on a 50% ResNet-20 pre-trained with the CIFAR-10 dataset. Figure 5.13 depicts the influence of each feature on achieving a lower top-1 error. Relying on $stride$ as the only layer embedding leads to an error rate of 31%, as it provides not enough information to discern the layers. Combining $stride$ with the number of filters n reduces the error rate to 18%. Consequently, combining all the features brings us to a 10.2% error margin. Nonetheless, our learning-based layer embedding outperformed the overall accuracy of AMC by a factor of two and achieved an error rate of 5.38%.

Analytical study on network pruning performance

We evaluated AGMC on various deep-learning models from bulky to mobile-friendly ConvNets, the last of which are already compressed and challenging to prune further. ResNet and VGG networks, often considered as bulky DNNs, involve billions of parameters, making them quite challenging to be deployed on edge devices due to their high memory consumption. We perform FLOPs-constrained pruning on over-parameterized DNNs by leveraging the RL agent to search for pruning ratios for each convolutional layer. However, ResNet networks contain residual connections and different pruning ratios between residual connected layers can cause feature-map dimension mismatch. As a remedy, we share the pruning ratio between the residual connection layers.

On the other hand, MobileNet models [150, 151] are considered as mobile-friendly DNNs that are equipped with customized convolutional blocks to reduce the number of parameters, making them suitable for deployment on edge devices. For instance, the MobileNet-v1 block splits a traditional convolution into a

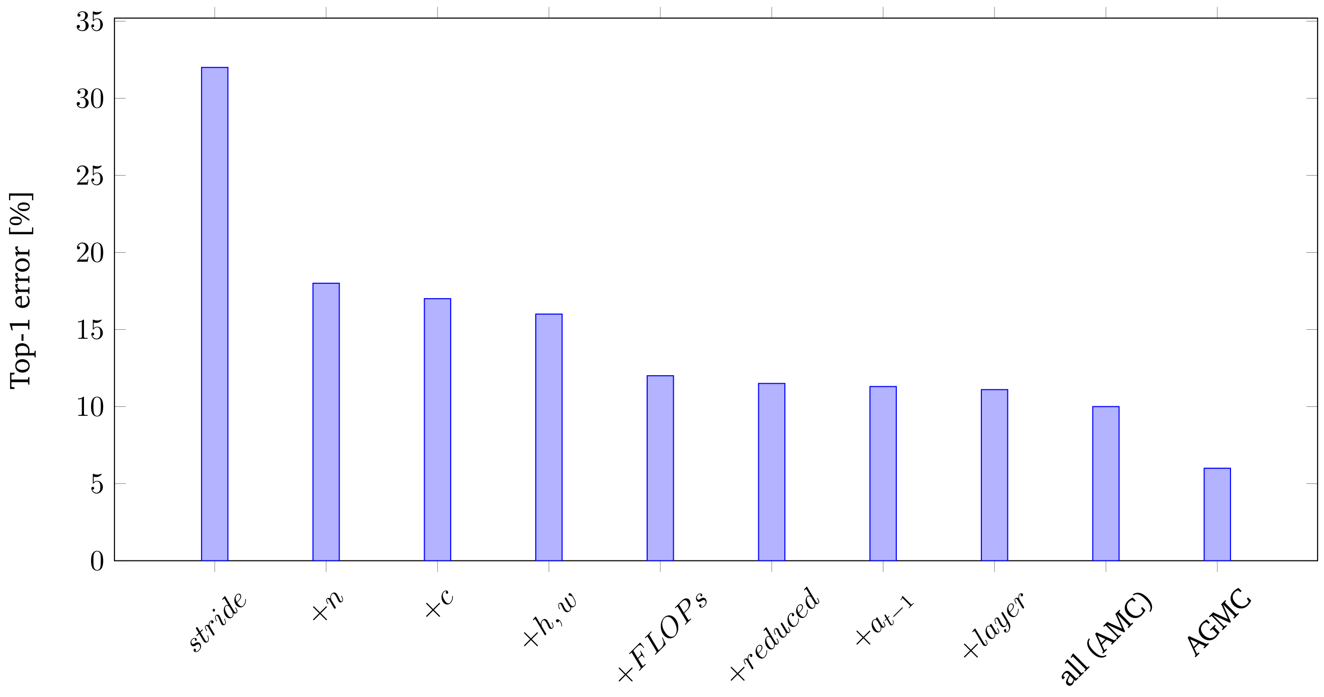


Figure 5.13: An error-rate comparison for individual AMC layer embedding, overall AMC, and AGMC layer embedding. Our method has achieved roughly 2× less error rate.

pair of point-wise and depth-wise convolutions. Furthermore, based on MobileNet-v1, MobileNet-v2 adds an additional linear expansion layer and introduces residual connections. To maintain the characteristics of the mobile-friendly DNNs, we have developed specific pruning strategies for them.

MobileNet-v1. The MobileNet-v1 block contains a depth-wise and a point-wise convolution. Instead of pruning them separately, we view the two convolutions together and only prune point-wise convolutions. The reason is that the depth-wise convolution only operates on one input channel and pruning that filter will cause information loss for the corresponding channel.

MobileNet-v2. Similar to MobileNet-v1, we prune linear expansion layers and point-wise convolutional layers. Since residual connections are between linear expansion layers, we share the linear expansion layers' pruning ratio.

Table 5.1 reports the pruning performance of AGMC in comparison with several state-of-the-art methods. Our method outperforms the empirical policies by a large margin on ResNet-20 and 3.6% on ResNet-56. Compared with AMC, AGMC achieved 5.02% and 2.56% higher accuracy on ResNet-20 and ResNet-56, respectively. For the VGG-16 model trained on the ImageNet dataset, we compared AGMC with state-of-the-art handcrafted channel reduction methods (i.e., SPP[180], FP[179], and RNP[181]) and AMC[176]. Results show that AGMC outperformed all the baselines methods by a large margin.

On MobileNet-v1/v2, we compare AGMC with the uniform pruning policy and the RL-based method AMC. Compared to the uniform policy, which sets the compression ratio uniformly, AGMC achieves a higher compression ratio with only 1.2% test accuracy loss. Furthermore, our efficient layer embedding outperforms AMC on MobileNet-v1 and MobileNet-v2, with the same target FLOPs.

We also measured the inference speed of the compressed ResNet-20/56, VGG-16, and MobileNet-v1 on

Table 5.1: Detailed comparison of AGMC to other methods according to pruning ratio, latency, and GPU memory usage.

Model	Method	FLOPs	$Acc.\%$	$\Delta Acc.$	Latency	GPU Memory
ResNet-20 (CIFAR-10)	Original	100%	91.73	0	0.32ms	1.1MB
	Deep	50%	79.6	-12.13	-	-
	Shallow	50%	83.2	-8.53	-	-
	Uniform	50%	84	-7.73	-	-
	SSL	52%	89.78	-2.39	-	-
	MorphNet	48%	90.1	-2.07	-	-
	Rethink	60%	91.07	-1.34	-	-
	SFP	58%	90.83	-1.37	-	-
	DSA	50%	91.38	-0.79	-	-
	AMC	50%	86.4	-5.33	-	-
	AGMC	50%	**91.42**	**-0.31**	0.30ms	565KB
ResNet-56 (CIFAR-10)	Original	100%	93.39	0	0.52ms	3.4MB
	Uniform	50%	87.5	-5.89	-	-
	Deep	50%	88.4	-4.99	-	-
	SSL	47%	91.22	-1.90	-	-
	MorphNet	52%	91.55	-1.57	-	-
	Rethink	50%	93.07	-0.73	-	-
	SFP	50%	92.26	-1.33	-	-
	AMC	50%	90.2	-3.19	-	-
	AGMC	50%	**92.76**	**-0.63**	0.48ms	1.8MB
VGG-16 (ImageNet)	Original	100%	70.50	0	20.52ms	528MB
	FP	20%	55.9	-14.6	-	-
	RNP	20%	66.92	-3.58	-	-
	SPP	20%	68.2	-2.3	-	-
	AMC	20%	69.1	-1.4	-	-
	AGMC	20%	**70.35**	**-0.15**	16.82ms	387MB
MobileNet-v1 (ImageNet)	Original	100%	70.60	0	11.02	17MB
	Uniform	56%	68.10	-2.50	-	-
	Uniform	41%	66.90	-3.70	-	-
	AMC	40%	68.90	-1.70	-	-
	AGMC	**40%**	**69.40**	**-1.2**	10.52ms	14MB
MobileNet-v2 (ImageNet)	Original	100%	71.80	0	-	-
	Uniform	70%	69.80	-2.00	-	-
	AMC	70%	70.80	-1.00	-	-
	AGMC	70%	**70.87**	**-0.93**	-	-

an Nvidia RTX 2080Ti GPU and compared it with the original model. We used a batch size of 32, and the compressed models are tested on CIFAR-10 and ImageNet datasets. As shown in Table 5.1, our pruning achieves notable GPU memory reduction. For VGG-16, the original model's GPU memory usage is 528 MB since it has dense layers and its first dense layer contains 25088 neurons. The 20% FLOPs VGG-16 with pruned convolutional layers significantly reduced the feature map size input to dense layers, taking 141 MB memory less than the original. Moreover, all of the models pruned by AGMC achieved remarkable inference speedup without losing considerable test accuracy. For instance, the 20% FLOPs VGG-16 achieved $1.22\times$ speedup on the ImageNet dataset.

