THESIS TO OBTAIN THE DEGREE OF DOCTOR
OF THE UNIVERSITY OF MONTPELLIER

In Automatic and Microelectronic Systems (SyAM)

Doctoral School: Information, Structures, Systems (I2S)

Research Unit: LIRMM

Advanced Profiling Techniques For Evaluating GPU Computing Efficiency Executing ML Applications

Presented by Paul DELESTRAC
September 17, 2024

Under the supervision of Lionel TORRES & David NOVO

Thesis Committee:

Olivier SENTIEYS, Professor, ENSSAT, University of Rennes, Inria/Irisa	Reviewer
Smail NIAR, Professor, LAMIH, University Polytechnique Hauts-de-France, CNRS	Reviewer
Michel ROBERT, Professor, LIRMM, University of Montpellier, CNRS	Examiner, President
Virginie FRESSE, Associate Professor, Jean Monnet University of Saint-Etienne, CNRS	Examiner
Francky CATTHOOR, Professor, imec, KU Leuven	Examiner
Lionel TORRES, Professor, LIRMM, University of Montpellier, CNRS	Thesis Director
David NOVO, Reasearch Scientist, LIRMM, University of Montpellier, CNRS	Thesis Co-Supervisor



Abstract

The rising complexity of Artificial Intelligence (AI) applications significantly increases demand for computing power to execute and train Machine Learning (ML) models, thus boosting the energy consumption of data centers. In response to this demand, Graphics Processing Units (GPUs) have been enhanced by application-specific hardware developments. For example, NVIDIA (the GPU market leader) added tensor cores to its GPUs in 2017, aiming to accelerate matrix multiplications, which are ubiquitous operations in ML models. As a result, the GPU architecture has become the default when it comes to training and running these ML models.

Building efficient ML computing systems relies on a deep understanding of the limits of the existing tightly-coupled hardware/software paradigm. Hence, hardware and software developers rely on profiling tools. These tools can help evaluate the performance of such systems by gathering metrics during the application execution and exposing useful insights to the developer, guiding the optimization decisions. However, the high abstraction of ML frameworks and the closed-source design of state-of-the-art GPU architectures obscure the execution process and/or limit the analysis scope that profiling tools can target. This makes performance evaluation tedious as developers often use these ML frameworks as black boxes and make decisions on profiling reports that can be misleading, ultimately missing important bottlenecks.

The main goal of this thesis is to help bridge the existing knowledge gap on the runtime execution of GPU-accelerated ML workloads and provide new profiling methodologies to evaluate performance and energy bottlenecks of such systems. Existing profiling solutions are limited in three ways. First, ML frameworks are designed to assist the development of ML models but are often used as a black box by the developers. In addition, the profiling tools that the frameworks provide do not give insights into the inner mechanisms composing the ML framework and decisions they take during runtime execution. Second, while these profiling tools provide high-level metrics (i.e., application-related metrics) on the GPU device execution (e.g., execution time, memory utilization), these metrics can be misleading and overestimate the utilization of the GPU resources. Lower-level profiling tools provide access to hardware performance

counters, providing architecture-related metrics (e.g., counting hardware events such as instructions issued, cache hits/misses), and insights on how to optimize GPU kernels. However, these tools cannot capture the efficiency of host/device interactions occurring at a higher level. Due to these inherent limitations of scope of both types of tools, the developer has to develop an expertise in using both profiling tools to find meaningful insights. Finally, when evaluating energy bottlenecks, the mentioned profiling tools cannot provide a detailed breakdown of the energy consumed by modern GPUs during ML training. To tackle these shortcomings, this thesis makes three key contributions organized as a top-down analysis of GPU-accelerated ML workloads.

First, we analyze ML frameworks' runtime execution on a CPU-GPU tandem. We propose a new profiling methodology that leverages data from an ML framework's profiler. We use this methodology to provide new insights into the runtime execution of inference, for three ML models. Our results show that GPU kernels' execution must be long enough to hide the runtime overhead of the ML framework, increasing GPU utilization. However, this strive for longer kernel execution leads to the use of bigger batches of data, seemingly pushing the need for more GPU memory.

Second, we analyze the utilization of GPU resources when performing ML training. We propose a new profiling methodology combining the use of multiple profilers to provide new insights into the utilization of the GPU's inner components. Our experiments, on two modern GPUs, suggest that bigger GPU memory helps enhance throughput and GPU utilization. However, our results also suggest that a plateau has been reached, eliminating the push for bigger batches. Furthermore, we observe that the fastest GPU cores (tensor cores) are idle most of the time, and the tested workloads are now limited by kernels that do not use these cores. Thus, our results suggest that the current GPU paradigm is reaching a saturation point.

Finally, we analyze the energy consumption of GPUs during ML training. We propose an energy model and calibration methodology that uses microbenchmarks to provide a breakdown of the GPU energy consumption. We implement and validate this approach with a modern NVIDIA GPU. Our results suggest that data movement is responsible for most of the energy consumption (up to 84% of the dynamic energy consumption of the GPU). This further motivates the push for newer architectures, optimizing memory accesses (e.g., processing in/near memory, vectorized architectures).

This thesis provides a comprehensive analysis of the performance and energy bottlenecks of GPU-accelerated ML workloads. We believe our contributions uncover some of the limitations of current GPU architectures and motivate the need for more advanced profiling techniques to design more efficient ML accelerators. We hope that our work will inspire future research in this direction.

Résumé de la thèse

L'augmentation en complexité des applications d'Intelligence Artificielle (IA) entraîne une demande accrue en puissance de calcul et en énergie pour entraîner et exécuter des modèles d'apprentissage automatique (ML). Pour répondre à cette demande, les processeurs graphiques (GPUs) ont reçu des amélioration matérielles visant spécifiquement les applications d'IA. Par exemple, NVIDIA (le leader du marché) a ajouté à ses GPUs des cœurs dédiés à accélérer les multiplications de matrices, qui sont dominantes dans la construction des modèles ML. En conséquence, les GPUs sont devenus l'architecture de prédilection pour cette catégorie d'applications.

Concevoir des systèmes plus efficients pour l'IA n'est possible qu'avec une connaissance approfondie des limites des systèmes existants, où matériel et logiciel sont étroitement couplés. Ainsi, développer ces systèmes repose sur l'utilisation d'outils de caractérisation qui peuvent assister à l'évaluation de performance. Ces outils mesurent d'importantes métriques durant l'exécution de l'application dans le but de présenter des rapports aux développeurs, guidant leurs décisions pour optimiser le système. Mais l'abstraction des plateformes d'IA et la nature fermée des architectures GPU modernes masquent le processus d'exécution et/ou limitent les capacités d'analyse des outils de caractérisation. Cela rend les performances du système difficiles à évaluer car les développeurs utilisent souvent ces plateformes d'IA comme des boites noires et prennent des décisions basées sur des rapports d'analyses qui peuvent induire en erreur sur les réels facteurs limitants.

L'objectif de cette thèse est caractériser les facteurs limitant la performance et augmentant la consommation énergétique des tâches d'IA exécutées avec des GPUs modernes. Cette thèse adresse trois limitations majeures des outils existants. Premièrement, les outils de caractérisation proposés par les plateformes de développement d'IA sont conçus pour aider les développeurs de modèles ML, mais ne donnent pas d'informations sur la charge additionnelle que représente l'exécution de ces plateformes. Deuxièmement, les outils de caractérisation proposés par les fabricants de GPUs permettent l'accès à des compteurs de performance, mais qui ne permettent pas d'estimer

l'efficacité des interactions entre le GPU et l'unité centrale (CPU). Enfin, pour caractériser la consommation énergétique des GPUs lors de l'entraînement d'IA, ces outils ne permettent pas d'obtenir une décomposition détaillée. Pour adresser ces limitations, cette thèse propose trois contributions.

Premièrement, nous analysons l'exécution des plateformes d'IA sur un couple CPU-GPU. Nous proposons une nouvelle méthodologie de caractérisation réutilisant les données fournies par des outils existants. Cette méthodologie permet d'extraire de nouvelles informations quant à l'exécution de modèles d'IA. Nous étudions l'exécution de trois modèles d'IA et nos résultats montrent que l'exécution des opérations destinées au GPU doit être suffisamment longue pour masquer le temps d'exécution de la plateforme d'IA, augmentant l'utilisation GPU. Pour autant, cette incitation à utiliser des opérations plus longues conduit à l'utilisation de lots de données plus conséquents, augmentant la demande en mémoire GPU.

Deuxièmement, nous analysons l'utilisation des ressources internes au GPU lors de l'entraînement. Nous proposons une nouvelle méthodologie de caractérisation combinant les outils proposés par les fabricants de GPUs et par les plateformes d'IA. Nos résultats suggèrent qu'un plafond de performance a été atteint, annulant les bénéfices à utiliser des lots de données plus larges pour l'entraînement. Nous observons que les cœurs les plus performants du GPU (tensor cores) restent inactifs durant la majorité du temps d'entraînement, limité par les opérations qui n'utilisent pas ces cœurs. Nos résultats suggèrent que les architectures GPU modernes ont atteint un point de saturation.

Enfin, nous analysons la consommation énergétique des GPUs lors de l'entraînement. Nous proposons une méthodologie basée sur l'utilisation de microprogrammes afin d'obtenir une décomposition de la consommation énergétique. Nos résultats suggèrent que les transferts de données sont responsables pour la majorité de la consommation énergétique dynamique du GPU (jusqu'à 84%). Ces résultats soutiennent la tendance de recherche pour des architectures cherchant optimiser les transferts de données (e.g., traitement en mémoire ou à proximité, architectures vectorielles).

Cette thèse propose une analyse approfondie des limites de performance et de la consommation énergétique des tâches d'IA exécutées à l'aide de GPUs modernes. Nous espérons que ce travail inspirera de futures recherches dans cette direction, pour concevoir des accélérateurs d'IA plus efficients.

Acknowledgements

Before diving into the core content of this thesis, I would like to thank many people without whom this PhD manuscript wouldn't exist. I believe many of them do not realize the extent of their contribution.

First, I would like to thank my thesis director Lionel Torres. His unbending support through the years helped me approach the thesis from a bird's eye view. Lionel also motivated the addition of energy consumption as a key metric for my research very early on. This led to one of the most important contributions of this thesis (Chapter 5).

David Novo, my PhD thesis co-supervisor, taught me everything I know about research thanks to his systematic, persistent, and sincere guidance over the years. I look up to his consistency and clearheadedness and will work hard to match them someday. Instead of telling me where to go, David taught me how to navigate this vast research ocean. For this, I will be forever grateful.

David also introduced me to imec and Franchy Catthoor. I would like to thank Francky for sharing his extensive experience with me and for his proactive feedback on this thesis work. During my time at imec, I also met a team full of great people. I thank Simei, Debjyoti, and Diksha for their direct participation in this research, but I also thank everyone from the imec team for their warm welcome.

During this PhD, I was a teaching assistant at Polytech Montpellier, the same school where I first studied computer science. I would like to thank Laurent, Eric, Pascal, Guy, and all the colleagues from Polytech for passing down their passion for teaching to me. Many thanks to the interns who helped with my research: Imane, Swan, and Pablo.

During this four-year journey at LIRMM, I spent a lot of time with many other PhD students. From the former colleagues: Theo, Loïc, Quentin, and Geoffrey; the newest team members: Bruno, and Aymen; and all the friends spending time with me at the lab: Luis, Soraya, Rafael, Xhesila, Kamilia, Lila, Mohamed, and many others. The moments of lightness we shared helped me bear the heaviness of the PhD process.

I would be remiss if I did not acknowledge the multifaceted support of Jonathan Miquel in this thesis. Jonathan has been one of the most hard-working colleagues I have ever met and working with him on Chapter 5 was one of my best research experiences. Jonathan has also been a close friend of mine for nearly eight years, but who's counting? Together with Rodrigue, Thibault, and Sébastien, we form a strong crew that distance cannot split. Thanks to Christina for being my friend for so many years and finding typos in my manuscript at the last minute, before sending it in for review. I also want to thank Colin for being a precious friend, despite us rarely seeing each other. I would like to express my gratitude to my beloved girlfriend, Léa, for her love and support. Relationships like these have been a major motivation for me to go through with this thesis, always with a smile on my face. And when I felt lonely, I could always count on them to cheer me up.

Finally, I am grateful for the love and support from my family over the years. Many thanks to my parents and my dear sister, Marie, for their understanding. I haven't been as available as I would've liked, but they never held it against me.

This thesis only exists thanks to all of you.

Contents

List of Figures **xiii**

List of Tables **xv**

List of Listings **xv**

List of Acronyms **xvii**

1 Introduction **1**
1.1 Context . . . 2
1.1.1 Machine Learning acceleration using GPUs . . . 2
1.1.2 Profiling performance and energy of Machine Learning (ML) workloads on Graphics Processing Units (GPUs) . . . 4
1.1.3 Related works . . . 5
1.2 Contributions . . . 6
1.2.1 Analysis of ML frameworks eager runtime execution . . . 6
1.2.2 Multi-level analysis of GPU utilization in ML training . . . 7
1.2.3 Energy evaluation of data movement in the GPU architecture . . . 8
1.3 Thesis outline . . . 8

2 Background **11**
2.1 ML inference and training execution flow . . . 12
2.1.1 ML models . . . 12
2.1.2 ML inference and training . . . 13
2.1.3 ML frameworks . . . 13
2.2 GPU architecture and programming model . . . 15
2.2.1 Programming model for NVIDIA GPUs: CUDA . . . 16
2.2.2 GPU execution model . . . 18
2.2.3 GPU memory hierarchy . . . 21
2.3 ML and GPU profiling tools . . . 22
2.3.1 ML profiling . . . 22
2.3.2 GPU kernels profiling . . . 23

3 A Deep Dive into Modern ML Frameworks 25
3.1 Context and motivation . . . 26
3.2 TensorFlow eager execution . . . 27
3.2.1 TensorFlow operations . . . 29
3.2.2 Execution context . . . 29
3.2.3 Enqueuing: Python host program . . . 29
3.2.4 Dequeuing: Executor thread . . . 31
3.2.5 Illustration example . . . 32
3.3 Analysis . . . 34
3.3.1 TensorFlow profiling capabilities . . . 34
3.3.2 Time spent in kernel execution . . . 35
3.3.3 Time distribution across eager execution phases . . . 36
3.3.4 Utilization of the scheduling queue . . . 37
3.4 Results . . . 38
3.4.1 Experimental setup . . . 38
3.4.2 Kernel execution time . . . 39
3.4.3 Time distribution between eager execution phases . . . 40
3.4.4 Utilization of the scheduling queue . . . 42
3.5 Related works . . . 44
3.6 Concluding thoughts and summary . . . 45

4 Multi-level Analysis of GPU Utilization in ML Training Workloads 47
4.1 Context and motivation . . . 48
4.2 Efficient ML training loop execution on GPU . . . 50
4.3 Proposed profiling methodology . . . 53
4.3.1 Profiling scope . . . 53
4.3.2 Gathering high-level and low-level metrics . . . 54
4.3.3 Coherent integration of traces . . . 54
4.4 Experimental setup . . . 55
4.5 Results . . . 57
4.5.1 Overall performance vs. memory utilization . . . 57
4.5.2 GPU compute resource utilization . . . 61
4.5.3 Implications and Insights . . . 66
4.6 Related works . . . 66
4.7 Summary . . . 69