5.5 Related work

This section reviews former studies and efforts related to hyperparameter tuning, DL-specific schedulers, and model compression methods.

5.5.1 Hyper-parameter tuning

Many former studies explored automatic hyper-parameter tuning for machine learning and deep-learning models [188, 189]. Various tuning parameters, including batch size and learning rate, can be optimized using these methods. However, these projects are primarily designed to increase the overall model accuracy and not the training efficiency.

Recently, adaptive hyper-parameters tuning for enhancing training efficiency is gaining momentum. These works mainly tune batch size and learning rate parameters, as they have the biggest influence on model quality and training progress. For instance, AdaBatch [190] increases the batch size at specific training iterations and linearly scales the learning rate. Another study [191] suggests increasing the batch size without decaying the learning rate. CABS [192] and Pollux [158] adaptively tune batch size and learning rate parameters during training using gradient statistics. Similarly, KungFu [168] also supports adaptive training of a single job by a unified set of APIs to monitor the training progress and define custom adaptation policies. None of the above works, however, studied NAS jobs and their training progress.

5.5.2 DL-specific job schedulers

As our scheduler falls into the category of multi-tenant scheduling, we mainly consider previous work that manage resources in a shared multi-tenant environment. Prior works can be grouped into two categories: (1) rigid, (2) elastic, and (3) elastic-adaptive schedulers.

Rigid schedulers

Rigid schedulers do not consider the performance scalability of DL-training jobs given the number of allocated resources. For example, Tiresias [193] and Gandiva [157] demand the user to define the number of GPUs at the time of job submission, which remains the same throughout the entire training. A slight advantage of Gandiva is its enhanced resource utilization through fine-grained time-sharing and job packing. Moreover, Gandiva's dynamic GPU allocation is not based on the job's scalability.

Elastic schedulers

This group of schedulers automatically select and adjust the number of resources allocated to each job given their progress to maximize the resource utilization and reduce the overall training time. Dolphin [194] is an elastic data-parallel machine-learning framework, in which the allocation of parameter servers and workers is dynamically adjusted based on a cost model. SLAQ [195] employs an online-fitting approach to estimate the training loss of classical machine-learning algorithms. It supports a broad set of optimization and solves a min-max problem to provide fairness among the jobs. Another scheduler is Dorm [196], which uses a utilization-fairness optimizer to schedule machine-learning jobs. These works did not consider special properties of DL jobs.

In contrast, Optimus [159] is a DL-specific scheduler that is based on the parameter-server execution model. It dynamically adjusts the allocation of workers and parameter servers based on online-fitted resource-performance models to achieve the best resource efficiency with minimum average job completion time. Optimus attempts to equally divide model parameters onto parameter servers to achieve load balancing due to its dependence on the parameter-server architecture. Bao et al. [197] also designed an online scheduling method for DL jobs. However, such works are based on simplified assumptions and a predefined performance model that can potentially lead to suboptimal performance in corner-case situations. Lastly, DL^2 scheduler [160] is a follow-up work from the authors of Optimus. They used reinforcement learning to perform online scheduling. By outperforming Optimus, they showed that using performance models is not always efficient.

In summary, existing elastic schedulers do not consider the statistical efficiency of DL training and the correlation of resource allocation with training parameters.

Elastic-adaptive schedulers

The last type of schedulers are the ones that not only adjust the allocated resources given the progress of the DL training but also adapt the training hyperparameters to enhance the training progress further. To the best of our knowledge, Pollux [158] and KungFu [168] are the only schedulers that attempt to schedule DL jobs. However, none of them were applied to NAS jobs. We also believe that performing online scheduling using reinforcement learning can further improve performance.

5.5.3 DNN model compression and acceleration

Various techniques have been introduced to compress DNN models into smaller versions and accelerate them with a variety of techniques. Since the focus of our work is on network pruning, we provide a detailed summary of such methods along with a brief description of other model compression methods. It is also worth mentioning that various methods are based on a combination of the methods below.

More optimal layers. Traditional convolution layers take a lot of space and computing power. With recent advances in DNN algorithms, more efficient layers have been introduced as a remedy. For instance, fully connected layers are never used in modern networks, as they have been replaced with 1×1 convolutions. Furthermore, mobile-friendly networks such as MobileNet [198] have introduced the concept of depth-wise separable convolutions, which require much fewer parameters but offer the same or even higher accuracy.

Knowledge distillation. This category of model-agnostic compression techniques resembles the "teacher-student" analogy, in which a small model is trained to imitate a larger pre-trained model [199, 200, 201]. Particularly, the knowledge is transferred from the larger to the smaller model by minimizing a loss function. Using this method, the student model inherits higher accuracy than the teacher and is more efficient for deployment on edge devices due to its lower computational complexity. However, the lack of an appropriate teacher model causes suboptimal student models. Moreover, training multiple student models requires a significant amount of computational resources [202].

Tensor factorization. It decomposes DNN weights into more lightweight pieces [203, 204]. A notable effort is the factorization of ordinary 3×3 convolutional layers into the sequence of 1×3 and 3×1 layers [205]. Moreover, Zhang et al. [206] factorized convolutional layers into 3×3 and 1×1. Both methods yielded higher performance.

Quantization. The process of reducing the number of bits used for representing the weights in a DNN is called quantization. Typically, DNN models are designed with 32-bits floating point numbers (i.e., FP32). Such models are bulky and memory intensive. By reducing the bit width to FP16 or even 8-bits integers, we can extensively save memory and space without incurring significant loss [89, 207, 208]. Using even lower bit widths, such as 4/2/1-bits, has also been evaluated with rather promising results [209].

Network pruning. Among other compression methods, network pruning [89, 210, 211] has been widely studied over the last decade. It shrinks the size of over-parameterized networks by eliminating a portion of parameters within each DNN layer via two different methods: (1) fine-grained pruning and (2) structured pruning. Fine-grained pruning [89] removes each unimportant element in weight tensors. On the other hand, structured pruning [179] prunes an entire block of weight tensors, such as channels, rows, columns, and blocks. Despite the higher compression ratio offered by fine-grained pruning, such a method has shown to be useful only on specific hardware accelerators [212, 213], as specialized algorithms are required to handle irregular sparse tensors. In contrast, structured pruning provides regularly pruned weights and can be used on commodity hardware.

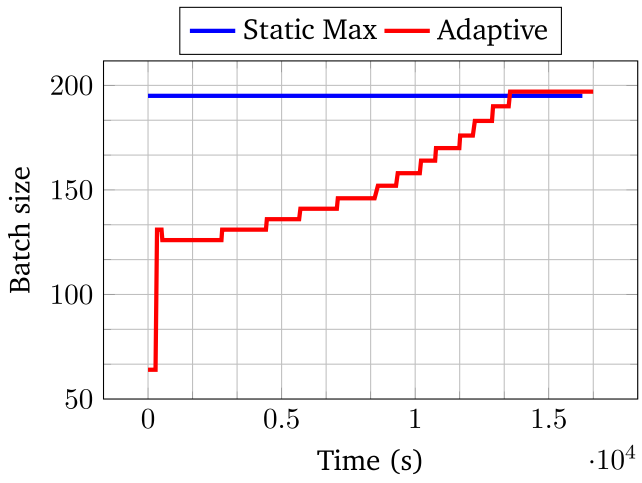
Various structured pruning policies such as handcrafted and AutoML-based methods are available. Empirical pruning policies (e.g., uniform, shallow, and deep [178, 179]) are mostly handcrafted by experts, demanding human effort and domain expertise. The uniform policy [178] sets the compression ratio uniformly, while the shallow and deep policies aggressively prune shallow and deep layers, respectively [179]. Such handcrafted empirical policies heavily rely on manually defined rules and fail to achieve the optimal compression ratio. Other handcrafted methods focusing on channel pruning are SPP [180] and FP [179]. SPP prunes DNN layers by measuring the reconstruction error. FP estimates the sensitivity of each layer, where layers with lower sensitivity are pruned more aggressively. Recently, RL-based automatic network pruning methods [214, 176, 215] have yielded better results. He et al. [176] proposed a reinforcement-learning-based method for network pruning, yet they still use handcrafted features to represent DNNs and ignore the rich structural information within computational graphs.

5.6 Discussion

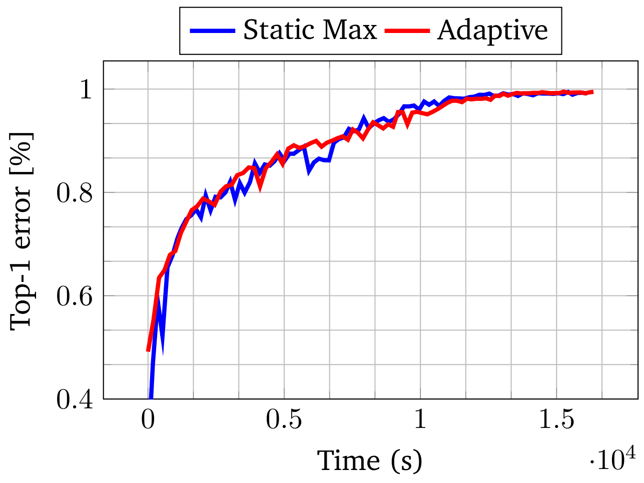
This chapter presented our early effort towards the acceleration and scalability of DL/NAS training. We showed the benefits of out platform in terms of speedup and accuracy in Section 5.4. Here, we aim to shed some light on other aspects, particularly by discussing further design choices and insights.

5.6.1 Distributed gradient synchronization method

We built our DL training platform with the assumption of using the all-reduce method for synchronizing the gradients across all the worker nodes. We made such a design choice partly because of fewer parameters (i.e., only number of workers) that need to be tuned. On the contrary, the parameter-server method requires the number of parameter servers and workers, making the tuning process much more complex.



(a) Batch size



(b) Accuracy

Figure 5.14: Adaptive batch size tuning versus using maximum possible batch size.

Moreover, parameter-server-based methods have shown to offer better results in those environments where the DL training is supposed to run on a large number of unreliable and heterogeneous nodes. However, all-reduce method works better on a small number of fast compute nodes with high bandwidth links [216]. Since computing clusters often provide the latter case, using all-reduce architecture seems more reasonable. Nevertheless, our training platform can be easily extended to support parameter-server architecture as well.

5.6.2 The real benefit of adaptive batch size tuning

In the experimental results section, we presented various aspects of our scheduler and the effect of adaptive tuning on NAS training jobs. Regardless of the positive impacts, one might prefer to use maximum batch size according to the capability of the hardware instead of adaptive batch size tuning. For this purpose, we compared adaptive batch-size tuning with constant maximum batch size and presented the results in 5.14. We trained a NAS method (DARTS) using the CIFAR-10 dataset on four Nvidia A100 GPUs for 90 epochs. Our experiments indeed show that using the maximum batch size can potentially lead to a slightly faster training. However, the accuracy diagram (see Figure 5.14b) shows that static maximum batch size suffers from unstable accuracy throughout the training. On the contrary, adaptive tuning attempts to maintain statistical efficiency, leading to a more stable and accurate model.

Moreover, the results presented in Figure 5.14 assumed that the number of GPUs does not change throughout the training process. However, if the scheduler decides to adjust the resources, the batch size needs to be updated as well. Using static batch size might lead to training failure or in the best case lead to inefficient execution. In contrast, our adaptive mechanism can swiftly recover and adjust the batch size according to the new resource configuration without any manual intervention.

In summary, our experiments prove that adaptive batch size tuning not only frees the user from the manual selection of hyper parameters, but also offers higher accuracy and stability for the trained model.

5.6.3 Further insights on network pruning using RL

The time required by the RL agent to find the best pruning policy largely depends on the RL agent's action space. In our case, the action space is the hidden layer's pruning ratio, which is between $[0, 1)$. Thus, even with deep networks, the search space is still manageable, as the number of layers are limited. For instance,

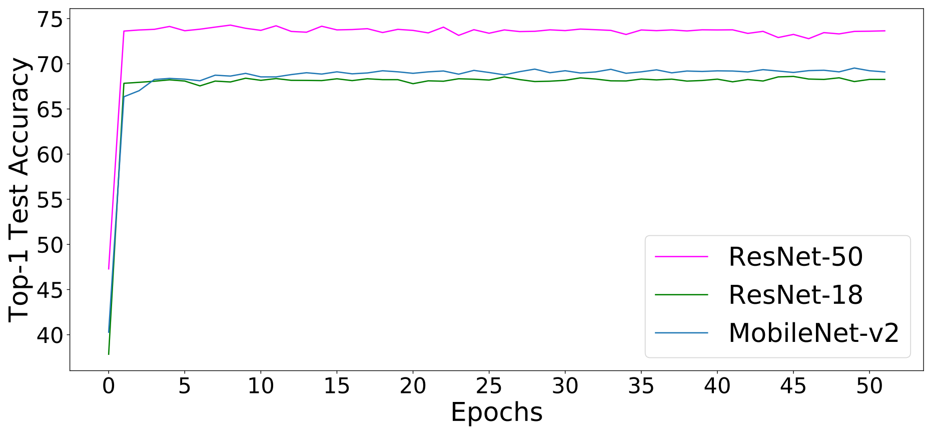


Figure 5.15: Accuracy recovery after different fine-tuning epochs.

the largest ResNet contains 110 convolutional layers. Compared to many popular and huge RL tasks that we find in self-driving cars or AlphaGo, our search space is acceptable and practical. Additionally, we recorded the RL search time on ResNet-56 with 300 episodes. It takes (320 ± 30) seconds on an RTX 8000 GPU to finish the entire search for the pruning ratios.

Furthermore, we observed that the models pruned by AGMC can quickly recover from the accuracy loss (see Figure 5.15). After only one epoch, we can compensate for almost all the accuracy loss. For all DNNs, the accuracy gap between one epoch fine-tuning and 150 epochs fine-tuning is less than 3%. Thus, we only needed to perform less than 50 fine-tuning epochs, which is far less than other state-of-the-art methods.

5.7 Conclusion

In this chapter, we argued that the scalability of deep neural networks could be improved not only through a more robust training platform but also by using smaller networks. Thus, we introduced a network training and pruning platform to address the scalability of deep-learning models.

We showed that blind horizontal scaling without adapting the training job causes resource underutilization. Thus, we proposed an elastic DL-aware scheduler that supports job-level and cluster-level scheduling, aiming to increase the throughput and efficiency of the training job. Our experiments validated that our scheduler expedites the training of normal deep-learning training as well as neural architecture search jobs. Furthermore, we noticed that adaptive batch size tuning could enhance resource utilization.

Moreover, by taking advantage of graph convolutional networks and reinforcement learning, we proposed a learning-based method to extract topology information of a given ConvNet. Based on the extracted information, our compression method automatically explores network pruning policies to find the best strategy for reducing the computational complexity of a given network while maintaining accuracy. Under the same compression ratio, experimental results on bulky and mobile-friendly ConvNets showed that learning the embeddings from the ConvNet structure can outperform all the rule-based embedding methods by a large margin. On bulky ConvNets, such as ResNet-56, our method performs better than all the baselines with only 0.63% accuracy drop. For already compact networks such as MobileNet-v1, we also achieved a higher compression ratio than baselines with only 1.2% accuracy loss. Such a low accuracy loss shows the superiority of our method compared with state-of-the-art methods.

By preventing significant accuracy loss while pruning, we can apply more aggressive compression and obtain a smaller model. Thus, with respect to the scalability perspective, the resulting model can now be potentially trained faster than the original model and also run faster on the target device.

6 Conclusion and outlook

The main goal of this dissertation was to present our solutions for performance analysis and optimization of data-intensive applications. Particularly, we aimed to design tools and platforms for accelerating and scaling their performance. Given the recent quantum leap in artificial intelligence, it is apparent that smart devices are pervading all aspects of our life. Thus, as a ubiquitous data-intensive use case, we decided to focus on deep-learning workloads. We bundled our threefold techniques into a single inclusive term, namely “performance engineering”, to highlight the necessity of a combined approach to achieve the desired performance. In this chapter, we summarize the contributions of this dissertation, highlight the impact of our work, and present the next possible steps as our future work.

6.1 Summary of contributions

From the perspective of performance engineering, this dissertation presented three main contributions. Each contribution contains additional novelties that are explained in detail in Chapters 2 to 5. In the following, we enumerate and summarize all the contributions that we have presented in our work.

Performance profiling. As data-intensive applications deal with a large amount of data and communication events, our first contribution was a novel performance profiling method to identify potential communication bottlenecks that arise from ineffective thread communications. Instead of analyzing the entire application, using LLVM instrumentation, we can find hotspots (i.e., regions with a high degree of communication) within the code and fully focus on those code sections.Using our memory-efficient code instrumentation profiler, we achieved the following points.

- We introduced a set of hardware-independent metrics to identify data communication bottlenecks. Inspired by data reuse distance, we proposed communication reuse distance (CRD) that reflects the effect of communication on the cache. Moreover, our proposed communication reuse ratio (CRR) matrix sheds light on the amount of reuse after a true communication between threads.
- We quantitatively analyzed the homogeneity and balance of CRR matrices to give a better perspective on comparing code variants.
- We showed that our method could detect communication bottlenecks in different code regions and also help programmers apply a suitable type of optimization, increasing the performance by up to 56%. Furthermore, we demonstrated that such an analysis is helpful for determining the input scalability of a given application with regard to its cache usage.