5 Analyzing GPU Energy Consumption in Data Movement and Storage 71
5.1 Context and motivation . . . 72
5.2 Background . . . 74

5.2.1 GPU cache hierarchy . . . 74
5.2.2 GPU performance counters and internal power sensors . . . 76
5.3 Related work . . . 76
5.4 Energy model of GPU data movement . . . 78
5.4.1 Background on GPU energy models . . . 78
5.4.2 Analytical energy model . . . 79
5.4.3 Influence of parallel accesses . . . 79
5.5 Methodology . . . 80
5.5.1 Parameter identification phase . . . 80
5.5.2 Calibration phase . . . 82
5.5.3 Evaluation phase . . . 87
5.6 Implementation . . . 88
5.6.1 Experimental setup . . . 88
5.6.2 Parameter identification . . . 89
5.6.3 Calibration . . . 89
5.6.4 Cross-validation of the calibration results . . . 91
5.7 Evaluation of complete applications . . . 92
5.7.1 Matrix multiplication . . . 92
5.7.2 ResNet-50 training iteration . . . 93
5.8 Summary . . . 95

6 Conclusion **97**
6.1 Summary of key contributions and findings . . . 98
6.1.1 Analysis of Machine Learning (ML) frameworks runtime for Machine Learning (ML) inference . . . 98
6.1.2 Multi-level analysis of Graphics Processing Unit (GPU) utilization for Machine Learning (ML) training . . . 99
6.1.3 Energy breakdown of data movement in the Graphics Processing Unit (GPU) architecture . . . 99
6.2 Future work . . . 100
6.2.1 Profiling methodologies for GPU-accelerated ML workloads . . . 100
6.2.2 Software optimizations . . . 102
6.2.3 Beyond the Graphics Processing Unit (GPU) architecture . . . 102
6.3 Concluding remarks . . . 104

Bibliography **105**

List of Figures

1.1 Some of the devices supported by TensorFlow . . . 3
1.2 ML frameworks are central to the execution of ML models but are often used as a black box by the user, creating an artificial abstraction layer between them and the target device . . . 4

2.1 Training vs. inference . . . 13
2.2 TensorFlow modes of execution . . . 15
2.3 CPU/GPU architecture comparison . . . 16
2.4 CUDA Streams . . . 16
2.5 CUDA software abstractions . . . 18
2.6 Warp scheduling and reasons for stalled warps . . . 19
2.7 NVIDIA A100 architecture . . . 20
2.8 GPU architecture memory hierarchy (software abstractions) . . . 21
2.9 Typical trace from an Machine Learning (ML) profiling tool . . . 23
2.10 NVIDIA Nsight Compute interface showing a "Speed-of-Light" kernel analysis . . . 24
2.11 Illustrated example of the difference between (a) regular execution of a GPU kernel and (b) profiled execution of a GPU kernel with kernel replay [40]. . . . 24

3.1 Eager execution of a TensorFlow operation . . . 28
3.2 Phases of TensorFlow eager execution (ASYNC) . . . 36
3.3 Schematic example of the execution of 5 GPU-compatible operations and resulting node queue utilization over time . . . 38
3.4 Execution time distribution (top) and execution time per item (bottom) for LeNet-5 (a), ResNet-50 (b), and BERT (c) inference on CPU-only and CPU-GPU tandem targets . . . 39
3.5 Distribution of TensorFlow eager execution time grouped by execution phase for LeNet-5 (a), ResNet-50 (b), and BERT (c) inference on a CPU-GPU tandem . . . 41
3.6 Execution time distribution with respect to the utilization of the scheduling queue . . . 42

3.7 Utilization of the scheduling queue over time for LeNet-5 (a), ResNet-50 (b), and BERT (c) inference on a CPU-GPU tandem 43

4.1 GPU utilization and throughput training ResNet-50 with different batch sizes. 49
4.2 Multi-level utilization of the GPU computing resources 50
4.3 (a) Typical eager execution trace; (b) CPU and GPU events are listed in the ML profiler trace file; GPU's performance counters are listed in the GPU profiler report 51
4.4 Normalized throughput vs. GPU memory usage for ResNet-50, BERT, and DLRM on the NVIDIA A100 and V100 GPUs with different software optimizations. Batch size is annotated on the markers. 59
4.5 Multi-level utilization vs. GPU memory usage. TensorFlow eager execution of a mixed precision ResNet-50 across various batch sizes on (a) A100 and (b) V100 61
4.6 Multi-level utilization (A100 workloads): GPU, SMSP (active & issue slots), and tensor core (active & issue slots); batch size chosen based on best throughput value for each workload; throughput normalized by best value for each model 63
4.7 Distribution of SMSP issue slot utilization for all kernels of ResNet-50 using XLA JIT with mixed precision for multiple batch sizes on (a) A100 and (b) V100 GPUs 64
4.8 Per-layer SMSP and TC utilization distributions (showing only layers with over 3% average TC issue slot utilization) 65
4.9 Cumulative distribution of kernels' TC issue slot utilization for eager workloads (A100 GPU). Batch size is chosen based on the best throughput value for each workload 66

5.1 NVIDIA A100 DRAM and data caches hierarchy 74
5.2 NVIDIA A100 LOAD memory access process flow 75
5.3 Different phases of the methodology: (1) parameters identification, (2) calibration, and (3) evaluation 81
5.4 Miss rate of the L1 and L2 caches of the A100 GPU, over a range of array sizes between 1 kB and 1 GB 82
5.5 Thread accesses of the microbenchmarks for different numbers of threads per block, adapting the stride accordingly 85
5.6 Power trace of a 10s kernel execution on the A100 GPU with base clock, annotated with markers 87
5.7 Calibration traces of the L1 cache LOAD microbenchmark with (a) one thread per block, (b) 1024 threads per block 89

5.8 L1 (a) and L2 (b) cache LOAD calibration results 90
5.9 L1, L2, DRAM and shared memory calibration results 91
5.10 DIV instruction energy evaluation based on the difference with the LOAD calibration results (L1, L2, and DRAM) 92
5.11 Energy breakdown of MatMul for multiple sizes 93
5.12 Energy breakdown and throughput of ResNet50 using multiple software optimizations, executed using TensorFlow 94

6.1 All-digital implementation of the TANIA architecture template 103

List of Tables

4.1 High-level (top) and low-level (bottom) metrics 56
4.2 Main GPU features 57
4.3 Comparison to related works 68

List of Listings

2.1 Example snippet (CPU and GPU code) defining and launching a CUDA kernel 17
3.1 Example using a sequence of operations using the TensorFlow Python API 32
5.1 Simplified LOAD microbenchmark kernel code 83

List of Acronyms

AI Artificial Intelligence

AIMC Analog In-Memory Compute

AOT Ahead-Of-Time

API Application Programming Interface

CNN Convolutional Neural Network

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

CUPTI NVIDIA CUDA Profiling Tools Interface

DL Deep Learning

DRAM Dynamic Random Access Memory

DSE Design Space Exploration

FIFO First-In-First-Out

FP Full Precision

FPGA Field-Programmable Gate Array

GPGPU General-Purpose computing on GPU

GPU Graphics Processing Unit

JIT Just-In-Time

LSTM Long Short-Term Memory

ML Machine Learning

MLIR Multi-Level Intermediate Representation

MP Mixed Precision

NLP Natural Language Processing

NVML NVIDIA Management Library

PT PyTorch

ROCm Radeon Open Compute

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Threads

SM Streaming Multiprocessor

SMSP Streaming Multiprocessor Sub-Partition

SST Structural Simulation Toolkit

TANIA Tiled ANalog In-memory Accelerator

TC Tensor Core

TDP Thermal Design Power

TF TensorFlow

TPU Tensor Processing Unit

VFU Vector Functional Unit

VWR Very Wide Register

CHAPTER 1

Introduction

Contents

1.1 Context **2**
1.1.1 Machine Learning acceleration using GPUs 2
1.1.2 Profiling performance and energy of Machine Learning (ML) workloads on Graphics Processing Units (GPUs) 4
1.1.3 Related works 5
1.2 Contributions **6**
1.2.1 Analysis of ML frameworks eager runtime execution 6
1.2.2 Multi-level analysis of GPU utilization in ML training 7
1.2.3 Energy evaluation of data movement in the GPU architecture 8
1.3 Thesis outline **8**

1.1 Context

1.1.1 Machine Learning acceleration using GPUs

With the growth of popularity and complexity of Artificial Intelligence (AI) applications, the need for more computing power to execute and train ML models has increased considerably. This increased demand for computing power also comes with a growing energy footprint for the data centers that host these applications [1]. The interest in energy-efficient computer architectures has been growing for decades in the scientific community [26, 85, 136]. Nowadays, this interest extends to architectures dedicated to ML training and inference (i.e., ML accelerators) [29, 56, 127] and energy-efficiency is an increasingly important metric in the industry [55] to try and balance the performance and energy consumption of these architectures. Nevertheless, for a given cost budget, reducing energy consumption is of lower priority, and performance increase is still the main objective [51].

ML models architectures are constantly evolving to better adapt to the human capabilities they are designed to reproduce (e.g., vision, speech, language). Hence, the training and inference have to be executed on hardware architectures that can adapt to the changes in the ML workloads. While programmable general-purpose architectures such as Central Processing Units (CPUs) have this flexibility, they are not the most efficient for ML workloads due to the high parallelism and memory bandwidth requirements of these workloads. In contrast, ML accelerators are domain-specific architectures designed to accelerate the execution of ML models. These accelerators are designed to efficiently execute the recurring operations in ML models (e.g., matrix-multiplication, convolution), leveraging reconfigurable hardware such as Field-Programmable Gate Array (FPGA) to adapt to the changes in the ML workloads. From these explorations have emerged a wide range of ML accelerators, each with their characteristics and trade-offs [29, 33, 72, 83].

Despite the active research around new architectures for domain-specific ML accelerators, GPUs (a term popularized by NVIDIA in 1999 [45]) are still the default when it comes to ML acceleration at scale (e.g., training large ML models server-side). NVIDIA GPUs specifically have been the most popular choice over the years, mostly thanks to the strong establishment of the Compute Unified Device Architecture (CUDA) ecosystem that provides a high-level programming language to exploit the high parallelism of the GPU architecture. AMD has been the biggest competitor to NVIDIA, proposing a similar devices and software stack with the Radeon Open Compute (ROCm) ecosystem [118] and is also benefiting from the recent rise of interest in AI applications.

Originally, the GPU architecture was only designed for graphics rendering. However, the high parallelism and memory bandwidth of GPUs make them a popular choice for many other general-purpose applications (i.e., General-Purpose computing on GPUs (GPGPUs)), including ML training and inference. The recent evolutions of modern GPU architectures (e.g., tensor cores in 2017 [109]) have enabled massive throughput increases for specific tasks (e.g., matrix-multiplication [141]), which is particularly beneficial for ML workloads [151, 155]. In addition to the raw performance, the dominance of GPUs in the ML domain can also be attributed to the availability of high-level programming languages (e.g., CUDA, OpenCL) to program these accelerators by writing GPU kernels. This makes GPU programming accessible to a wide range of developers, without knowledge of the intricate details of the GPU architecture.

In this context, open-source ML frameworks (e.g., TensorFlow [2], PyTorch [121]) are major catalysts for the democratization of ML applications. ML frameworks are software tools that facilitate the development and deployment of ML models by abstracting the operation of the ML accelerator (e.g., GPU) away from the user. This high abstraction greatly enables ML application developers to focus on the functionality of ML models without worrying about the underlying hardware and low-level implementation details. However, this wide support increases the number of enabling tasks that the framework has to execute before launching execution of the core operations (i.e., kernels) on the ML accelerator. These enabling tasks constitute the framework's *runtime*, which can add time and energy overhead if not efficiently hidden by the execution of core operations. Due to this underlying complexity, ML frameworks are often operated as black boxes even though their runtime execution is key in achieving efficient execution and peak performance on the GPU accelerator.



Figure 1.1: *Some of the devices supported by TensorFlow*

To run efficiently on a plethora of hardware platforms, ML frameworks support a wide range of vendor-specific libraries to run the different operations of the ML models. These optimized libraries are designed to leverage the specific features of the hardware platform to try and achieve peak performance (e.g., cuDNN [34], which is part of the CUDA ecosystem for NVIDIA GPUs). Hence, the performance of these libraries is as important as the performance of the ML framework itself. These ecosystems represent a major competitive advantage for the GPU vendors over newer, more specialized architectures. However, the limited composability of these libraries can limit the scope of optimizations that can be applied to a complete ML model. Hence, ML frameworks

also provide compilers that can optimize the execution of the ML model by leveraging a wider scope of software optimizations [32, 44, 128, 129]. The programmable nature of GPUs is key in enabling these software optimizations for a wide range of ML models.

Building better, more efficient computing systems for ML relies on a deep understanding of the limitations of both parts of this tightly coupled hardware/software paradigm. On the one side, while software optimizations can greatly improve the performance of the ML model, they have to be guided by hardware insights to be the most effective. On the other side, the design space for ML accelerators is too large to be exhaustively explored. Reducing this design exploration space also lies in the understanding of the limitations of the software stack that will run on the accelerator.

Thus, **the main objective** of this thesis is to uncover and analyze the current limitations of ML inference and training on modern GPUs.

1.1.2 Profiling performance and energy of ML workloads on GPUs

Profiling tools (or software monitors [69]) are tools that expose application metrics to the user (e.g., execution time, memory usage, energy consumption) during the execution of a workload on a given system (see Fig. 1.2). This allows the user to extract meaningful insights, which guides the development and optimization process. Hence, profiling tools are key for software developers and hardware designers to build efficient systems. Various profiling tools exist and they each cover a different part of the software/hardware stack (e.g., ML profilers [100, 124, 148], Linux `perf` [14], Intel VTune for Intel CPUs [125], NVIDIA Nsight for NVIDIA GPUs [114]).