Performance portability. Our second contribution was designing and implementing a framework for the automatic generation of efficient and performance-portable convolution kernels for various GPU platforms. We analyzed and compared the syntax and taxonomy of three GPU programming interfaces: CUDA, OpenCL, and Vulkan. Our comparative study illuminated the similarities and differences among the three APIs, paving the path for designing a code portability layer. We employed a synergy of meta-programming, symbolic execution, and auto-tuning to specialize a challenging ConvNet operation named Winograd convolution. Our detailed achievements in this line of work are listed below.

- We developed a method for generating tensor GPU kernels coded in any of the target APIs from a single source that abstracts away the syntactic differences. We implemented our approach in a state-of-the-art CNN inference framework called Boda and analyzed the programmability and performance portability of the generated kernels. Based on our experiments, our method reduces the programming effort by 98% when code portability between different APIs is demanded. Furthermore, we showed that Vulkan offers better performance than other APIs on our convolution benchmarks and sometimes performs better than vendor libraries. Such an observation can encourage the programmers to consider Vulkan as their first library of choice for deploying GPU codes on devices with no application-specific library support.
- We proposed a method based on symbolic computation to create minimal yet efficient recipes that replace the straightforward matrix multiplication method within Winograd transformations. Our empirical evaluation illuminated that choosing the right output tile size m, depending on the filter size r, can significantly reduce the number of arithmetic operations while offering acceptable accuracy (e.g., $F(m = 6, r = 3)$, $F(m = 4, r = 5)$). To the best of our knowledge, this critical observation went unnoticed so far.
- We enabled generating performance-portable Winograd convolutions with the help of template meta-programming. Our runtime analysis shows that we can use the same Winograd meta-code to run on a multitude of GPU platforms, including a mobile GPU, and compete with vendor ConvNet libraries, such as cuDNN, MIOpen, and the ARM compute library.

Performance scalability. Lastly, to enhance the scalability of deep-learning applications, we developed a specialized training platform and equipped it with a novel topology-aware network pruning algorithm. The overarching goal is to enable rapid and scalable network training, neural architecture search, and model compression. As a result, a training job can be easily scaled to a multitude of computing nodes, leading to faster model design with less operating costs. Further achievements are described below.

- We developed a deep-learning-aware scheduler that supports cluster-level and job-level scheduling. Using our scheduler, deep-learning training jobs can be scaled up efficiently to a multitude of nodes, depending on the number of concurrently running jobs. Moreover, our proposed scheduler considers the unique behavior of neural architecture search jobs and adapts the required resources to better harness the computing power and expedite the network design process.
- We proposed an automatic model compression method by employing graph convolutional networks and reinforcement learning. To the best of our knowledge, this is the first work to model DNNs as computational graphs to enhance model compression. We showed the superiority of our learning-based method by comparing its performance with rule-based DNN embedding, where we outperform them and achieve higher accuracy with fewer required FLOPs.

6.2 Outlook

We believe that the provided methods and techniques in this dissertation do not necessarily end here as they have the potential to be extended and studied in future research projects. Moreover, due to the increasing demand from the industry to use and deploy AI applications on their products, we strongly believe that the idea behind this work has the merit to be transferred to industry. All in all, we envision various possible future research directions that are worth investigating. This section provides an outlook towards such research and development opportunities.

6.2.1 Extended application domain

While the first contribution of our thesis was generic enough to be used for any data-intensive application, the remainder of our work primarily targeted deep-learning applications. Such a decision was not only made due to the unique characteristics of deep learning, but also due to feasibility given the limited available time to finish the thesis project. Nonetheless, we believe that our performance portability layer has the potential to be fully extended to other application domains as well. For example, any numerical operation that operates on tensors should be able to benefit from our methodology. Thus, foreseeable future work would be to explore the potential operations and application domains. Particularly, scientific simulations and computations that use tensors to represent large data volumes can be analyzed for performance portability.

6.2.2 Supporting other hardware

Within this work, we discussed the portability of ConvNet operations on various GPUs. However, efficient AI deployment becomes challenging when dealing with low-budget edge devices not necessarily equipped with GPUs. Thus, a potential future work would be to support additional hardware platforms, such as CPUs, TPUs, ASICs, and FPGAs. Nonetheless, the concepts that we introduced for efficient model search, compression, and deployment using efficient operations can also be used to target such devices. To achieve this goal, we need to design more lightweight ConvNets, compress them to a higher degree, and deploy the compressed models on CPUs instead. Such an effort enables affordable AI for all sorts of mass-produced devices, such as autonomous vehicles (e.g., cars, trains, buses, trucks), UAVs[1], and IoT[2] devices.

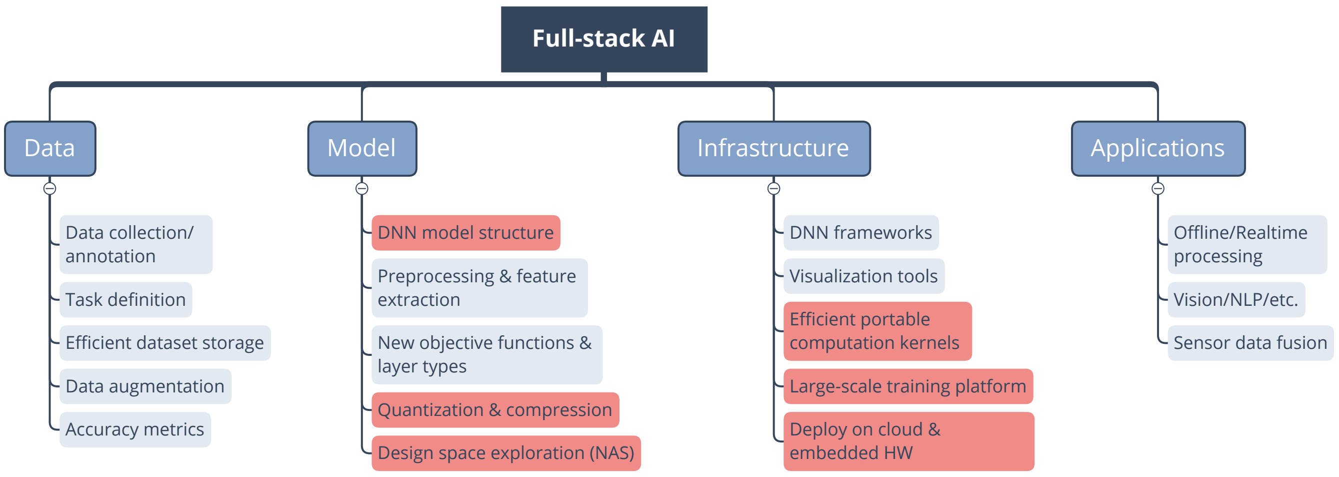
6.2.3 Full-stack AI

We consider our work presented in this dissertation as an initial step towards a full-stack AI platform. A full-stack AI platform consists of various components that work hand-in-hand to create a fully functional AI-based system. As shown in Figure 6.1, the main categories are data, model, infrastructure, and application. In this work, we primarily worked on the model and infrastructure parts and achieved noteworthy results. We can complement the overall workflow and create a full-fledged AI design pipeline by attaching other components and extending the current methods.

6.3 Final remarks

Given the rapid growth in the amount of data and connected smart devices, there will be an increasing demand for more efficient computing. We strongly believe that performance engineering of data-intensive applications, particularly deep-learning models, will remain a hot topic for the foreseeable future. Hence, it is critical to constantly design and invent new tools and methods to assist programmers in pinpointing performance bottlenecks and making their code run efficiently on various platforms. With new hardware generations being introduced every couple of months, an ecosystem of performance-engineering tools is vital to harness the available computing power, increase productivity, and reduce programming effort. In this work, we focused on the challenges associated with data-intensive applications, particularly in executing deep-learning models. We introduced three different techniques to address performance analysis, portability, and scalability. The first was a generic profiling method, whereas the last two contributions were focused on AI workloads. As a result, we managed to identify communication bottlenecks in various data-intensive applications. Moreover, we improved the portability and scaled up the performance of deep-learning

[1]Unmanned aerial vehicles
[2]Internet of things


Full-stack AI
Data
Data collection/ annotation
Task definition
Efficient dataset storage
Data augmentation
Accuracy metrics
Model
DNN model structure
Preprocessing & feature extraction
New objective functions & layer types
Quantization & compression
Design space exploration (NAS)
Infrastructure
DNN frameworks
Visualization tools
Efficient portable computation kernels
Large-scale training platform
Deploy on cloud & embedded HW
Applications
Offline/Realtime processing
Vision/NLP/etc.
Sensor data fusion
Figure 6.1: The components of a full-stack AI platform. The highlighted boxes are the components that we directly or indirectly targeted in this thesis.

workloads on various platforms. We look forward to pursuing our visions mentioned in Section 6.2 to further demonstrate the value of our approach.

Bibliography

[1] Rajkumar Buyya, Christian Vecchiola, and S. Thamarai Selvi. “Chapter 8 - Data-Intensive Computing: MapReduce Programming”. In: *Mastering Cloud Computing*. Ed. by Rajkumar Buyya, Christian Vecchiola, and S. Thamarai Selvi. Boston: Morgan Kaufmann, 2013, pp. 253–311.

[2] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[3] Apache Software Foundation. *Hadoop*. Version 0.20.2. Feb. 19, 2010.

[4] Dario Amodei and Danny Hernandez. *AI and compute*. July 1, 2021. URL: `https://openai.com/blog/ai-and-compute/`.

[5] Ross Girshick, Forrest Iandola, Trevor Darrell, and Jitendra Malik. “Deformable part models are convolutional neural networks”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 437–446.

[6] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully convolutional networks for semantic segmentation”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 3431–3440.

[7] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. “Large-scale video classification with convolutional neural networks”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 1725–1732.

[8] Andre Esteva et al. “Deep learning-enabled medical computer vision”. In: *NPJ digital medicine* 4.1 (2021), pp. 1–9.

[9] Nicolas Villain. *The role of artificial intelligence in medical imaging: from research to clinical routine*. 2019. URL: `https://www.aphc.info/wp-content/uploads/2019/01/pre232-villain-nicolas.pdf`.

[10] Matthew W. Moskewicz. “Implementing Efficient, Portable Computations for Machine Learning”. PhD thesis. University of California, Berkeley, 2017.

[11] Krste Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UC Berkeley, 2006.

[12] Torsten Hoefler, William Gropp, William Kramer, and Marc Snir. “Performance modeling for systematic performance tuning”. In: *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2011, pp. 1–12.

[13] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[14] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. “Simics: A full system simulation platform”. In: *Computer* 35.2 (2002), pp. 50–58.

[15] German Florez, Zhen Liu, Susan M Bridges, Anthony Skjellum, and Rayford B Vaughn. “Lightweight monitoring of MPI programs in real time”. In: *Concurrency and Computation: Practice and Experience* 17.13 (2005), pp. 1547–1578.

[16] Nick Barrow-Williams, Christian Fensch, and Simon Moore. “A communication characterisation of Splash-2 and Parsec”. In: *Proc. of International Symposium on Workload Characterization (IISWC)*. IEEE. 2009, pp. 86–97.

[17] Eduardo Henrique Molina da Cruz, Zanata Alves, Alexandre Carissimi, Philippe Olivier Alexandre Navaux, Christiane Pousa Ribeiro, and J Mehaut. “Using memory access traces to map threads and data on hierarchical multi-core platforms”. In: *Proc. of International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2011, pp. 551–558.

[18] Karl Fuerlinger, Nicholas J Wright, and David Skinner. “Effective performance measurement at petascale using IPM”. In: *Proc of. 16th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2010, pp. 373–380.

[19] Shoaib Kamil, John Shalf, Leonid Oliker, and David Skinner. “Understanding ultra-scale application communication requirements”. In: *International Symposium on Workload Characterization (IISWC)*. IEEE. 2005, pp. 178–187.

[20] Orianna DeMasi, Taghrid Samak, and David H Bailey. “Identifying HPC codes via performance logs and machine learning”. In: *Proc. of the first workshop on Changing landscapes in HPC security*. ACM. 2013.

[21] Sean Peisert. *Fingerprinting communication and computation on HPC machines*. Tech. rep. Lawrence Berkeley National Laboratory, 2010.

[22] Chao Ma, Yong Meng Teo, Verdi March, Naixue Xiong, Ioana Romelia Pop, Yan Xiang He, and Simon See. “An approach for matching communication patterns in parallel applications”. In: *Proc. of International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2009, pp. 1–12.

[23] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Proc. of International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2004, pp. 75–86.

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *ACM Sigplan notices* 40.6 (2005), pp. 190–200.

[25] Derek Bruening and Saman Amarasinghe. “Efficient, transparent, and comprehensive runtime code manipulation”. PhD thesis. Massachusetts Institute of Technology, Department of Electrical Engineering, 2004.

[26] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[27] Zhen Li, Rohit Atre, Zia Ul Huda, Ali Jannesari, and Felix Wolf. “Unveiling Parallelization Opportunities in Sequential Programs”. In: *Journal of Systems and Software* 117 (July 2016), pp. 282–295. DOI: `10.1016/j.jss.2016.03.045`.

[28] Ashay Rane and James Browne. “Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics”. In: *Proc. of the 21st international conference on parallel architectures and compilation techniques*. 2012, pp. 147–156.

[29] Ali Jannesari and Walter F Tichy. "On-the-fly race detection in multi-threaded programs". In: *Proc. of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*. ACM. 2008, p. 6.

[30] Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter F Tichy. "Helgrind+: An efficient dynamic race detector". In: *Proc. of International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2009, pp. 1–13.

[31] Ali Jannesari and Walter F. Tichy. "Library-independent data race detection". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* PP.99 (2013), pp. 1–11.

[32] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. "SD3: A scalable approach to dynamic data-dependence profiling". In: *Proc. of 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2010, pp. 535–546.

[33] Xu Liu and John M.Crummey. "Pinpointing data locality problems using data-centric analysis". In: *Proc. of 9th International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2011, pp. 171–180.

[34] Georges Da Costa and Jean-Marc Pierson. "Characterizing applications from power consumption: a case study for HPC benchmarks". In: *Information and Communication on Technology for the Fight against Global Warming*. Springer, 2011, pp. 10–17.

[35] Xu Liu and John M.Crummey. "A data-centric profiler for parallel programs". In: *Proc. of SC13, International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2013, p. 28.

[36] François Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, P-A Wacrenier, and Raymond Namyst. "Structuring the execution of OpenMP applications for multicore architectures". In: *Proc. of International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, pp. 1–10.

[37] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. "Collecting performance data with PAPI-C". In: *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[38] Eduardo HM Cruz, Matthias Diener, and Philippe Olivier Alexandre Navaux. "Using the translation lookaside buffer to map threads in parallel applications based on shared memory". In: *Proc. of the International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE. 2012, pp. 532–543.

[39] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315.

[40] R. Hornung and J. Keasler. *The RAJA Portability Layer: Overview and Status*. Lawrence Livermore National Lab (LLNL), 2014.

[41] Ronan Keryell, Ruyman Reyes, and Lee Howes. "Khronos SYCL for OpenCL: A Tutorial". In: *Proc. of the 3rd International Workshop on OpenCL*. IWOCL '15. Palo Alto, California: Association for Computing Machinery, 2015.

[42] Arya Mazaheri, Felix Wolf, and Ali Jannesari. "Unveiling Thread Communication Bottlenecks Using Hardware-Independent Metrics". In: *Proc. of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10.

[43] Arya Mazaheri, Johannes Schulte, Matthew Moskewicz, Felix Wolf, and Ali Jannesari. “Enhancing the programmability and performance portability of GPU tensor operations”. In: *Proc. of the 25th Euro-Par Conference, Göttingen, Germany*. Vol. 11725. Lecture Notes in Computer Science. Springer, Aug. 2019, pp. 213–226.

[44] Johannes Schulte. “Achieving Efficiency and Portability in Convolutional Neural Networks Using Vulkan API”. Bachelor’s thesis. Technical University of Darmstadt, Department of Computer Science, 2018.

[45] Arya Mazaheri, Tim Beringer, Matthew Moskewicz, Felix Wolf, and Ali Jannesari. “Accelerating Winograd Convolutions using Symbolic Computation and Meta-programming”. In: *Proc. of the 15th EuroSys Conference, Heraklion, Crete, Greece*. 40. ACM, Apr. 2020, pp. 1–14.

[46] Tim Beringer. “Decreasing the Computational Complexity of Convolutional Neural Networks with Winograd Convolution”. Bachelor’s thesis. Technical University of Darmstadt, Department of Computer Science, 2018.

[47] Sixing Yu, Arya Mazaheri, and Ali Jannesari. “Auto Graph Encoder-Decoder for Model Compression and Network Acceleration”. In: *Proc. of the International Conference on Computer Vision (ICCV), Montreal, Canada*. IEEE, Oct. 2021.

[48] Tim Beringer. “Adaptive White-box Scheduler for Deep Learning Training”. MA thesis. Technical University of Darmstadt, Department of Computer Science, 2021.

[49] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. “Parcae: a system for flexible parallel execution”. In: *ACM SIGPLAN Notices* 47.6 (2012), pp. 133–144.

[50] Ashay Rane and James Browne. “Performance optimization of data structures using memory access characterization”. In: *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE. 2011, pp. 570–574.

[51] Milind Kulkarni, Vijay Pai, and Derek Schuff. “Towards architecture independent metrics for multicore performance analysis”. In: *ACM SIGMETRICS Performance Evaluation Review* 38.3 (2011), pp. 10–14.

[52] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. “The PARSEC benchmark suite: Characterization and architectural implications”. In: *Proc. of International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM. 2008, pp. 72–81.

[53] Meng-Ju Wu and Donald Yeung. “Efficient reuse distance analysis of multicore scaling for loop-based parallel programs”. In: *ACM Transactions on Computer Systems (TOCS)* 31.1 (2013).

[54] Derek L Schuff, Benjamin S Parsons, and Vijay S Pai. “Multicore-aware reuse distance analysis”. In: *Proc. of International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2010, pp. 1–8.

[55] *NVIDIA Collective Communications Library (NCCL)*. 2021. URL: `https://developer.nvidia.com/nccl`.