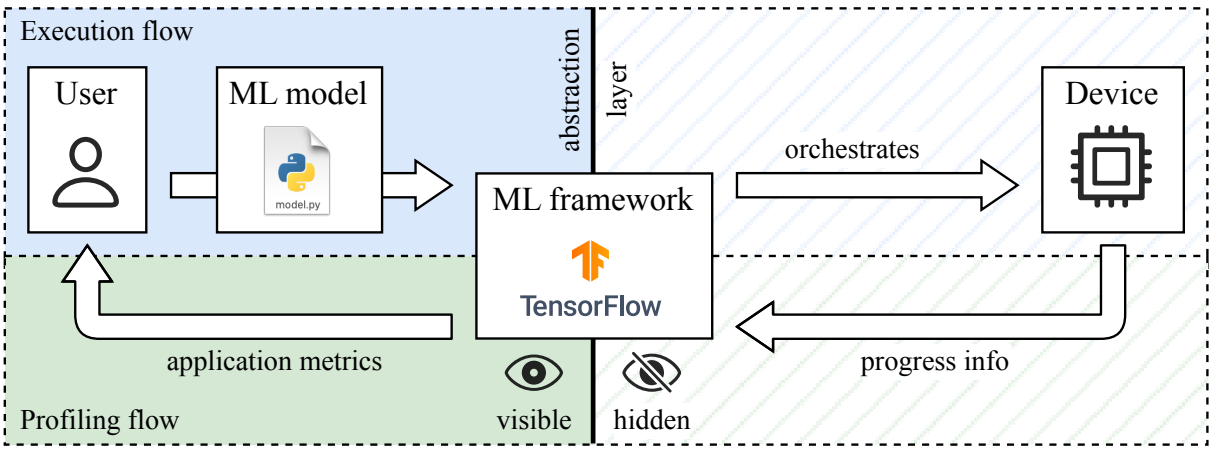


Figure 1.2: *ML frameworks are central to the execution of ML models but are often used as a black box by the user, creating an artificial abstraction layer between them and the target device*

Popular ML frameworks provide profiling tools that help developers compose ML models, monitor the training process, and identify bottlenecks in the software execution of these models. For example, ML profilers can provide insights on the convergence of the training process, the accuracy of the model, the execution time of each operation of the ML model, the memory usage, and the communication speed between the host and the accelerator. While these profiling tools are essential for software developers to build efficient ML models, they do not provide insights into the performance and energy efficiency of the accelerator architecture executing the operations of the ML model. GPU vendors also provide profiling tools that help developers build GPU kernels that are optimized for the target GPU architecture. These profiling tools can provide access to the performance counters of the GPU architecture. This helps kernel developers to assess if the GPU architecture is efficiently executing the kernel and to identify bottlenecks in its execution.

While these tools are essential for software developers to build efficient ML models and GPU kernels, computer architects have to ensure that the software stack is as optimized as possible for the hardware platform before being able to evaluate the limitations of the hardware. Hence, they have to combine the insights from multiple profiling tools across multiple levels of abstraction of the software/hardware stack. This is a challenging task as it requires expertise in the ML software stack and the GPU architecture to be able to interpret the insights from all the different profiling tools.

Thus, towards the main objective of this thesis, we provide new profiling methodologies to evaluate the performance and energy consumption of the GPU architecture executing ML workloads.

1.1.3 Related works

Here, we present an overview of works that are related to the contributions of this thesis. In the following chapters, we will provide more detailed analyses of these works and how they relate to our contributions.

The scientific community has been actively researching new methodologies to palliate the limitations of the existing profiling tools. They aim to provide new insights on the performance and energy consumption of the hardware executing ML models.

On the performance analysis side, there have been efforts to build new ML profiling platforms using existing profiling tools. Some of these platforms try to identify bottlenecks by doing automatic post-processing of the output data from a single profil-

ing tool [130, 159, 160], or aggregate the output data from multiple profiling tools [58, 90, 91, 151]. However, these platforms do not analyze the runtime execution of the ML frameworks, which is key in understanding the performance of ML execution and the limitations of the interactions between the ML framework and the GPU architecture. In addition, these platforms limit their analyses of GPU resource utilization to a high-level, application-related view, counting the time when a GPU kernel is being executed. Hence, these platforms ignore how efficiently the GPU kernel actually uses inner GPU resources (e.g., active GPU cores cycles, instruction issuing opportunities). This lack of fine-grained insights can be misleading and can over-evaluate how intensively the GPU resources are used. Hence, more fine-grained analyses are needed to understand the limitations of the GPU architecture when executing ML workloads, to further guide the software and hardware optimizations.

On the energy analysis side, there have been efforts in the scientific community to build energy models for the GPU architecture [6, 20, 61, 64, 94, 97, 101, 135, 154]. However, these models and methodologies are either based on outdated GPU architectures, rely on source code analysis of the precompiled GPU kernels, or use ML models to predict the power consumption of the GPU architecture, or ignore crucial parameters of the GPU execution that can lead to over-evaluations of the energy consumption. Moreover, these methodologies are also limited when it comes to evaluating energy consumption at a finer granularity (i.e., inner components of the GPU architecture), as manufacturer profiling tools only provide access to sensors that measure the overall power consumption of the GPU device [113]. For example, using these profiling methodologies, it is not possible to evaluate a detailed breakdown of the energy consumption of modern GPU architectures executing ML workloads.

1.2 Contributions

In this thesis, to tackle the shortcomings in the state-of-the-art, we propose three main contributions.

1.2.1 Analysis of ML frameworks eager runtime execution

First, we focus on profiling runtime execution of ML frameworks for ML inference. ML frameworks have high programming abstraction to facilitate the development and deployment of ML models. However, such an abstraction also obfuscates the runtime execution of the model and complicates the understanding and identification of per-

formance bottlenecks. Thus, we analyze how a modern ML framework manages code execution from a high-level programming language. We focus on the TensorFlow eager execution, which remains obscure to many users despite being the simplest mode of execution across ML frameworks. While this contribution focuses on this specific mode of execution, our approach can still be applied to other modes of execution in TensorFlow and other ML frameworks. We describe in detail the process followed by the runtime to execute code on a CPU-GPU tandem. We propose new metrics to uncover and analyze how the framework manages to reduce the performance overhead of its runtime execution and tries to maximize the utilization of the accelerator (GPU). We use the previously defined metrics to conduct an in-depth analysis of the inference process of two Convolutional Neural Networks (CNNs) (LeNet-5 and ResNet-50) and a transformer (BERT) looking at different batch sizes. Our results show that GPU kernels' execution needs to be long enough to exploit thread parallelism, effectively hide the runtime overhead of the ML framework, and increase GPU utilization. This strive for longer kernel execution can be achieved by using bigger batches of data during inference, seemingly pushing the need for more GPU memory.

1.2.2 Multi-level analysis of GPU utilization in ML training

Second, we focus on evaluating how efficiently ML frameworks use GPU computing resources when training ML models. To uncover this, we first describe an ideal reference execution of a GPU-accelerated ML training loop and identify relevant metrics that can be measured using existing profiling tools. Second, we describe a methodology to produce a coherent integration of traces obtained from different profiling tools. Third, we leverage the metrics from the integrated trace to analyze the impact of different software optimizations (e.g., mixed-precision, various ML frameworks, and execution modes) on throughput, memory, and computing resources utilization at multiple levels of hardware abstraction (i.e., whole GPU, SM subpartitions, issue slots, and tensor cores). Our experiments, on representative workloads, suggest that increased GPU memory capacity helps enhance throughput and GPU utilization from a high level. However, our results also suggest that a plateau has been reached, eliminating the push for bigger batch sizes. Furthermore, we observe that the tensor cores, which are the GPU cores delivering the highest raw computational power, are kept idle most of the time, and the evaluated ML training workloads are now constrained by kernels not using tensor cores. Thus, our results suggest that the current GPU paradigm is reaching a saturation point and motivates further research into programmable architectures to sustainably accelerate ML training workloads.

1.2.3 Energy evaluation of data movement in the GPU architecture

Finally, we investigate the energy efficiency of modern GPUs. While peak GPU performance drastically improved with recent architectural innovations, GPUs still consume a considerable amount of energy. Hence, identifying energy bottlenecks in the GPU architecture is crucial to designing more energy-efficient architectures in the future. However, because of the complexity and proprietary nature of modern GPU architectures, providing a detailed breakdown of the GPU energy consumption is not trivial, particularly when it comes to moving and storing data. The goal of this chapter is to estimate a lower bound for the energy consumed by data movement and storage in modern GPU architectures. To this end, we propose a basic GPU energy model along with a new profiling methodology to calibrate the model. This methodology leverages specific microbenchmarks and performance counters to evaluate the energy consumption of specific memory accesses in the GPU architecture. We implement this methodology on an NVIDIA A100 GPU and challenge the consistency of the calibration results by cross-validating with modified microbenchmarks with additional instructions. Finally, using the calibrated model, we evaluate the breakdown of energy consumption for workloads of increasing complexity, up to a training iteration of ResNet-50 with different software optimizations. Previous works have identified data movement as a major bottleneck in modern Von Neumann architectures both for execution time [18] and CPU energy consumption [74]. Our proposed methodology provides a way to easily evaluate the energy consumption of data movement in the GPU architecture for any application. Our results support the claims of the state-of-the-art and suggest that data movement is also responsible for most of GPUs' energy consumption, representing up to 84% of the dynamic energy consumption of the GPU. This further motivates the push for newer architectures that try to reduce the length of the data path (e.g., processing in-memory/near memory, vectorized architectures).

1.3 Thesis outline

We structure this thesis around six chapters. In each chapter, we give specific context and motivation for the contribution and describe related works from the literature. In Chapter 2, we introduce relevant background about ML inference and training execution flow on GPUs, and existing profiling tools from the ML frameworks and GPU vendors. In Chapter 3, we present our first contribution, which proposes to deconstruct the runtime execution of an example ML framework (TensorFlow) for inference and analyze its interaction with the GPU architecture. In Chapter 4, we present our second

contribution, which proposes a methodology to evaluate the utilization of the computing resources of the GPU architecture at multiple levels of abstraction during ML training. In Chapter 5, we present our third contribution, which proposes a methodology to evaluate the energy consumption of the data movement and storage in modern GPU architectures during and provide a detailed energy breakdown of an example ML model training workload. Finally, in Chapter 6, we conclude this thesis by summarizing the main contributions and discussing future research directions.

CHAPTER 2

Background

Contents

2.1 ML inference and training execution flow **12**
2.1.1 ML models 12
2.1.2 ML inference and training 13
2.1.3 ML frameworks 13
2.2 GPU architecture and programming model **15**
2.2.1 Programming model for NVIDIA GPUs: CUDA 16
2.2.2 GPU execution model 18
2.2.3 GPU memory hierarchy 21
2.3 ML and GPU profiling tools **22**
2.3.1 ML profiling 22
2.3.2 GPU kernels profiling 23

In this chapter, we introduce the main concepts and technologies that are used in this thesis. We start by presenting the basics of Machine Learning (ML) inference and training, and the software frameworks that support the development, implementation, training, and deployment of ML models. Then, we introduce the Graphics Processing Unit (GPU) architecture and programming model, as it is the main accelerator studied in this thesis. Finally, we present the profiling tools that ML and GPU developers use to analyze the performance of their applications.

2.1 ML inference and training execution flow

Artificial Intelligence (AI) is a broad domain that aims to develop systems to perform tasks that usually require human intelligence by mimicking human cognitive functions (e.g., visual perception, speech recognition, language translation, decision-making). Machine Learning (ML) (a subcategory of AI) is focused on building ML models using statistical algorithms and adjustable parameters (i.e., weights and biases). Tweaking these parameters allows the model to learn from data (i.e., training), to then make predictions or take decisions that are not explicitly programmed (i.e., inference).

2.1.1 ML models

ML models are composed of millions of parameters (even trillions [52]). These parameters are tunable and allow to capture knowledge within the model (i.e., *training*) which is used as a base for the model to give prediction when prompted with new unseen data (i.e., *inference*). The parameters of the model are organized within different layers, that host operations with varying shapes and sizes. The layers and operations are chosen by the developer of the ML model targetting a specific application (i.e., Deep Learning (DL) models).

For example, Convolutional Neural Networks (CNNs) are composed of many convolution layers, which capture features from 2D data by adjusting the parameters of the filters of the convolution operation. Convolution operations are usually performed using matrix multiplications (and accumulation), which are ubiquitous operations in numerous types of layers composing ML models.

2.1.2 ML inference and training

The ML inference process consists of feeding the input data (e.g., a picture) to an ML model with previously-trained parameters for it to predict an output (e.g., the object identified in the picture). The ML training process is significantly more computationally expensive than inference. It consists of feeding the ML model with a huge amount of labeled data from a dataset (e.g., a bank of pictures labeled with their description), in multiple iterations (i.e., epochs), and adjusting the model's parameters to learn the patterns in the data. This training process is done in two main steps: forward pass and backward pass. During the forward pass, similarly to the inference process, input data is fed to the model (typically in batches) for which the model predicts an output. Then, the error between the predicted output and the ground truth (i.e., the label of the input data) is calculated using a loss function. During the backward pass, the error is back-propagated through the model, and its parameters are adjusted using optimization algorithms (e.g., SGD, Adam). This backward pass is also known as backpropagation with gradient descent. An epoch ends when the model has seen all the training data. Epochs are repeated until the model converges to a satisfactory level of accuracy (i.e., the error is minimized). Fig. 2.1 illustrates these processes.

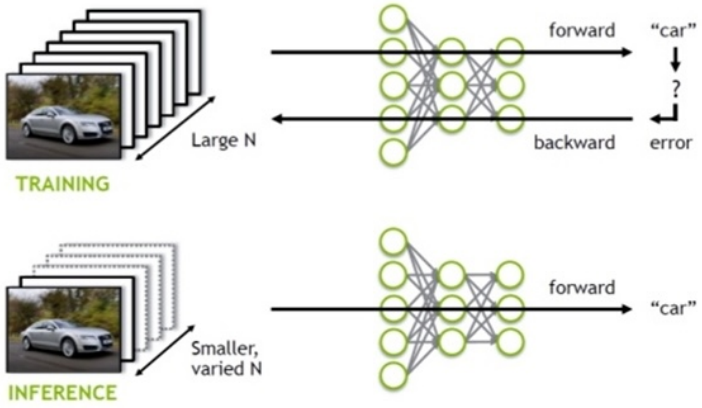


Figure 2.1: *Training vs. inference*

2.1.3 ML frameworks

ML frameworks are software tools that facilitate the development, training, and deployment of ML models. While many ML frameworks and libraries exist (e.g., MXNet, SciKit-learn, MLKit, Keras, TensorRT, ONNX), the most popular include TensorFlow (TF) [2] and PyTorch (PT) [121], which are open-source and maintained by Google and Meta, respectively. These frameworks provide Application Programming Interfaces (APIs), typically in Python, to describe ML models and orchestrate their inference and

training processes. ML frameworks also provide a set of pre-implemented ML algorithms and models. For example, using TensorFlow, one can implement a pre-trained image classification model (e.g., ResNet-50) using a single line of Python code:

```
tf.keras.applications.ResNet50(weights='imagenet')
```

ML frameworks are designed to be hardware-agnostic, meaning that the same code must run efficiently on a plethora of devices (e.g., Central Processing Units (CPUs), GPUs, Tensor Processing Units (TPUs)). To achieve this, ML frameworks rely on hardware-specific libraries (e.g., cuDNN [34] for NVIDIA GPUs) to link the high-level Python code to the lower-level hardware-specific code.

ML frameworks runtimes can follow different modes of execution, such as eager execution, graph execution, or compiled execution (see Fig. 2.2).

- Eager execution is the default execution mode in TensorFlow and PyTorch. It is an imperative interface where operations from the ML model are executed immediately as they are called from Python. This execution flow enables fast debugging with immediate run-time errors. Therefore, eager execution is most useful in the development phase of ML models. However, it is restricted to operation-level optimizations, which limits its performance.

- Graph execution extends the optimization scope by transforming the Python code into a graph of operations, which it optimizes before execution [86]. Therefore, graph execution is useful in the deployment or training phases as it achieves higher performance at the expense of easy debugging. However, the composability of the graph is limited to the same interface as eager execution, which limits the optimization scope.

- Just-In-Time (JIT) compiled execution is a mode of execution where the built graph is compiled into machine code before execution. Hence, the executed accelerator code is not limited to the same interface as eager and graph execution, which removes the overhead of dispatching the Python's operations and enables more aggressive optimizations (e.g., device-specific). This mode usually achieves the highest performance, however, the compilation time can be significant and needs to be amortized during execution. For example, XLA is an open-source compiler [129] that can be used as a backend for multiple ML frameworks and is the default backend for TensorFlow.