[56] Matthias Diener, Eduardo HM Cruz, Marco AZ Alves, and Philippe OA Navaux. “Communication in shared memory: Concepts, definitions, and efficient detection”. In: *Proc. of 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE. 2016, pp. 151–158.

[57] Arya Mazaheri, Ali Jannesari, Abdolreza Mirzaei, and Felix Wolf. “Characterizing Loop-Level Communication Patterns in Shared Memory”. In: *Proc. of 44th International Conference on Parallel Processing (ICPP)*. IEEE. 2015, pp. 759–768.

[58] Chen Ding and Yutao Zhong. “Predicting whole-program locality through reuse distance analysis”. In: *ACM SIGPLAN Notices*. Vol. 38. 5. ACM. 2003, pp. 245–257.

[59] Kristof Beyls and Erik D’Hollander. “Reuse distance as a metric for cache behavior”. In: *IASTED Conference on Parallel and Distributed Computing and systems*. Vol. 14. 2001, pp. 350–360.

[60] Kristof Beyls and Erik H D’Hollander. “Generating cache hints for improved program efficiency”. In: *Journal of Systems Architecture* 51.4 (2005), pp. 223–250.

[61] Yunlian Jiang, Eddy Zhang, Kai Tian, and Xipeng Shen. “Is reuse distance applicable to data locality analysis on chip multiprocessors?” In: *Compiler Construction*. Springer. 2010, pp. 264–282.

[62] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. “Studying multicore processor scaling via reuse distance analysis”. In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM. 2013, pp. 499–510.

[63] Zhen Li, Ali Jannesari, and Felix Wolf. “An efficient data-dependence profiler for sequential and parallel programs”. In: *Proc. of 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. 2015.

[64] Sean Whalen, Sophie Engle, Sean Peisert, and Matt Bishop. “Network-theoretic classification of parallel computation patterns”. In: *International Journal of High Performance Computing Applications* (2012).

[65] Sean Whalen, Sean Peisert, and Matt Bishop. “Multiclass classification of distributed memory parallel computations”. In: *Pattern Recognition Letters* 34.3 (2013), pp. 322–329.

[66] Daniel Sanchez, Luke Yen, Mark D Hill, and Karthikeyan Sankaralingam. “Implementing signatures for transactional memory”. In: *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*. IEEE. 2007, pp. 123–133.

[67] *MurmurHash family of hash functions*. 2021. URL: https://github.com/aappleby/smhasher.

[68] Andrei Broder and Michael Mitzenmacher. “Network applications of bloom filters: A survey”. In: *Internet mathematics* 1.4 (2004), pp. 485–509.

[69] Matthias Diener, Eduardo HM Cruz, Marco AZ Alves, Mohammad S Alhakeem, Philippe OA Navaux, and Hans-Ulrich Heiß. “Locality and balance for communication-aware thread mapping in multicore systems”. In: *Proc. of the European Conference on Parallel Processing*. Springer. 2015, pp. 196–208.

[70] Christian Bienia, Sanjeev Kumar, and Kai Li. “PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors”. In: *International Symposium on Workload Characterization (IISWC)*. IEEE. 2008, pp. 47–56.

[71] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. “The SPLASH-2 programs: Characterization and methodological considerations”. In: *ACM SIGARCH Computer Architecture News*. Vol. 23. 2. ACM. 1995, pp. 24–36.

[72] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. “Rodinia: A benchmark suite for heterogeneous computing”. In: *Proc. of International Symposium on Workload Characterization (IISWC)*. IEEE. 2009, pp. 44–54.

[73] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. “Gprof: A call graph execution profiler”. In: *ACM Sigplan Notices*. Vol. 17. 6. ACM. 1982, pp. 120–126.

[74] David Eklov and Erik Hagersten. “StatStack: Efficient modeling of LRU caches”. In: *International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE. 2010, pp. 55–65.

[75] Ahmad Faraj and Xin Yuan. “Communication characteristics in the NAS parallel benchmarks”. In: *IASTED PDCS*. 2002, pp. 724–729.

[76] Ingyu Lee. “Characterizing communication patterns of NAS-MPI benchmark programs”. In: *IEEE Southeastcon 2009*. IEEE. 2009, pp. 158–163.

[77] Eduardo HM Cruz, Matthias Diener, Laércio L Pilla, and Philippe OA Navaux. “An efficient algorithm for communication-based task mapping”. In: *Proc. of Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2015, pp. 207–214.

[78] Matthias Diener, Eduardo HM Cruz, Laércio L Pilla, Fabrice Dupros, and Philippe OA Navaux. “Characterizing communication and page usage of parallel applications for thread and data mapping”. In: *Performance Evaluation* 88 (2015), pp. 18–36.

[79] Matthias Diener, Eduardo HM Cruz, and Philippe OA Navaux. “Locality vs. balance: Exploring data mapping policies on NUMA systems”. In: *Proc. of 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2015, pp. 9–16.

[80] Yutao Zhong, Xipeng Shen, and Chen Ding. “Program locality analysis using reuse distance”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31.6 (2009).

[81] Qingpeng Niu, James Dinan, Qingda Lu, and P Sadayappan. “PARDA: A fast parallel reuse distance analysis algorithm”. In: *26th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE. 2012, pp. 1284–1294.

[82] Derek L Schuff, Milind Kulkarni, and Vijay S Pai. “Accelerating multicore reuse distance analysis with sampling and parallelization”. In: *Proc. of International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2010, pp. 53–63.

[83] Andrej Karpathy. *Software 2.0*. URL: `https://karpathy.medium.com/software-2-0-a64152b37c35l`.

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

[85] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep residual learning for image recognition”. In: *Proc. of the IEEE conference on computer vision and pattern recognition (CVPR)*. 2016, pp. 770–778.

[86] Olga Russakovsky et al. “Imagenet large scale visual recognition challenge”. In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[87] Alex Krizhevsky and Geoffrey Hinton. *Learning multiple layers of features from tiny images*. Tech. rep. 0. Toronto, Ontario: University of Toronto, 2009.

[88] Matthijs Hollemans. *How fast is my model?* July 1, 2021. URL: `https://machinethink.net/blog/how-fast-is-my-model/`.

[89] Song Han, Huizi Mao, and J. William Dally. “Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding”. In: 2015.

[90] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. `https://d2l.ai`. 2020.

[91] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[92] Tianqi Chen et al. “TVM: An automated end-to-end optimizing compiler for deep learning”. In: *Proc. of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 578–594.

[93] Martín Abadi et al. “TensorFlow: A system for large-scale machine learning”. In: *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Vol. 16. 2016, pp. 265–283.

[94] Nadav Rotem et al. *Glow: Graph lowering compiler techniques for neural networks*. 2018. eprint: `1805.00907`.

[95] Matthew W Moskewicz, Ali Jannesari, and Kurt Keutzer. “A Metaprogramming and Autotuning Framework for Deploying Deep Learning Applications”. In: *arXiv preprint arXiv:1611.06945* (2016).

[96] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. “A survey on neural architecture search”. In: *arXiv preprint arXiv:1905.01392* (2019).

[97] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. “A comprehensive survey of neural architecture search: Challenges and solutions”. In: *ACM Computing Surveys (CSUR)* 54.4 (2021), pp. 1–34.

[98] Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. “Neural architecture search: A survey.” In: *Journal of Machine Learning Research (JMLR)* 20.55 (2019), pp. 1–21.

[99] Andrei L Toom. “The complexity of a scheme of functional elements realizing the multiplication of integers”. In: *Soviet Mathematics Doklady*. Vol. 3. 4. 1963, pp. 714–716.

[100] Stephen A. Cook and Stål O. Aanderaa. “On the minimum computation time of functions”. In: *Transactions of the American Mathematical Society* 142 (1969), pp. 291–314.

[101] Shmuel Winograd. *Arithmetic complexity of computations*. Vol. 33. Siam, 1980.

[102] Andrew Lavin and Scott Gray. “Fast algorithms for convolutional neural networks”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4013–4021.

[103] Barbara Barabasz, Andrew Anderson, and David Gregg. *Error analysis and improving the accuracy of Winograd convolution for deep neural networks*. 2018. eprint: `1803.10986`.

[104] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. “Optimizing n-dimensional, Winograd-based convolution for manycore CPUs”. In: *Proc. of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM. 2018, pp. 109–123.

[105] Partha Maji, Andrew Mundy, Ganesh Dasika, Jesse G. Beu, Matthew Mattina, and Robert Mullins. *Efficient Winograd or Cook-Toom convolution kernel implementation on widely used mobile CPUs*. 2019. eprint: `1903.01521`.

[106] Bichen Wu et al. “Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search”. In: *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 10734–10742.

[107] Hanxiao Liu, Karen Simonyan, and Yiming Yang. “Darts: Differentiable architecture search”. In: *arXiv preprint arXiv:1806.09055* (2018).

[108] Han Cai, Ligeng Zhu, and Song Han. “ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware”. In: *Proc. of International Conference on Learning Representations (ICLR)*. 2019.

[109] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. “Once for All: Train One Network and Specialize it for Efficient Deployment”. In: *International Conference on Learning Representations*. 2020.

[110] Alexander Zai and Brandon Brown. *Deep reinforcement learning in action*. Manning Publications, 2020.

[111] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. "A comprehensive survey on graph neural networks". In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24.

[112] Karen Simonyan and Andrew Zisserman. *Very deep convolutional networks for large-scale image recognition*. 2014. eprint: `1409.1556`.

[113] Sergey Ioffe and Christian Szegedy. *Batch normalization: Accelerating deep network training by reducing internal covariate shift*. 2015. eprint: `1502.03167`.

[114] Michael Mathieu, Mikael Henaff, and Yann LeCun. *Fast training of convolutional networks through FFTs*. 2013. eprint: `1312.5851`.

[115] Matthew W Moskewicz, Ali Jannesari, and Kurt Keutzer. "Boda: A Holistic Approach for Implementing Neural Network Computations". In: *Proc. of International Conference on Computing Frontiers*. CF'17. Siena, Italy: ACM, 2017, pp. 53–62.

[116] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming". In: *Parallel Computing* 38.8 (2012), pp. 391–407.

[117] The Khronos Group. *Khronos Vulkan Registry*. 2019. URL: `https://www.khronos.org/registry/vulkan` (visited on 02/15/2019).

[118] The Khronos Group. *Khronos SPIR-V Registry*. 2019. URL: `https://www.khronos.org/registry/spir-v` (visited on 02/15/2019).

[119] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. "A comprehensive performance comparison of CUDA and OpenCL". In: *Proc. of International Conference on Parallel Processing (ICPP)*. IEEE. 2011, pp. 216–225.

[120] Junghyun Kim, Thanh Tuan Dao, Jaehoon Jung, Jinyoung Joo, and Jaejin Lee. "Bridging OpenCL and CUDA: a comparative analysis and translation". In: *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis*. SC'15. ACM. 2015, pp. 1–12.

[121] Adrian Sampson. "Let's Fix OpenGL". In: *Leibniz International Proceedings in Informatics*. Vol. 71. LIPIcs '17. Schloss Dagstuhl, Leibniz-Zentrum füer Informatik. 2017.

[122] Nadjib Mammeri and Ben Juurlink. "VComputeBench: A Vulkan Benchmark Suite for GPGPU on Mobile and Embedded GPUs". In: *Proc. of International Symposium on Workload Characterization*. IISWC'18. IEEE, 2018, pp. 25–35.

[123] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. "cuDNN: Efficient primitives for deep learning". In: *arXiv preprint arXiv:1410.0759* (2014).

[124] Andrew Lavin. *maxDNN: An efficient convolution kernel for deep learning with Maxwell GPUs*. 2015. eprint: `1501.06633`.

[125] Kevin Vincent, Kevin Stephano, Michael Frumkin, Boris Ginsburg, and Julien Demouth. "On improving the numerical stability of Winograd convolutions". In: *Proc. of the International Conference on Learning Representations (ICLR)*. 2017.

[126] Cedric Nugteren. "CLBlast: A tuned OpenCL BLAS library". In: *Proc. of the International Workshop on OpenCL (IWOCL)*. ACM. 2018, p. 5.

[127] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. “Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption”. In: *Proc. of Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. ACM. 2017, pp. 1–6.

[128] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. *Training deep neural networks with low precision multiplications*. 2014. eprint: `1412.7024`.

[129] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. “Deep learning with limited numerical precision”. In: *Proc. of the International Conference on Machine Learning*. 2015, pp. 1737–1746.

[130] Min Lin, Qiang Chen, and Shuicheng Yan. “Network in network”. In: *arXiv preprint arXiv:1312.4400* (2013).

[131] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. “Going deeper with convolutions”. In: *Proc. of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[132] Kamran Karimi, Neil G Dickson, and Firas Hamze. “A performance comparison of CUDA and OpenCL”. In: *arXiv preprint arXiv:1005.2581* (2010).

[133] Rafael Sachetto Oliveira, Bernardo Martins Rocha, Ronan Mendonça Amorim, Fernando Otaviano Campos, Wagner Meira, Elson Magalhães Toledo, and Rodrigo Weber dos Santos. “Comparing CUDA, OpenCL and OpenGL Implementations of the Cardiac Monodomain Equations”. In: *Proc. of 9th International Conference on Parallel Processing and Applied Mathematics*. PPAM’11. Torun, Poland: Springer-Verlag, 2011, pp. 111–120.

[134] Ching-Lung Su, Po-Yu Chen, Chun-Chieh Lan, Long-Sheng Huang, and Kuo-Hsuan Wu. “Overview and comparison of OpenCL and CUDA technology for GPGPU”. In: *Proc. of Asia Pacific Conference on Circuits and Systems*. APCCAS’12. IEEE. 2012, pp. 448–451.

[135] Hercules Cardoso Da Silva, Flávia Pisani, and Edson Borin. “A Comparative Study of SYCL, OpenCL, and OpenMP”. In: *Proc. of International Symposium on Computer Architecture and High-Performance Computing Workshops*. SBAC-PADW’16. IEEE. 2016, pp. 61–66.

[136] Partha Maji and Robert Mullins. “On the reduction of computational complexity of deep convolutional neural networks”. In: vol. 20. 4. Multidisciplinary Digital Publishing Institute, 2018, p. 305.

[137] Jason Cong and Bingjun Xiao. “Minimizing computation in convolutional neural networks”. In: *Proc. of the International Conference on Artificial Neural Networks*. Springer. 2014, pp. 281–290.

[138] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. *Fast convolutional nets with fbfft: A GPU performance evaluation*. 2014. eprint: `1412.7580`.

[139] Aleksandar Zlateski, Zhen Jia, Kai Li, and Fredo Durand. “The Anatomy of Efficient FFT and Winograd Convolutions on Modern CPUs”. In: *Proc. of the ACM International Conference on Supercomputing*. ICS. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 414–424.

[140] David Budden, Alexander Matveev, Shibani Santurkar, Shraman Ray Chaudhuri, and Nir Shavit. “Deep tensor convolution on multicores”. In: *Proc. of the 34th International Conference on Machine Learning*. 2017, pp. 615–624.

[141] Athanasios Xygkis, Lazaros Papadopoulos, David Moloney, Dimitrios Soudris, and Sofiane Yous. “Efficient Winograd-based convolution kernel implementation on edge devices”. In: *Proc. of the 55th Annual Design Automation Conference*. ACM. 2018, p. 136.

[142] Hyunsun Park, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. “Zero and data reuse-aware fast convolution for deep neural networks on GPU”. In: *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE. 2016, pp. 1–10.

[143] Sheng Li, Jongsoo Park, and Ping Tak Peter Tang. *Enabling sparse Winograd convolution by native pruning*. 2017. eprint: `1702.08597`.

[144] Xingyu Liu, Jeff Pool, Song Han, and William J Dally. *Efficient sparse-Winograd convolutional neural networks*. 2018. eprint: `1802.06367`.

[145] Intel. *PlaidML*. 2019. URL: `https://www.intel.ai/plaidml` (visited on 04/20/2021).

[146] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zach DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. “Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions”. In: *arXiv preprint arXiv:1802.04730* (2018).

[147] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. “DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications”. In: *Proc. of 15th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys’17. ACM, 2017, pp. 82–95.

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

[149] Simon Moser and Oscar Nierstrasz. “The effect of object-oriented frameworks on developer productivity”. In: *Computer* 29.9 (1996), pp. 45–51.

[150] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *CoRR* abs/1704.04861 (2017). arXiv: `1704.04861`.

[151] Mark Sandler, G. Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: 2018, pp. 4510–4520.

[152] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. “Shufflenet: An extremely efficient convolutional neural network for mobile devices”. In: *Proc. of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 6848–6856.

[153] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. “Shufflenet v2: Practical guidelines for efficient cnn architecture design”. In: *Proc. of the European conference on computer vision (ECCV)*. 2018, pp. 116–131.

[154] Sachin Mehta, Hannaneh Hajishirzi, and Mohammad Rastegari. “DiCENet: Dimension-wise convolutions for efficient networks”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020).

[155] Gao Huang, Shichen Liu, Laurens Van der Maaten, and Kilian Q Weinberger. “Condensenet: An efficient densenet using learned group convolutions”. In: *Proc. of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2752–2761.

[156] Microsoft. *NNI (Neural Network Intelligence)*. July 1, 2021. URL: `https://github.com/microsoft/nni`.

[157] Wencong Xiao et al. “Gandiva: Introspective cluster scheduling for deep learning”. In: *Proc. of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 595–610.

[158] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. “Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning”. In: *Proc. of 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 2021, pp. 1–18.

[159] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. “Optimus: an efficient dynamic resource scheduler for deep learning clusters”. In: *Proc. of the Thirteenth EuroSys Conference*. 2018, pp. 1–14.

[160] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, Chen Meng, and Wei Lin. “Dl2: A deep learning-driven scheduler for deep learning clusters”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.8 (2021), pp. 1947–1960.