- Ahead-Of-Time (AOT) compilation is different from JIT as it compiles the graph into machine code before execution. In this mode, all of the optimizations are

done at design and compile time and built into the executable. This mode is useful for deployment as it removes the compilation overhead at runtime and includes the ML framework runtime into the executable along with the model's graph. As a result, it is less flexible than all the other modes of execution and produces bigger executables. For example, TensorFlow Lite [143] is a framework that uses AOT compilation to deploy ML models on mobile and embedded devices.

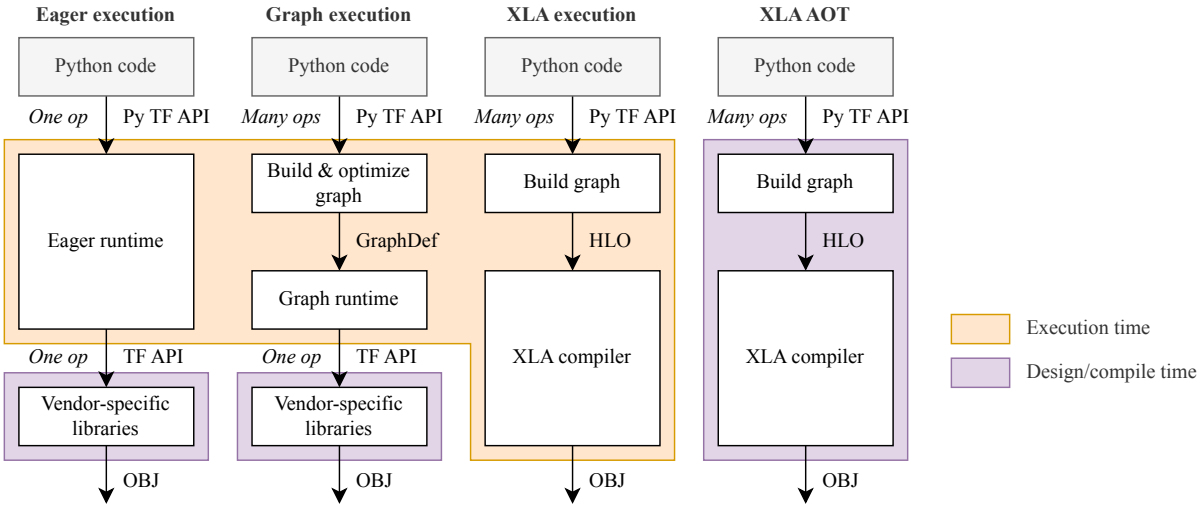


Figure 2.2: *TensorFlow modes of execution*

2.2 GPU architecture and programming model

The general-purpose CPU architecture is heavily optimized to reduce the latency of single threads (e.g., using tricks such as branch prediction, and out-of-order execution to reduce the latency of long-latency operations such as memory accesses). In contrast, the GPU architecture is designed to sacrifice single-thread performance, having a simpler execution pipeline but executing thousands of threads in parallel (i.e., Single Instruction Multiple Threads (SIMT)) to achieve high compute throughput on data-parallel workloads (e.g., Single Instruction Multiple Data (SIMD)). Fig. 2.3 illustrates the high-level architectural difference between CPUs and GPUs. However, a GPU is a programmable accelerator and needs a CPU to manage its computing tasks. Together they form a host/device pair (CPU-GPU tandem). In this section, we first present the GPU CUDA programming model and how GPUs execute code. Then, we present the GPU architecture through its execution model and memory hierarchy. In addition, we illustrate some of the key concepts using the examples of the NVIDIA V100 and A100 GPUs, which implement the Volta and Ampere architectures, respectively. We only describe NVIDIA GPUs, as they are the main accelerators studied in this thesis.

However, the concepts presented here are general and similar in other GPU vendor implementations (e.g., AMD, Intel).

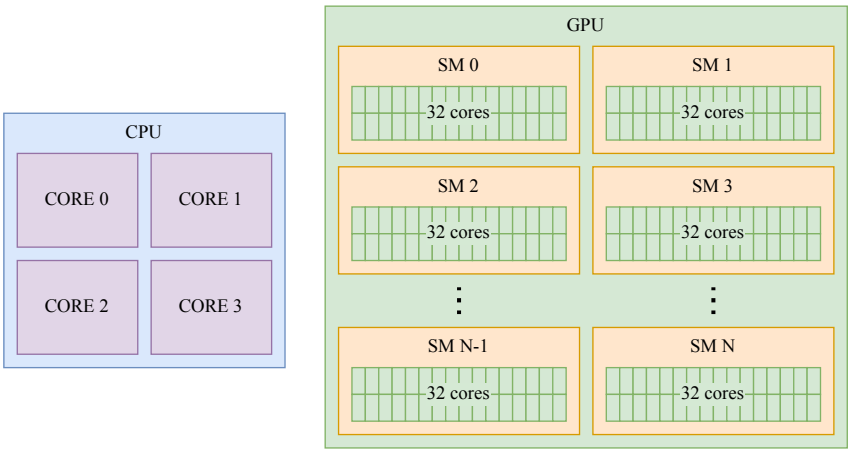


Figure 2.3: *CPU/GPU architecture comparison*

2.2.1 Programming model for NVIDIA GPUs: CUDA

To execute code on GPU devices, NVIDIA exposes a programming platform and model called Compute Unified Device Architecture (CUDA). CUDA provides a programming language that extends C++ to program several software abstractions to allow the execution of GPU functions called *kernels* in multiple *streams*.

A *kernel* is a CUDA function callable from the CPU that runs on the GPU. When developing a kernel, the programmer must describe the behavior of both the CPU and GPU in the source code. Listing 2.1 presents example snippets of both CPU and GPU code. The GPU code (i.e., kernel code) expresses the behavior of one thread. This thread is then replicated thousands of times to execute the same code on different data (i.e., SIMD execution model) when it is *invoked* (or *launched*) from the CPU code. The CPU code also manages the CPU-GPU memory exchanges (e.g., using `MemcpyH2D` to transfer from host to device and `MemcpyD2H` for the reverse operation). It is also possible to specify the stream on which to run the kernel.

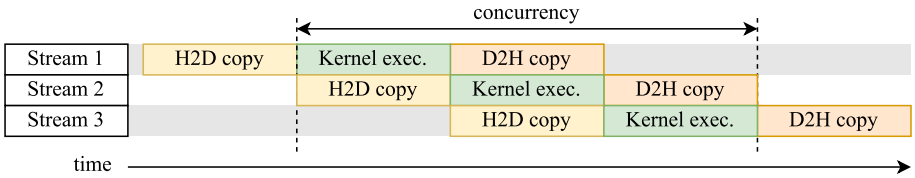


Figure 2.4: *CUDA Streams*

Streams is a CUDA programming model feature where different work can be submitted to multiple queues and processed independently by the GPU. As illustrated in Fig. 2.4, the use of streams enables concurrency between data transfer and kernel execution. CUDA defines a *stream* as a sequence of operations executing sequentially in

the order as issued from the host CPU. However, operations issued in separate streams can be executed concurrently and can overlap. In concrete, CUDA exposes the following operations as independent tasks that can operate concurrently with one another: computation on the host, computation on the device (i.e., kernel execution), memory transfers from the host to the device (i.e., H2D copy, `MemcpyH2D`), memory transfers from the device to the host (i.e., D2H copy, `MemcpyD2H`), memory transfers within the memory of a given device and among devices (i.e., D2D copy, `MemcpyD2D`).

```
// Device code (kernel definition)
__global__ void VecAdd(float* A, float* B, float* C) {
   int i = threadIdx.x;
   C[i] = A[i] + B[i];
}

// Host code (launches the kernel)
int main() {
   int T = 24;
   size_t size = T * sizeof(float);

   // Allocate input vectors h_A and h_B in host memory
   float* h_A = (float*)malloc(size);
   float* h_B = (float*)malloc(size);
   float* h_C = (float*)malloc(size);

   // Initialize input vectors
   ...

   // Allocate vectors in device memory
   float* d_A; cudaMalloc(&d_A, size);
   float* d_B; cudaMalloc(&d_B, size);
   float* d_C; cudaMalloc(&d_C, size);

   // Copy vectors from host memory to device memory
   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

   // Invoke kernel
   int threadsPerBlock = 12;
   int blocksPerGrid = 2;
   VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, T);

   // Copy result from device memory to host memory
   // h_C contains the result in host memory
   cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

   // Free device memory
   cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
   // Free host memory
   ...
}
```

Listing 2.1: *Example snippet (CPU and GPU code) defining and launching a CUDA kernel*

To simplify development, CUDA provides a set of software abstractions to the programmer to manage the execution of the kernel (see Fig. 2.5). During execution, when the GPU kernel is invoked, all the application threads are launched to the GPU for execution in a *grid*. Threads of the grid are further organized into *thread blocks*. The number of thread blocks in a grid and the number of threads in a block are specified by the programmer in the CPU code as the *kernel dimensions*. These software abstractions bring regularity to the kernel execution, enabling automatic scalability of the kernel to GPUs with different compute capabilities. This regularity of the software abstractions is matched in the hardware abstractions of the GPU architecture. This formalism is crucial to exploit the parallelism potential of SIMT execution model.

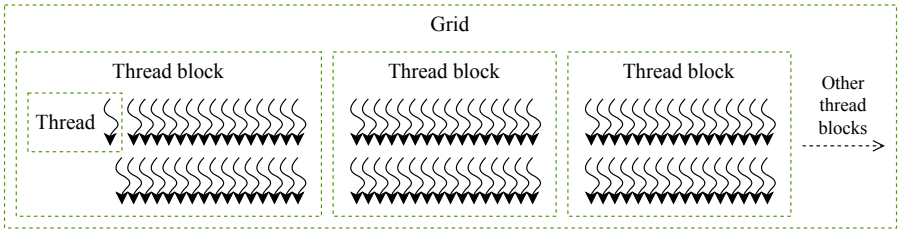


Figure 2.5: *CUDA software abstractions*

2.2.2 GPU execution model

From a hardware perspective, when a kernel is launched, the thread blocks of its grid are distributed and queued by a *thread block scheduler* to the GPU's multiple cores, called *Streaming Multiprocessors (SMs)*. The number of SMs in a GPU is fixed depending on the GPU model and its architecture design depends on the *compute capabilities* [41]. Hence, when the number of thread blocks outnumbers the SMs, the thread block scheduler queues the additional thread blocks for execution when SMs are available again (i.e., *dynamic scheduling*). This mechanism is key to the automatic scalability of the GPU architecture and helps keep the SMs busy and the GPU utilization high.

Thread blocks issued to an SM are automatically sent to one of the Streaming Multiprocessor Sub-Partitions (SMSPs) where threads are gathered into *warps* (i.e., groups of 32 threads) by a *warp scheduler*. A warp can be seen as a list of SIMD instructions, duplicated to 32 threads, which have to be executed in the SMSP's *execution units*. SMSPs host multiple execution units that can execute different types of instructions (e.g., integer, floating-point, load/store, special functions, tensor operations).

While the dynamic scheduling of thread blocks and the warp scheduling are entirely performed by the hardware, the programmer has to dimension the kernel accordingly. For example, the programmer has to dimension the kernel (i.e., grid and

thread blocks size) around the fixed warp size constraint to maximize the *theoretical occupancy* (i.e., maximum number of *active warps* per SM) relatively to the device limit (i.e., maximum number of *resident warps* per SM) [39].

Instructions within a warp are executed sequentially and in *lockstep*, meaning that all threads in a warp execute the same instruction at the same time (i.e., SIMT execution model). In the case of a divergent branch (e.g., an `if` statement), divergent threads are masked to disable their execution (i.e., *predicated execution*). Despite the sequential execution of instructions within the same warp, the warp scheduler can interleave the execution of instructions from different warp to hide the latency of long-latency operations (e.g., memory accesses). This mechanism is called *fine-grain multithreading*.

A given GPU model can host a maximum number of *resident* warps per SM. However, at kernel launch, this number can be further limited depending on the kernel's dimensions (i.e., *theoretical occupancy*). The distribution of thread blocks to the SMSPs balances the total occupancy of the GPU. At the level of an SMSP, its assigned warps are called *active* warps. At every cycle, out of all active warps in the SMSP, some can be *stalled* (i.e., cannot issue instructions to the execution units). Warps can be stalled for various reasons, such as waiting for an instruction to be fetched from memory, available execution units (i.e., the pipeline is busy), dependencies on previous memory or compute instructions or a thread synchronization barrier. Warps that are not stalled are called *eligible* warps and are candidates for issuing instructions. At every cycle, the warp scheduler chooses a *selected* warp out of the eligible warps from which to issue one instruction to the SMSP's execution units. Hence, for example, if a GPU has 2 SMs and each SM has 4 SMSPs, the warp scheduler can issue 8 instructions at every cycle. This hierarchy of occupancy is illustrated in Fig. 2.6.

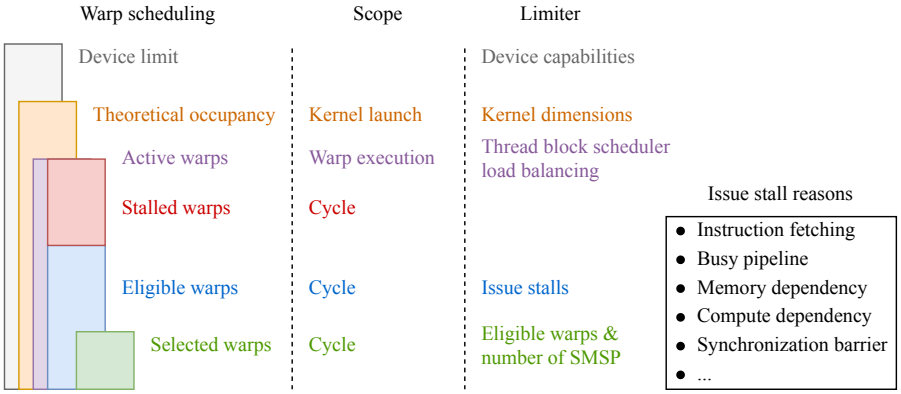
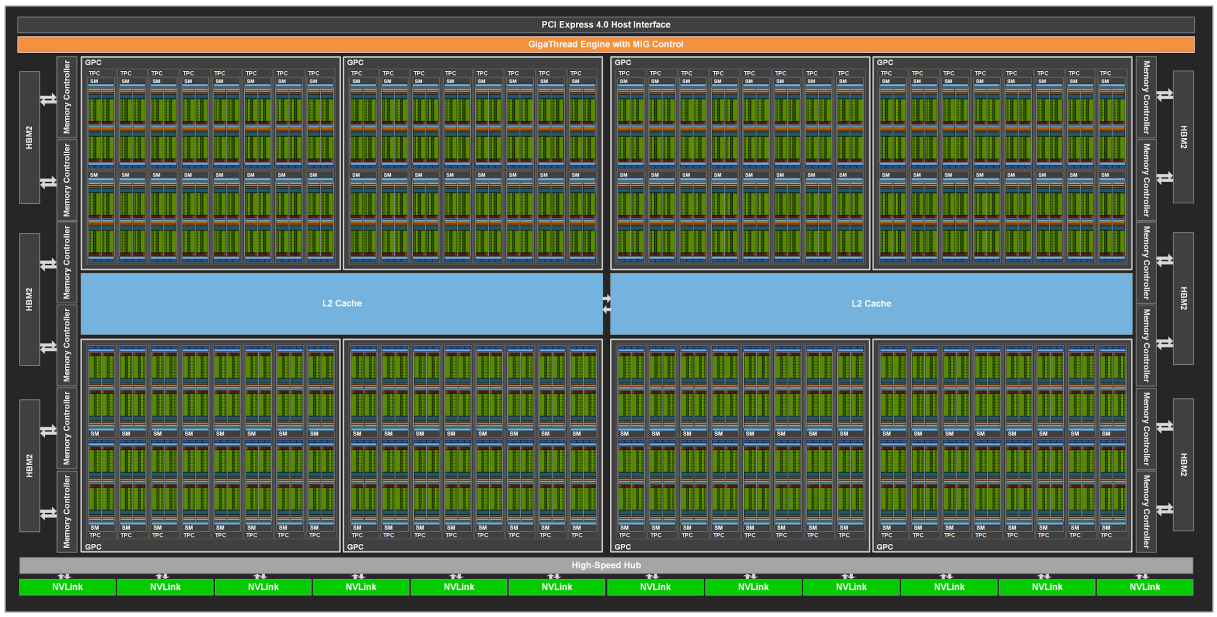


Figure 2.6: *Warp scheduling and reasons for stalled warps*

For example, the NVIDIA V100 and A100 GPUs have respectively 80 SMs and 108 SMs (see Fig. 2.7), with 4 SMSPs per SM. Both GPUs can host a maximum of 64 warps per SM (i.e., 2048 threads) and have 64 FP32 execution units per SMSP (among other execution units such as integer, load/store, special functions, and tensor cores).