[161] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. “Firecaffe: near-linear acceleration of deep neural network training on compute clusters”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2592–2600.

[162] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. “An empirical model of large-batch training”. In: *arXiv preprint arXiv:1812.06162* (2018).

[163] Morris A. Jette, Andy B. Yoo, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *Proc. of Job Scheduling Strategies for Parallel Processing (JSSPP)*. Lecture Notes in Computer Science. Springer-Verlag, 2002, pp. 44–60.

[164] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. “Mesos: A platform for fine-grained resource sharing in the data center.” In: *Proc. of Symposium on Networked Systems Design And Implementation (NSDI)*. Vol. 11. 2011. 2011, pp. 22–22.

[165] Vinod Kumar Vavilapalli et al. “Apache hadoop yarn: Yet another resource negotiator”. In: *Proc. of the 4th annual Symposium on Cloud Computing*. 2013, pp. 1–16.

[166] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. “Large-scale cluster management at Google with Borg”. In: *Proc. of the Tenth European Conference on Computer Systems*. 2015, pp. 1–17.

[167] Ruben Mayer and Hans-Arno Jacobsen. “Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools”. In: *ACM Computing Surveys (CSUR)* 53.1 (2020), pp. 1–37.

[168] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. “Kungfu: Making training in distributed machine learning adaptive”. In: *Proc. of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2020, pp. 937–954.

[169] Alexander Sergeev and Mike Del Balso. “Horovod: fast and easy distributed deep learning in TensorFlow”. In: *arXiv preprint arXiv:1802.05799* (2018).

[170] Tyler B Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. *AdaScale SGD: A Scale-Invariant Algorithm for Distributed Training*. 2020.

[171] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *Proc. of the International Conference on Learning Representations (ICLR)*. 2017.

[172] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. “Modeling Relational Data with Graph Convolutional Networks”. In: *The Semantic Web*. Ed. by Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam. Cham: Springer International Publishing, 2018, pp. 593–607. ISBN: 978-3-319-93417-4.

[173] Sheng Kai Tai, Richard Socher, and D. Christopher Manning. “Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks”. In: *International Workshop on the ACL2 Theorem Prover and Its Applications* (2015).

[174] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: `1707.06347 [cs.LG]`.

[175] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. “Continuous control with deep reinforcement learning”. In: *arXiv preprint arXiv:1509.02971* (2015).

[176] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. “AMC: Automl for model compression and acceleration on mobile devices”. In: *Proc. of the European Conference on Computer Vision (ECCV)*. 2018, pp. 784–800.

[177] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *international conference on learning representations* (2015).

[178] Yihui He, Xiangyu Zhang, and Jian Sun. “Channel pruning for accelerating very deep neural networks”. In: *Proc. of the IEEE International Conference on Computer Vision*. 2017, pp. 1389–1397.

[179] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. “Pruning Filters for Efficient ConvNets”. In: *CoRR* abs/1608.08710 (2016). arXiv: `1608.08710`.

[180] Huan Wang, Qiming Zhang, Yuehai Wang, and Roland Hu. “Structured Probabilistic Pruning for Deep Convolutional Neural Network Acceleration”. In: *British Machine Vision Conference* (2017).

[181] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. “Runtime neural pruning”. In: *Proc. of the Advances in Neural Information Processing Systems*. 2017, pp. 2181–2191.

[182] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. “MorphNet: Fast & Simple Resource-Constrained Structure Learning of Deep Networks”. en. In: *Proc. of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Salt Lake City, UT: IEEE, June 2018, pp. 1586–1595.

[183] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. “Learning Structured Sparsity in Deep Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 29. Curran Associates, Inc., 2016.

[184] Xuefei Ning, Tianchen Zhao, Wenshuo Li, Peng Lei, Yu Wang, and Huazhong Yang. “DSA: More Efficient Budgeted Pruning via Differentiable Sparsity Allocation”. en. In: *Proc. of European Conference on Computer Vision (ECCV)*. Vol. 12348. Series Title: Lecture Notes in Computer Science. Springer International Publishing, 2020, pp. 592–607.

[185] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. “Rethinking the Value of Network Pruning”. In: *International Conference on Learning Representations (ICLR)* (2019).

[186] A Krizhevsky and G Hinton. “Learning multiple layers of features from tiny images”. In: (2009).

[187] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[188] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. “Algorithms for hyper-parameter optimization”. In: *Advances in neural information processing systems* 24 (2011).

[189] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. “Efficient and Robust Automated Machine Learning”. In: *Proc. of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS’15. Montreal, Canada: MIT Press, 2015, pp. 2755–2763.

[190] Aditya Devarakonda, Maxim Naumov, and Michael Garland. “Adabatch: Adaptive batch sizes for training deep neural networks”. In: *arXiv preprint arXiv:1712.02029* (2017).

[191] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. “Don’t decay the learning rate, increase the batch size”. In: *arXiv preprint arXiv:1711.00489* (2017).

[192] Lukas Balles, Javier Romero, and Philipp Hennig. “Coupling adaptive batch sizes with learning rates”. In: *arXiv preprint arXiv:1612.05086* (2016).

[193] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. “Tiresias: A GPU cluster manager for distributed deep learning”. In: *Proc. of 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2019, pp. 485–500.

[194] Yun Seong Lee Lee, Markus Weimer, Youngseok Yang, and Gyeong-In Yu. “Dolphin: Runtime optimization for distributed machine learning”. In: *Proc. of ICML ML Systems Workshop*. 2016.

[195] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. “Slaq: quality-driven scheduling for distributed machine learning”. In: *Proc. of the 2017 Symposium on Cloud Computing*. 2017, pp. 390–404.

[196] Peng Sun, Yonggang Wen, Nguyen Binh Duong Ta, and Shengen Yan. “Towards distributed machine learning in shared clusters: A dynamically-partitioned approach”. In: *IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE. 2017, pp. 1–6.

[197] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. “Online job scheduling in distributed machine learning clusters”. In: *IEEE Conference on Computer Communications*. IEEE. 2018, pp. 495–503.

[198] Mark Sandler, G. Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *CVPR* (2018), pp. 4510–4520.

[199] Antonio Polino, Razvan Pascanu, and Dan Alistarh. “Model compression via distillation and quantization”. In: *ICLR* (2018).

[200] Wonpyo Park, Dongju Kim, Yan Lu, and Minsu Cho. “Relational knowledge distillation”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 3967–3976.

[201] E. Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. “Distilling the Knowledge in a Neural Network”. In: *CoRR* (2015).

[202] Xu Lan, Xiatian Zhu, and Shaogang Gong. “Knowledge Distillation by On-the-Fly Native Ensemble”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 7527–7537.

[203] Sridhar Swaminathan, Deepak Garg, Rajkumar Kannan, and Frederic Andres. “Sparse Low Rank Factorization for Deep Neural Network Compression”. In: *Neurocomputing* (2020), pp. 185–196.

[204] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran. “Low-rank matrix factorization for Deep Neural Network training with high-dimensional output targets”. In: *Proc. of IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013, pp. 6655–6659.

[205] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. “Speeding up convolutional neural networks with low rank expansions”. In: *arXiv preprint arXiv:1405.3866* (2014).

[206] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. “Accelerating very deep convolutional networks for classification and detection”. In: *IEEE transactions on pattern analysis and machine intelligence* 38.10 (2015), pp. 1943–1955.

[207] J. Wang, H. Bai, J. Wu, and J. Cheng. “Bayesian Automatic Model Compression”. In: *IEEE Journal of Selected Topics in Signal Processing* 14.4 (2020), pp. 727–736. DOI: `10.1109/JSTSP.2020.2977090`.

[208] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. “Quantization and training of neural networks for efficient integer-arithmetic-only inference”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2704–2713.

[209] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. “Xnor-net: Imagenet classification using binary convolutional neural networks”. In: *Proc. of European Conference on Computer Vision*. Springer. 2016, pp. 525–542.

[210] Babak Hassibi and G. David Stork. “Second Order Derivatives for Network Pruning: Optimal Brain Surgeon”. In: *NIPS* (1992), pp. 164–171.

[211] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. “Importance estimation for neural network pruning”. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 11264–11272.

[212] Houxiang Ji, Linghao Song, Li Jiang, Hai Halen Li, and Yiran Chen. “ReCom: An efficient resistive accelerator for compressed deep neural networks”. In: *Proc of. Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 237–240.

[213] Song Han et al. “ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA”. In: *Proc. of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2017), pp. 75–84.

[214] Xia Xiao, Zigeng Wang, and Sanguthevar Rajasekaran. “AutoPrun: Automatic Network Pruning by Regularizing Auxiliary Parameters”. In: *Advances in Neural Information Processing Systems (NIPS)* (2019), pp. 13681–13691.

[215] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. “AutoCompress: An Automatic DNN Structured Pruning Framework for Ultra-High Compression Rates.” In: *Proc. of Artificial Intelligence Conference (AAAI)*. 2020, pp. 4876–4883.

[216] Salem Alqahtani and Murat Demirbas. “Performance analysis and comparison of distributed machine learning systems”. In: *arXiv preprint arXiv:1909.02061* (2019).