(a) *Full GPU architecture*



(b) *Streaming Multiprocessor (SM) architecture*

Figure 2.7: *NVIDIA A100 architecture*

2.2.3 GPU memory hierarchy

During kernel execution, threads may access data from multiple addressable memory spaces. A thread has access to a set of *registers* and its *local memory*. These memory spaces are typically used to store the thread's local variables and intermediate results. While the registers are fast, they are limited in size. Local memory, which is slower, is used when the register space is exhausted (i.e., register spilling). Multiple threads in a thread block can share data using fast *shared memory*. All threads executing in the GPU can read and write data to and from the *global memory*, and read data from the *constant memory* and *textures memory*. The global memory is the largest memory space and is typically used to store the kernel's input and output data. Global, constant, and texture memories can also be accessed by the host (i.e., CPU) through the PCI-Express bus linking it to the GPU.

These software abstractions are mapped to the GPU memory hierarchy (illustrated in Fig. 2.8). The largest memory space is the device *main memory*, Dynamic Random Access Memory (DRAM), typically organized in multiple banks of DDR or HBM technology. The DRAM hosts the local, global, constant, and texture memory spaces. As the DRAM is slow, GPUs have multiple levels of *hardware-managed cache* memory to reduce the latency of memory accesses by exploiting data locality automatically, without programmer intervention. These levels of cache memory are increasingly smaller and faster, because closer to the execution units. Shared memory, like the lower levels of cache memory, is local to the SM. However, this memory has to be explicitly addressed by the programmer in software for threads within the same thread block to share data.

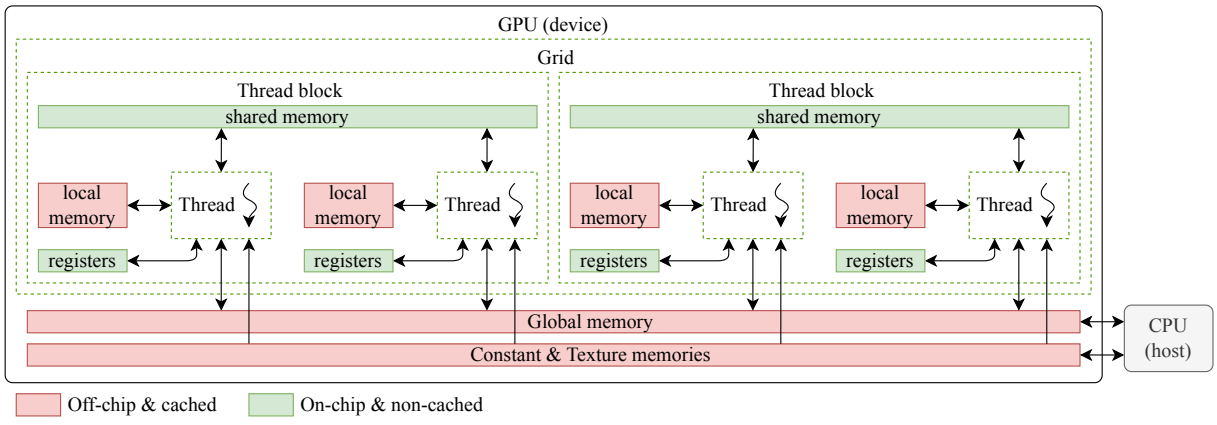


Figure 2.8: *GPU architecture memory hierarchy (software abstractions)*

2.3 ML and GPU profiling tools

Optimization of any software or hardware system requires a deep understanding of its behavior. *Profiling tools* (or *profilers*) are used to analyze the performance of these systems and identify bottlenecks. A profiler runs alongside the profiled application, collecting performance data (e.g., execution time, call stack, memory usage) to expose insights to the developer that would be hidden in normal execution. These insights are usually exposed through a report or a visualization tool. Profilers are only used during the development phase as they introduce overheads, affecting the application's performance. Profiling tools try to minimize this overhead by either using sampling techniques or giving the user control over the number of collected data points. In general, the more detailed the profiling data, the higher the overhead. In this section, we present vendor-provided profiling tools for ML models and GPU kernel development. These tools, along with related works presented in each chapter, constitute the starting point for the thesis work.

2.3.1 ML profiling

Popular ML frameworks provide profiling tools to analyze the performance of ML models (e.g., TensorFlow Profiler [148], PyTorch Profiler [124]). These tools expose APIs to the developer to instrument the high-level code of the ML model with specific annotations. The tools also provide some automatic instrumentation and graphical user interfaces to visualize the performance of the model either at design time or at runtime. For example, TensorFlow Profiler provides a graphical interface called TensorBoard [142] This tool can help analyze the composition of the model at design time (e.g., graph visualization), or identify performance bottlenecks at runtime (e.g., input pipeline performance, global memory usage). Using the TensorFlow Profiler can introduce overheads (e.g., memory usage, execution time). Execution time overhead was measured around 2% when training a small image classification model on a GPU [36].

TensorFlow and PyTorch profilers provide time traces of the execution of the model (see Fig. 2.9). These traces are saved in the Chrome Tracing format [59], originally designed for performance profiling for web and Android applications [37]. This format is JSON-based and can be visualized using the Chromium's built-in tracing tool (i.e., `chrome://tracing`). On these traces, ML profilers annotate timestamps for software events from the ML model execution (e.g., kernel launches, memory transfers). In addition, profilers also interact with hardware-specific profiling tools, such as NVIDIA's CUPTI [111] to expose additional GPU metrics (e.g., occupancy, memory usage).

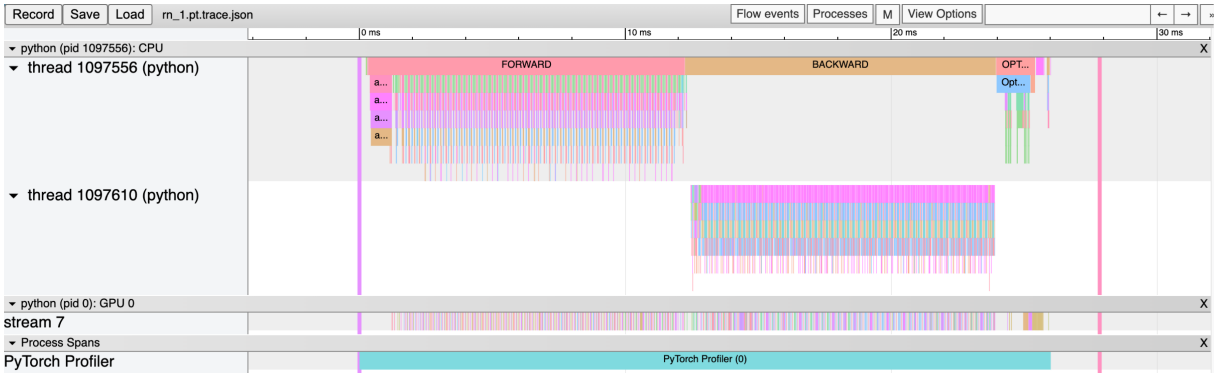


Figure 2.9: *Typical trace from an ML profiling tool*

2.3.2 GPU kernels profiling

GPU vendors provide profiling tools to analyze the performance of GPU kernels. These tools give the developer access to *performance counters*, which are hardware metrics that are collected by the GPU during kernel execution. On top of gathering the counters, these tools can also provide analyses and visualizations of the performance data to give more insights to the developer. For example, NVIDIA provides multiple profiling tools for its GPUs:

- Nsight Systems [116] is a system-level performance analysis tool. It enables the developer to analyze host-device interactions, such as memory transfers, kernel launches, and synchronization primitives. It allows the developer to tackle optimization for CPU-bound and communication-bound workloads.

- Nsight Compute (replacing NVIDIA Visual Profiler [42]) is a kernel-level performance analysis tool. This profiling tool is accessible through either a graphical user interface [114] or a command-line interface (`ncu`) [115]. It is built on top of NVIDIA CUDA Profiling Tools Interface (CUPTI) [111], which is a low-level interface to access the GPU's performance counters. Nsight Compute provides a simpler interface to access the performance counters and can also provide additional metrics and analyses, composed of the raw performance counters. It allows the developer to tackle GPU kernel optimization by providing insights on the GPU architecture runtime behavior when executing such kernel (see Fig. 2.10). For example, Nsight Compute can provide insights on reasons for issue stalls by sampling the GPU pipeline state.

Similarly to ML profiling tools, GPU profiling tools can introduce overheads. Previous works show that CUPTI adds $1\mu s$ overhead when profiling CPU-GPU data transfers and $3\mu s$ for each kernel launch [58]. In addition, when profiling using Nsight Compute, these overheads can be dramatically higher. Due to the limited number of hardware counters available on the GPU, Nsight Compute has to replay kernels multiple times to gather all required metrics (i.e., kernel replay [40]). Optionally, instead of replaying the kernels multiple times, Nsight Compute can replay the complete application to gather all required metrics (i.e., application replay). However, application replay is only possible when the number of kernels in the application is deterministic and constant between runs. The number of replays increases with the number of metrics (see Fig. 2.11). Hence, the introduced time overhead depends on the number of gathered metrics. When using kernel replay, additional time and memory overhead are added because the kernel state has to be saved and restored between replays.

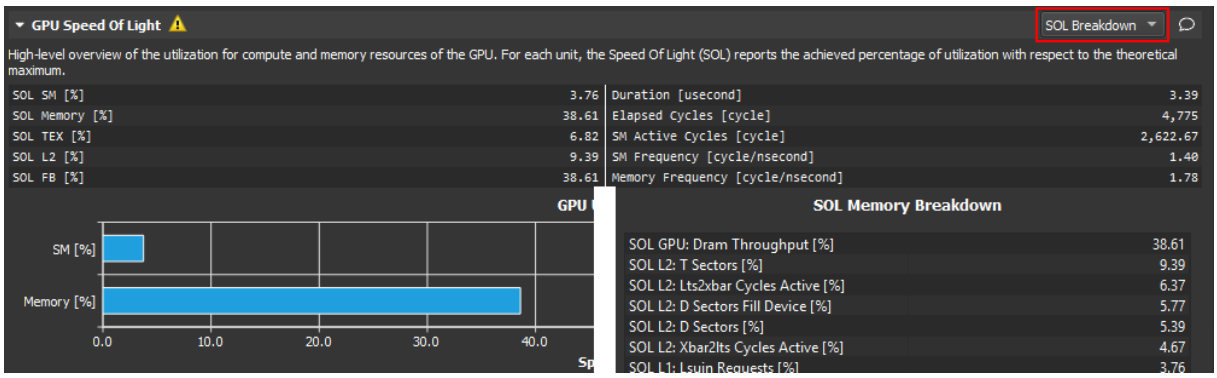
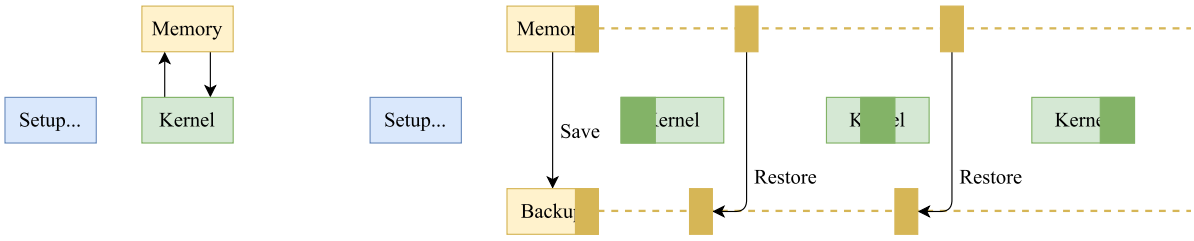


Figure 2.10: *NVIDIA Nsight Compute interface showing a "Speed-of-Light" kernel analysis*



(a) Regular Application Execution

(b) Execution with Kernel Replay.
All memory is saved, and memory written by the kernel is restored in-between replay passes.

Figure 2.11: *Illustrated example of the difference between (a) regular execution of a GPU kernel and (b) profiled execution of a GPU kernel with kernel replay [40].*

CHAPTER 3

A Deep Dive into Modern ML Frameworks

Contents

3.1 Context and motivation **26**
3.2 TensorFlow eager execution **27**
3.2.1 TensorFlow operations 29
3.2.2 Execution context 29
3.2.3 Enqueuing: Python host program 29
3.2.4 Dequeuing: Executor thread 31
3.2.5 Illustration example 32
3.3 Analysis **34**
3.3.1 TensorFlow profiling capabilities 34
3.3.2 Time spent in kernel execution 35
3.3.3 Time distribution across eager execution phases 36
3.3.4 Utilization of the scheduling queue 37
3.4 Results **38**
3.4.1 Experimental setup 38
3.4.2 Kernel execution time 39
3.4.3 Time distribution between eager execution phases 40
3.4.4 Utilization of the scheduling queue 42
3.5 Related works **44**
3.6 Concluding thoughts and summary **45**

3.1 Context and motivation

Developing Machine Learning (ML) models from scratch is not a trivial task. The models have to be composed of hundreds of operations, on multidimensional data and have to run efficiently on a plethora of devices. As described in Chapter 1, open-source ML frameworks, such as TensorFlow [2], PyTorch [121] or MXNet [31], are major catalysts of the recent explosion in the number of new ML models and applications. These ML frameworks facilitate the development and deployment of ML models by providing high-level interfaces (typically in Python) to describe the models and orchestrate their execution on different devices. The high abstraction of ML frameworks greatly enables application developers to focus on the functionality of ML models without worrying about low-level implementation details. This way, they automatically distribute the model execution across CPUs (i.e., the host) and available accelerators (i.e., the devices that have to be controlled by the host), such as GPUs.

Because of their central role in the development and deployment of ML models, ML frameworks can have a major impact on the overall performance of the ML application. However, on the flip side, their high abstraction level results in a big disparity in performance between the different frameworks [54] and their different modes of execution. For example, TensorFlow can train Long Short-Term Memory (LSTM) models $5\times$ faster than PyTorch [54]. In addition, such an abstraction obfuscates the run-time execution of the model and complicates the understanding and identification of performance bottlenecks.

For example, TensorFlow, one of the most popular ML frameworks, offers multiple different modes of execution (e.g., eager execution, graph execution, XLA). The default mode of execution, eager execution [5], is an imperative interface that executes operations immediately as they are called from Python. This enables fast debugging with immediate run-time errors. However, it is restricted to operation-level optimizations, which limits its performance compared to other modes of execution, as described in Chapter 2. While eager execution is often used only in the development phase of ML models, its performance is still key for developers to assess the validity of some design choices and thus, to iterate quickly on the model development.

TensorFlow supports a wide range of hardware platforms (e.g., CPUs, GPUs, TPUs), orchestrates the execution of models across multiple levels of abstraction, and provides different optimizations and execution strategies (e.g., graph optimizations, memory optimizations, kernel fusion). Due to this complexity, the TensorFlow codebase has grown to more than 3 million lines of code, which is practically inaccessible to most

users despite being open source [150]. As a result, users often operate TensorFlow as a black box and cannot harness the full power of the framework. We believe this to be a common trend in the community concerning modern ML frameworks.

Our **main goal** in this chapter is to demystify how a modern ML framework manages code execution from a high-level programming language. To this end, we focus on the TensorFlow eager execution, which remains somewhat of a mystery to many users despite being the simplest mode of execution in TensorFlow.

In this chapter, we propose an analysis of the TensorFlow eager execution runtime performance when running ML inference on a CPU-GPU tandem. We first describe the main steps followed by TensorFlow to execute high-level Python code on a CPU-GPU tandem. We analyze and describe how TensorFlow transforms high-level Python code into execution kernel calls to the GPU and CPU, when memory is allocated, and what triggers data transfers between CPU and GPU. From this description, we propose new metrics to expose and analyze the performance overhead that ML framework's runtime can introduce. Finally, to evaluate the extent of this overhead, we demonstrate a new in-depth profiling approach that focuses on the run-time execution of the ML framework running inference of representative ML models. To implement this approach, we develop the *TensorFlow eager runtime profiler*, a tool that extends the profiling tools provided by TensorFlow. We open source this tool in our GitLab repository [144].

3.2 TensorFlow eager execution

TensorFlow runtime is responsible for executing a user-defined model on a selected device, such as a CPU or a GPU. For the user to be able to build ML models, TensorFlow provides several APIs (e.g., JavaScript, C++, Java, etc.) that expose the TensorFlow operations to the user to interact with the underlying C++ backend of TensorFlow. The most complete, best documented, and most popular of them is the TensorFlow Python API. As discussed in Chapter 2, the TensorFlow runtime can follow multiple execution modes. In this chapter, we focus on the TensorFlow eager execution runtime.

TensorFlow eager execution is the default mode of execution of TensorFlow. It executes operations immediately as they are called from the user-defined model in Python. This behavior enables line-by-line debugging, which is not as straightforward when operations are buried in an optimized graph (i.e., when using graph execution or JIT compilation). TensorFlow eager execution follows the same specific steps to execute

each operation. We group the steps in three different phases: enqueuing, dequeuing, and kernel execution (illustrated as ①, ②, and ③ in Fig. 3.1, respectively).

TensorFlow eager execution can execute *synchronously* (SYNC) or *asynchronously* (ASYNC). In the case of SYNC execution, the enqueuing and dequeuing phases are merged into a single thread which is synchronized with the device kernel execution. This single main thread starts with the call of an operation in Python, launches the execution on the device, and waits for the return value before executing the next operation. In the case of ASYNC execution, the enqueuing and dequeuing phases are performed by separate threads, separated by a *scheduling queue*. The enqueuing thread starts with the call of an operation in Python and pushes one or several *computation nodes* in the scheduling queue. Asynchronously, the dequeuing thread pops the nodes from the queue and manages their execution. A computation node is a software object gathering all the information needed to call kernel execution on the target device (CPU or GPU).

In this section, we provide a detailed description of these different execution phases within the TensorFlow eager execution runtime when running on a CPU-GPU tandem. We first describe how models are composed in Python using TensorFlow operations (Section 3.2.1) and how TensorFlow keeps track of the execution context (Section 3.2.2). Then, we detail the enqueuing (Section 3.2.3) and dequeuing (Section 3.2.4) threads and how they orchestrate the execution of the TensorFlow operations. Finally, we illustrate the complete process with a simple example (Section 3.2.5).

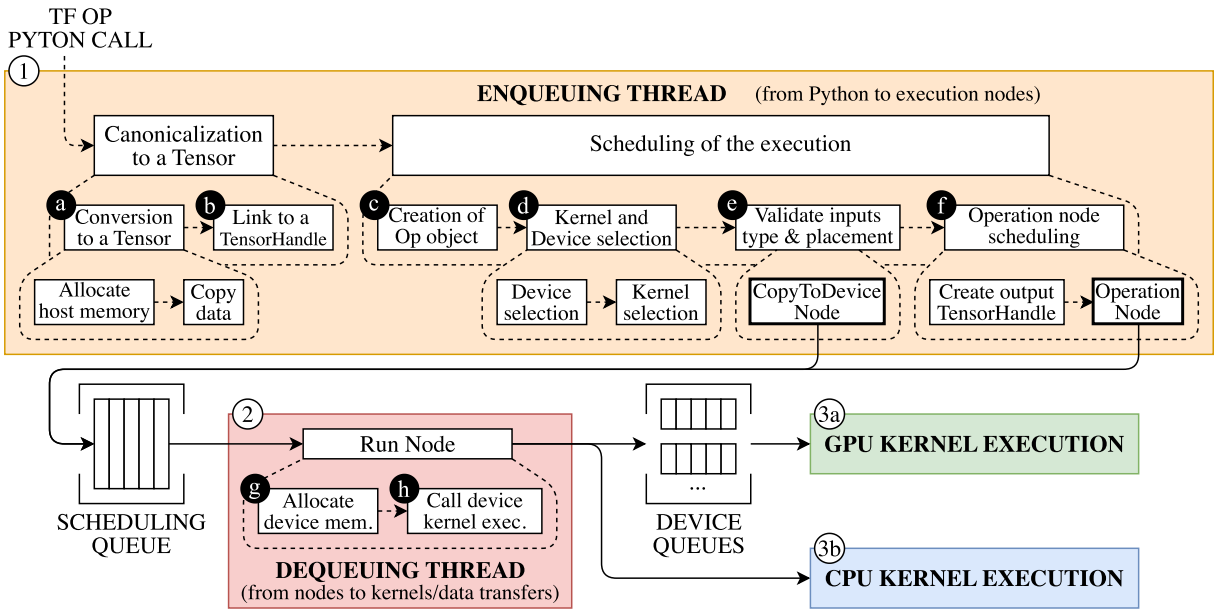


Figure 3.1: *Eager execution of a TensorFlow operation*

3.2.1 TensorFlow operations

To compose a TensorFlow model in Python, the user chooses TensorFlow operations (e.g., matmul, relu, etc.) as defined in the Python API [7]. TensorFlow operations are linked to optimized kernels: CPU kernels for every operation, and CUDA kernels for GPU-compatible operations. These kernels are accessible from the Python API via wrapper functions, which are automatically generated for every registered operation when building TensorFlow from source.

TensorFlow includes a very complete set of already registered operations but still allows developers to add custom operations [145]. To register a new operation in TensorFlow, developers must (1) define its interface by specifying its inputs, outputs, and attributes types and shapes in a C++ module, (2) implement a CPU *OpKernel* for the operation, (3) implement a CUDA kernel if GPU compatibility is intended, and (4) link the previously defined kernel(s) with the specific operation.

3.2.2 Execution context

When starting to execute the Python wrapper function of a given TensorFlow operation, the runtime has to know which mode of execution has to be used to run (e.g., eager or graph mode). This user-configurable parameter is stored in the *execution context*, which is globally accessible from both the Python frontend and the C++ backend of TensorFlow. The context is a collection of configuration parameters used by the runtime to execute the Python wrapper functions, such as execution mode, device placement policy (i.e., policy to use when running an operation on a device with inputs that are not on that device), etc. The context is initialized at the start of the Python program execution and is referred to whenever the runtime needs these configuration parameters to make choices during the execution. Most of the data stored in the context is thread-local, which is important for the runtime to be thread-agnostic. Thus, multiple TensorFlow programs can safely run in parallel.

3.2.3 Enqueuing: Python host program

Fig. 3.1 shows the main steps in the enqueuing thread ①. The eager execution of an operation starts with the *canonicalization* ⓐ of its inputs to the *Tensor* data type. At this point, memory is allocated on the host to store the inputs after the conversion to tensors. A Tensor is a multi-dimensional array defined by three parameters: a data type,

a shape, and the actual values. However, the TensorFlow runtime does not directly manipulate the Tensor data type but interacts with it through a *TensorHandle* ⓑ. A TensorHandle represents a tensor (or a not-yet-computed *FutureTensor*) that lives (or will live) on a device. A TensorHandle can provide the shape and type of the tensor even if the actual value is yet to be computed. This mechanism allows the runtime to race ahead and continue parsing Python operations without having to wait for the results of the computations. As a result, the main Python thread can run in parallel with an execution thread (i.e., ASYNC execution) instead of stalling while the operation is being executed (i.e., SYNC execution). In ASYNC execution, if the executed kernels are sufficiently heavy to run, the Python thread can race ahead of the device execution and keep the device utilization high by hiding its latency. However, when the runtime needs a tensor value to take a decision (e.g., a data-dependent if-then-else statement), it will stall until the required value becomes available (see Section 3.2.5 for an example).

After securing the inputs, the runtime gathers all the remaining information needed for the operation to execute in an *Op* object ⓒ. The Op object includes the name of the operation to run (as a string), pointers to the input data (i.e., encapsulated in a TensorHandle), and operation-dependent attributes (e.g., if one of the inputs needs to be transposed before the operation). Then, the runtime selects a device to execute the operation according to the execution context (if a device is specified by the user) or follows an internal heuristic to find the fastest available device. Based on the name of the operation, the type and shape of its inputs, and the target device, the runtime also selects the kernel (i.e., OpKernel) to execute ⓓ.

At this point, the runtime validates the placement of the inputs of the operation ⓔ. If the inputs are not placed in the device targeted to execute the operation, the runtime schedules the necessary data-transfer operations. In the case of ASYNC execution, the runtime creates a specific node to handle data transfer that can be stored in a scheduling queue. For example, in the case of a *host to device data transfer* action, which is automatically inserted by the runtime, the input data corresponds to the input data of the original operation (i.e., TensorHandle), the target device is the same as in the operation, and the prompted OpKernel links to a CUDA *MemcpyH2D* call.

Once the input placement is verified and scheduled, the runtime schedules the operation itself ⓕ. First, it creates a TensorHandle for the output of the operation, which is a FutureTensor. This allows the runtime to schedule future operations taking this output as an input without having to wait for the actual Tensor value (i.e., ASYNC execution). Then, the runtime creates a second node object gathering the Op object, a pointer to the selected device, and the selected OpKernel.

In the case of SYNC execution, the runtime runs a node immediately and stalls while waiting for a return value. In the case of ASYNC execution, the runtime will insert the node in a scheduling queue and directly return control of the enqueuing thread. This architecture enables a layer of parallelism between the enqueuing and dequeuing threads, which TensorFlow uses to try to hide the latency of the main Python thread (enqueuing) behind the executor thread (dequeuing).

3.2.4 Dequeuing: Executor thread

The nodes in the scheduling queue are dequeued by a separate *executor thread* following the enqueuing order (i.e., First-In-First-Out (FIFO)). The executor thread stalls or executes a queued node depending on whether the targeted device is busy or not.

The execution of a node consists of two main steps: the memory allocation of the operation outputs (g), and the call for kernel execution on the targeted device (h). A kernel describes the computations to perform and can also allocate memory for intermediate results when needed. TensorFlow includes a rich set of optimized kernels implemented using external libraries such as Eigen [60] and cuDNN [34], in major part for GPU devices. Eigen is a C++ template library for linear algebra that can generate kernels for multiple input data types and target devices using the same codebase. This library is particularly useful for complicated linear algebra operations, as it eliminates the need to write optimized code for every supported data type, shape, and device. cuDNN is a library developed by NVIDIA that provides GPU-accelerated primitives for deep learning such as convolution, pooling, normalization, activation layers, and tensor transformation.

As described in Chapter 2, the GPU programming model allows for asynchronous execution of kernels through the use of CUDA streams. TensorFlow has a very specific design for using CUDA streams. There is a main *compute* stream, a main pair of *host_to_device* and *device_to_host* streams, as well as a vector of *device_to_device* streams (used when computations are distributed over multiple devices). The main compute stream can handle transfers and computations. Secondary compute streams can be used, however, they cannot perform computations concurrently with the main stream. This single-compute-thread implementation simplifies the management of the GPU device. It bets on the fact that streams do not provide a significant performance improvement when running large kernels on the GPU, assuming that the kernels occupy most of the GPU's resources.

3.2.5 Illustration example

In Listing 3.1, we illustrate the most important execution steps of TensorFlow eager runtime using a simple example. We consider the example to run in ASYNC mode on a system with a local CPU host and a local CUDA-compatible GPU device.

```
1  import tensorflow as tf
2  import numpy as np
3
4  with tf.device("/GPU:0"):
5    x = np.random.randn(64, 64)
6    y = tf.constant(np.random.randn(64, 64))
7
8    z = tf.matmul(x, y, transpose_b=True)
9
10   output = tf.nn.relu(z)
11
12 print(output)
```

Listing 3.1: *Example using a sequence of operations using the TensorFlow Python API*

The gist of the example is a sequence of Python and TensorFlow operations that creates a random matrix of size 64×64, creates a second random matrix of the same size explicitly converted to a constant Tensor, performs matrix multiplication of the two matrices, performs a ReLU activation on the resulting output, and prints the result.

We now describe in detail how TensorFlow executes the code. Line `1` imports the TensorFlow package into the Python program. This import has the side effect of initializing the *execution context* which stores the configuration information needed by the runtime. By default, the execution mode specified in the context is eager execution. Line `2` imports the *NumPy* package, which is a popular Python package that offers mathematical functions to work with multidimensional arrays and matrices. In the example, it is used to create random inputs to feed the operations. Line `4` defines a TensorFlow scope (delimited by the `with` statement) that we use to specify the device on which we want to execute the operations (here, the GPU). Then, this information is stored in the execution context and is referred to during the subsequent steps of the process. Line `5` uses the NumPy package to initialize a Python variable, storing a random matrix of shape 64×64 and of default data type (i.e., float64). Despite being called in the TensorFlow scope, this operation is independent of TensorFlow and does not invoke any TensorFlow runtime functionality.

Line 6 executes in multiple steps. The first step executes the same operation as Line 5: it creates a random matrix of shape 64×64 and of type float64. The second step is a call to the TensorFlow operation *tf.constant*. The execution of this operation is then taken up by the TensorFlow eager execution runtime. The runtime starts with the canonicalization of the input to a Tensor ⓐ: it allocates host memory to store the resulting Tensor and copies the actual data of the matrix to the Tensor. The type and shape of the Tensor data follow the shape and type of the NumPy matrix: 64×64 and float64. Then, the Tensor is linked to a TensorHandle ⓑ that lives in the CPU. The runtime creates an Op object ⓒ which gathers: the operation name (i.e., *EagerConst* is the tf.constant operation name), the newly created TensorHandle, and no attributes (the operation does not have any). Next, the runtime selects the target device (i.e., GPU as specified by the TensorFlow scope in Line 4 and stored in the execution context) and the kernel to run (*IdentityOp* for executing tf.constant in the GPU) ⓓ. The runtime validates the placement of the input ⓔ, which should also be in the GPU. However, the input TensorHandle currently lives in CPU memory, so the runtime creates a *CopyToDevice* node to execute the required data transfer and inserts it in the scheduling queue. Then, the runtime moves to the node scheduling of the tf.constant operation ⓕ: creating a TensorHandle for the output (linked to a FutureTensor and lives on the GPU device) and the corresponding node object. Finally, the runtime inserts the node in the queue before returning the control to Python.

Line 8 follows a very similar process as Line 6 but with four key differences. First, the *tf.matmul* operation takes `x` and `y` as inputs. While `y` is already a Tensor, `x` is still a NumPy array. Hence, when converting the inputs to Tensors ⓐ, the runtime prompts a tf.constant operation scheduling to ensure the conversion of `x` and its placement on the GPU. This tf.constant operation also triggers the creation of a TensorHandle ⓑ which lives on the GPU device. Second, the tf.matmul Op object ⓒ includes the `transpose_b` attribute, which is passed as an argument to the device kernel when executing the node. Third, the runtime selects the GPU as the execution device and chooses an OpKernel ⓓ that is linked to a CUDA kernel, from the cuDNN library, named `volta_dgemm_128x64_tn`. Fourth, as the two TensorHandles linked to the inputs already live in the GPU device, the runtime does not need to prompt any data transfer (i.e., no CopyToDevice node is created).

Line 10 launches the execution of the *tf.nn.relu* operation, which follows the same process as the previous operation but with some simplifications. As its input is already a Tensor and lives on the GPU device, the runtime skips the conversion to Tensor ⓐ and the CopyToDevice node when validating the placement of the input ⓔ. This operation is linked to a GPU kernel named `Relu_GPU_DT_DOUBLE_DT_DOUBLE_kernel`.

Line `12` calls the Python `print` function on the Tensor `z`. This function is overloaded by TensorFlow: when the `print` function is called on a Tensor, the runtime uses the corresponding TensorHandle to know where the data is located. In this case, the data is a FutureTensor in the GPU. Accordingly, the runtime needs to copy the value of the Tensor `z` from the GPU to the CPU and convert it to a Python printable data type (i.e., NumPy array), which is then printed by Python. However, to execute this operation TensorFlow needs the actual value of the Tensor, which is only produced after the execution of the tf.nn.relu operation. As a result, the enqueuing thread stalls and waits for the value of the Tensor `z` to be produced.

In parallel with the enqueuing thread, the executor thread (i.e., dequeuing thread) has been processing the nodes in the scheduling queue. For each node, the executor thread allocates GPU memory for the output (g) and calls the corresponding GPU kernels (f). Once all nodes have been dequeued and executed on the GPU device, the numerical value of the Tensor `z` is stored in the GPU memory. Thus, the enqueuing thread is notified that the value of `z` is available and resumes processing Line `12`. It sends a Memcpy command to copy the data from the GPU to the host, storing it as a printable data type (i.e., NumPy array). Finally, the Python interpreter prints the value of `z` in the user's console.

3.3 Analysis

In this section, we present our approach to estimating the runtime performance overhead of TensorFlow eager execution using a target CPU-GPU tandem. We leverage TensorFlow profiling capabilities to analyze TensorFlow eager execution on a given workload. From this analysis, we extract three key metrics: the share of the execution time spent in CPU or GPU kernel execution, the distribution of the execution time across the different phases of eager execution for each operation, and the scheduling queue utilization over time.

3.3.1 TensorFlow profiling capabilities

As described in Chapter 2, TensorFlow offers a profiling tool to analyze the performance of a model (i.e., TensorFlow Profiler [149]). This profiler collects performance data to help understand hardware resource utilization and identify performance bottlenecks in the model. It collects profiling data sent by the TensorFlow runtime (e.g., the execution time of operations and functions of the runtime) and device metrics (e.g.,

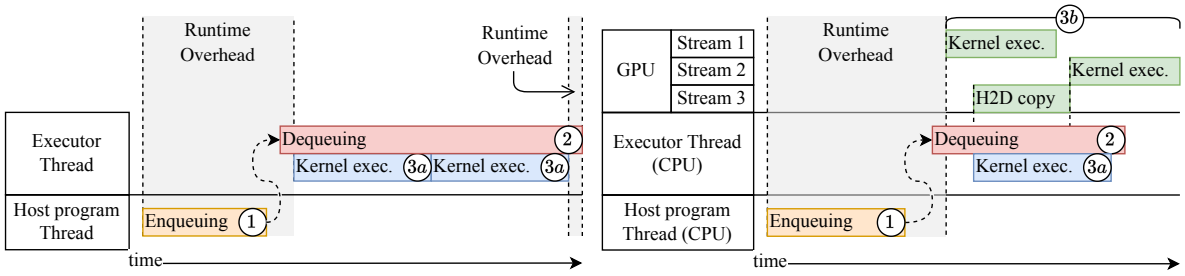
kernel execution times, memory usage) recovered using the NVIDIA CUDA Profiling Tools Interface (CUPTI) [111]. Based on the collected data, TensorFlow offers different visualization tools (i.e., TensorBoard [142]) to gain insight into the input pipeline of the executed model, the distribution of TensorFlow operations between host and device, statistics about GPU and CPU kernels, memory profiling, and training statistics.

These tools provided by the profiler focus on helping developers build their models and help them debug performance for inference and training. However, these tools do not provide metrics to understand and analyze the inner mechanisms of the TensorFlow runtime. This makes the user unaware of the performance overhead introduced by the framework itself, and the optimization choices made by the runtime. Thus, in this section, we propose to extend the profiler with new analysis metrics to gain a further understanding of how the framework optimizes and schedules the execution of operations. Fortunately, the data gathered by the profiler are saved in local JSON files during execution. Therefore, we can leverage these data to provide new metrics and analysis to highlight how the framework optimizes and schedules the execution of operations. We open source these extensions [144] and hope to help average users harness the full power of the TensorFlow eager runtime.

3.3.2 Time spent in kernel execution

For a TensorFlow program to achieve high performance, the time spent executing CPU and GPU kernels, which correspond to the real operations in the ML model, should dominate the total execution time. In practice, however, an ML runtime also needs to execute other enabling tasks such as parsing the Python code, copying data between CPU and GPU, etc.

Our first analysis goal is to provide insight into how execution time is divided between kernel execution and the rest, which we consider to be the overhead of the runtime (annotated "Runtime Overhead" on Fig. 3.2). To this end, we evaluate the distribution of execution time by parsing the data gathered by the TensorFlow profiling tool. We distribute the profiled events between three categories: CPU kernel execution, GPU kernel execution, and the rest of the execution. Additionally, when executing on a CPU-GPU tandem, the CPU and GPU kernel execution times can overlap, which is the best-case scenario regarding computing resource utilization (like in Fig. 3.2b). Thus, we also track how often this happens.



(a) *CPU only execution* **(b)** *CPU-GPU tandem execution*

Figure 3.2: *Phases of TensorFlow eager execution (ASYNC)*

3.3.3 Time distribution across eager execution phases

The TensorFlow eager execution of an operation (described in Section 3.2) can be divided into three major phases as shown in Fig. 3.1: *enqueuing* ①, *dequeuing* ②, and *kernel execution* ③. Fig. 3.2 illustrates the phases for two different scenarios: (a) CPU-only execution, and (b) CPU-GPU tandem execution.

First, the TensorFlow runtime handles the groundwork of execution: collecting input data and attributes for the operation, selecting a device on which to execute and a corresponding kernel, and packaging everything into a node. This node will be either: executed directly by the executor thread (i.e., dequeuing thread), stalling the enqueuing thread (SYNC mode); or, queued for later execution, releasing control of the enqueuing thread (ASYNC mode). The enqueuing and dequeuing threads are both running on the host. We define this part of the execution as the *enqueuing* (① in Fig. 3.2).

Second, as soon as a node is placed in the queue, the executor thread handles the last steps before the kernel execution: allocating memory for the output, instantiating the kernel to execute, and calling the kernel execution on the targeted device. We define this second phase of the operation execution as the *dequeuing* (② in Fig. 3.2).

In the case of CPU kernel execution, both dequeuing and CPU kernel execution are performed by the executor thread. However, we separate CPU kernel execution time (③*a* in Fig. 3.2a) from the rest of the dequeuing (② in Fig. 3.2a).

Finally, the executor thread issues the corresponding kernel(s) to the targeted device for execution (③*a* and ③*b* in Fig. 3.2a and 3.2b, respectively). As our host/device execution target is a CPU-GPU tandem, this third phase of the execution corresponds to GPU kernel execution, for most of the operations. However, some operations have only a CPU kernel (see Section 3.2.1). In such case, the execution corresponds to *CPU kernel execution time*. Thanks to the stream mechanism (described in Section 3.2.4), data transfers between CPU and GPU can occur while a GPU kernel is executed.

The time spent enqueuing and dequeuing has to be shorter than the kernel execution to take advantage of the latency hiding mechanisms described in Section 3.2. Thus, we also analyze the distribution of the execution times of each phase.

3.3.4 Utilization of the scheduling queue

As described in Section 3.2, TensorFlow eager execution includes a mechanism to hide node execution scheduling time: the *scheduling queue*. The last part of our analysis focuses on the utilization of this scheduling queue. To this end, we study the dequeuing phase and the kernel execution phase with two different metrics: scheduling queue utilization over time, and the distributions of the execution time between phases, when the scheduling queue is empty or not empty (i.e., loaded with at least one node).

To profile the utilization of the scheduling queue over time, we use the rest of the profiling events (not sorted as kernel execution) at our disposal and distribute them in two new categories: enqueuing events and dequeuing events. We use these two categories to deduce the utilization of the scheduling queue over time. We assume that a node is in the queue from the end of the enqueuing phase until the beginning of the dequeuing phase. With this assumption, we can recreate the scheduling queue utilization over time.

The role of the scheduling queue is to hide the node execution scheduling time. Thus, we cross-analyze this queue utilization with kernel execution timings and dequeuing timings. We consider three scenarios:

1. When CPU or GPU kernels are executing, the scheduling time is masked regardless of the utilization of the scheduling queue (annotated "kernel exec." time in Fig. 3.3). This is the ideal scenario.

2. When the executor thread is active (i.e., the thread is dequeuing a node from the scheduling queue), the scheduling time is masked regardless of the scheduling queue utilization (annotated "Runtime overhead: dequeuing node" in Fig. 3.3). While this case is less ideal (and still considered runtime overhead), it still shows that the scheduling queue is being used.

3. When the executor thread is stalling, waiting for the enqueuing thread to insert a node in the queue (annotated "Runtime overhead" in Fig. 3.3), the scheduling queue is underutilized (i.e., the dequeuing thread is starved).

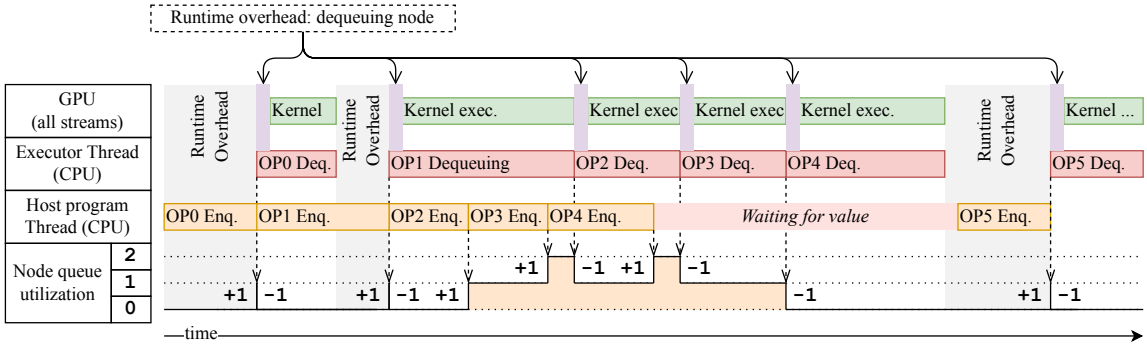


Figure 3.3: *Schematic example of the execution of 5 GPU-compatible operations and resulting node queue utilization over time*

3.4 Results

In this section, we use the approach from Section 3.3 to analyze TensorFlow eager execution on three inference workloads: two Convolutional Neural Networks (CNNs) (i.e., LeNet-5 and ResNet-50) and a Transformer-based model (i.e., BERT). The two CNNs are tried-and-tested models in the research community and are mainly used for image classification tasks. The BERT model, more recent, is one of the first Transformer-based models and represents a baseline for Natural Language Processing (NLP) tasks.

3.4.1 Experimental setup

To perform our analysis of the TensorFlow eager execution runtime, we used a CPU-GPU tandem (i.e., two Intel Cascade Lake 6248 with a total of 40 cores and 192GB of RAM, paired with an NVIDIA V100 with 32GB of dedicated RAM). We use TensorFlow [2] v2.8.0 as the target framework for our experiments.

The LeNet-5 model is recreated using TensorFlow, following the 1989 paper from LeCun et al. [88]. We use the ResNet-50 pre-trained model from the TensorFlow API Keras Applications [146], which follows the architecture described by He et al. [63]. The BERT pre-trained model is retrieved from the HuggingFace library [16] and follows the architecture described by Devlin et al. [50]. We execute the inference in the pre-trained models on TensorFlow with three representative batch sizes (one, medium, and high) chosen experimentally for each model and according to the memory restrictions of the hardware. For each experiment, we run the inference once with the same batch size before profiling the execution. This enables us to compare our workloads with already cached OpKernels, avoiding irregularities (e.g., `red_zone_checker` kernel checks) in the execution time due to kernel instantiations.

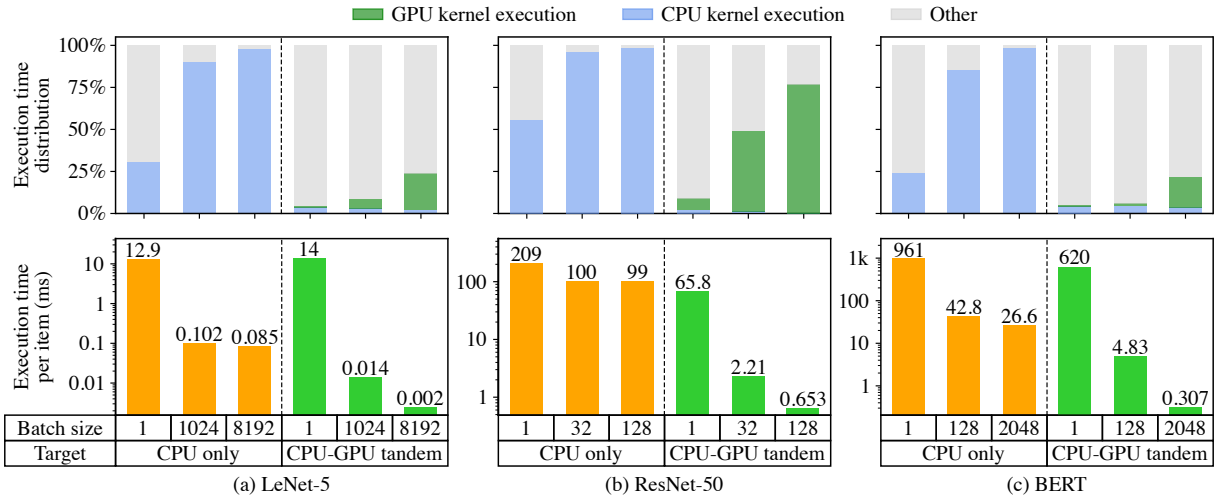


Figure 3.4: *Execution time distribution (top) and execution time per item (bottom) for LeNet-5 (a), ResNet-50 (b), and BERT (c) inference on CPU-only and CPU-GPU tandem targets*

3.4.2 Kernel execution time

We evaluate the share of the execution time dedicated to CPU and GPU kernel execution running on both our CPU-only target and our CPU-GPU tandem. We also evaluate the execution time per item (i.e., execution time divided by batch size) using the same workloads on the same platforms.

Fig. 3.4 (top) shows the execution time distribution (between CPU kernel execution, GPU kernel execution, and the rest) for LeNet-5 (a), ResNet-50 (b), and BERT (c). We make three observations. First, all the workloads show an increase in kernel execution share when increasing the batch size, both when executing on CPU-only and on CPU-GPU tandem. For example, on a CPU-GPU tandem, the LeNet-5 kernel execution time increases from 5% to 24% for a batch size of 1 and 8192 images (28×28 black and white), respectively; the ResNet-50 kernel execution time increases from 7% to 75% for a batch size of 1 and 128 images (224×224 RGB), respectively; and the BERT kernel execution time increases from 5% to 22% for a batch size of 1 and 2048 sentences (512 tokens long), respectively. Second, the execution time of the CPU kernels becomes negligible when running on a CPU-GPU tandem. We observe that 3% of the execution time is spent in CPU kernel execution on average. In addition, the concurrent execution time between CPU and GPU kernel executions, which we include as part of the GPU kernel execution time in the figure, represents less than 1% of the total execution time. This shows that TensorFlow manages to use the GPU for most of the kernel execution when computing on large batch sizes. However, for small batch sizes, we can see that CPU kernel execution time can be higher than GPU kernel execution time. Finally, the overhead of the framework (i.e., *Other* in Fig. 3.4) is considerable across all the workloads when using small batch sizes. It represents 95% of the execution time when running

LeNet-5 on a CPU-GPU tandem with a single image. For the remaining workloads executing on a CPU-GPU tandem, it represents more than half of the execution time. The only exception being the ResNet-50 with a batch size of 128 images, where the framework overhead is reduced to 23% of the total execution time. We see two main reasons for explaining this behavior. The first reason is that, when using a CPU-GPU tandem, operations executed on the GPU are executed faster, compared to only executing on a CPU target, thanks to the acceleration provided by the GPU device. This results in less GPU kernel computing time, leading the framework scheduling to be dominant. This fact is coupled with the second reason, which is intrinsic to how eager execution works. Eager execution has to schedule every operation one by one. Hence, for the latency of the framework to be negligible for one operation, its enqueuing time has to be significantly shorter than the execution time of its kernel.

Fig. 3.4 (bottom) shows the execution time per item, for LeNet-5 (a), ResNet-50 (b), and BERT (c). We observe that the execution time per item drastically reduces as the batch size increases. For example, the execution time per sentence of BERT becomes $37\times$ and $2019\times$ faster when increasing the batch size from 1 to 2048, running on a CPU-only and CPU-GPU tandem, respectively. LeNet-5 and ResNet-50 follow the same trend, exhibiting performance improvements per image of $7000\times$ and $100\times$ when increasing their respective batch sizes.

3.4.3 Time distribution between eager execution phases

We evaluate the distribution of the execution times of the different execution phases in the eager execution of all the operations of each workload.

Fig. 3.5 shows the distribution of the operations execution time between enqueuing time, dequeuing time, GPU execution time, and CPU execution time for LeNet-5 (a), ResNet-50 (b), and BERT (c). Each distribution includes a marker to indicate the minimum, mean, and maximum values. We make three observations.

First, the enqueuing, dequeuing, and CPU execution times are largely independent of the batch size. We observe that the average enqueuing latency is around $0.1ms$ for each of our workloads. Although increasing the batch size induces larger kernels to run, this does not affect on the time needed to enqueue and dequeue the operations to and from the scheduling queue. Moreover, this batch size increase does not affect CPU kernel execution time. GPUs are more optimized to run highly dimensional computations. Hence, TensorFlow seems to map to GPU kernels most of the operations whose computational complexity is affected by the batch size.

Second, the GPU execution time increases with the batch size. LeNet-5 goes from an average of $6\mu s$ to $180\mu s$ of GPU execution time when increasing the batch size from 1 to 8192 images. The average GPU execution time of ResNet-50 goes from $12\mu s$ to $500\mu s$ when increasing the batch size from 1 to 128 images. Finally, BERT goes from an average of $6\mu s$ to $200\mu s$ when increasing the batch size from 1 to 2048 sentences. This supports our previous observations, as larger batch sizes induce larger kernel runs, which results in relatively more GPU execution time. This is consistent with previous observations from the state-of-the-art [81, 82].

Third, the enqueuing time is shorter than the dequeuing time, independently of the batch size. Throughout all the workloads, the average dequeuing time is around $2\times$ the average enqueuing time.

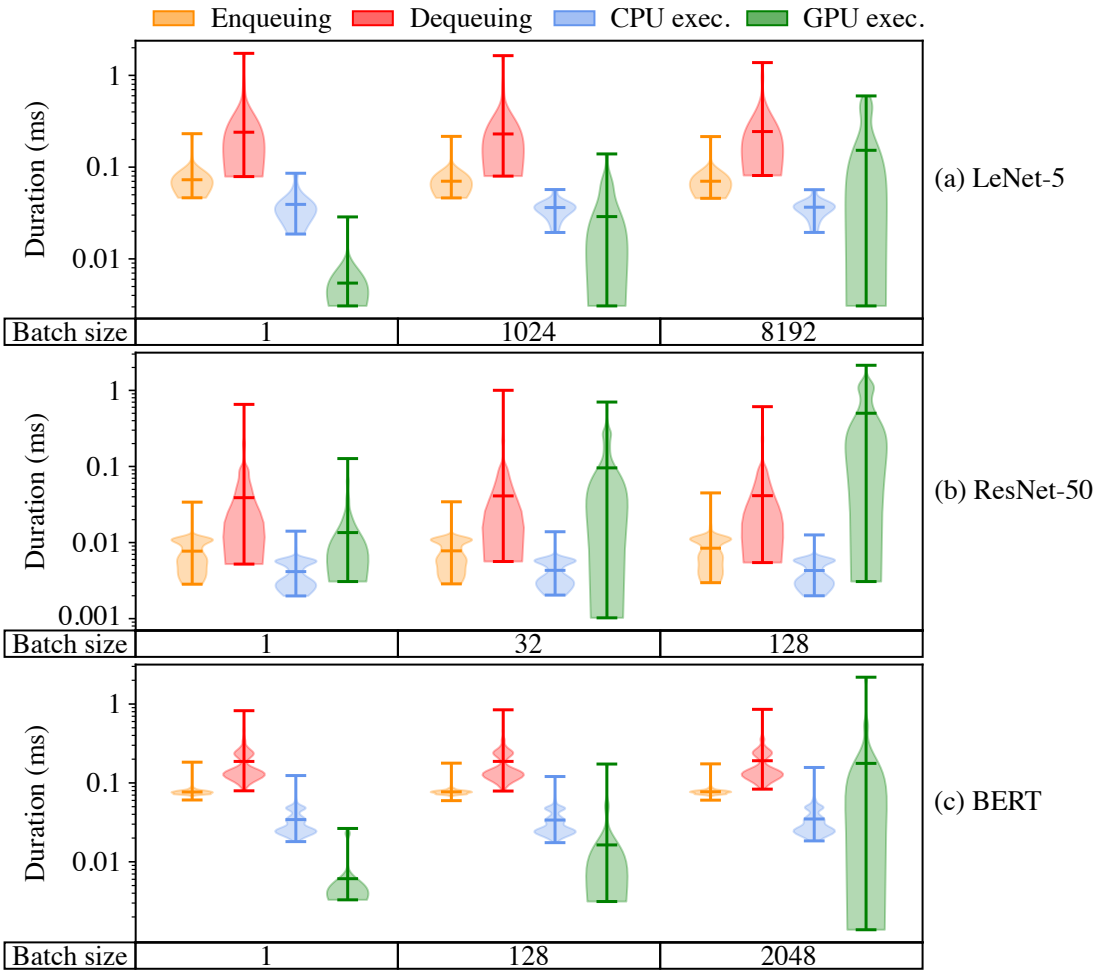


Figure 3.5: *Distribution of TensorFlow eager execution time grouped by execution phase for LeNet-5 (a), ResNet-50 (b), and BERT (c) inference on a CPU-GPU tandem*

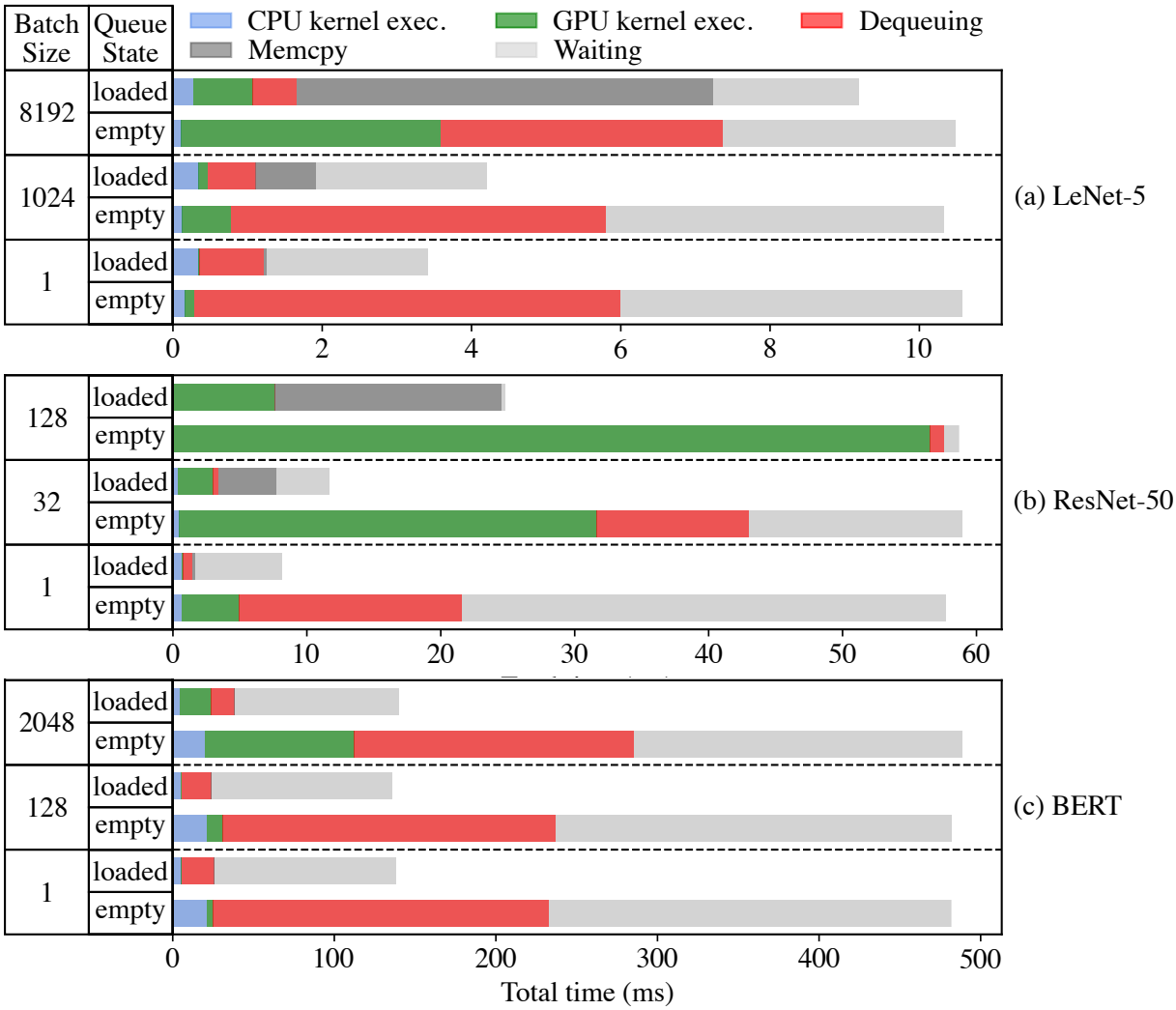


Figure 3.6: *Execution time distribution with respect to the utilization of the scheduling queue*

3.4.4 Utilization of the scheduling queue

We study the utilization of the scheduling queue and the corresponding execution time distribution across five categories: GPU kernel execution time, CPU kernel execution time, dequeuing time, memory transfer time, and waiting time (i.e., rest of the execution). When several categories apply simultaneously, we prioritize the first category in this list. For example, we classify the concurrent time of GPU kernel execution and dequeuing as GPU kernel execution time. Fig. 3.6 shows the execution time distribution with respect to the utilization of the scheduling queue for LeNet-5 (a), ResNet-50 (b), and BERT (c). We make three observations.

First, the utilization of the scheduling queue of the CNN workloads increases when increasing the batch size. The ResNet-50 goes from $8ms$ of queue utilization time to around $25ms$ ($3.1\times$) when increasing the batch size from 1 to 128. The LeNet-5 queue

utilization also grows $3\times$ when increasing the batch size from 1 to 8192. In contrast, the utilization of the queue is minimally affected by the batch size in the case of BERT.

Second, the scheduling queue is empty during most of the execution time. BERT exhibits an empty queue for 72% of its execution time in all batch sizes. ResNet-50 has an empty queue for 86% (58%) of the time for a batch size of 1 (128) images.

Third, the waiting time represents a lower portion of the total execution time when increasing the batch size. BERT and ResNet-50 spend 10% and 31% less time waiting when increasing their batch sizes from 1 to 2048 and 128, respectively. However, we observe an exception with LeNet-5: when the queue is loaded, the time spent waiting for a node to be enqueued represents 64% for a batch size of 1 and 82% for a batch size of 8192 images.

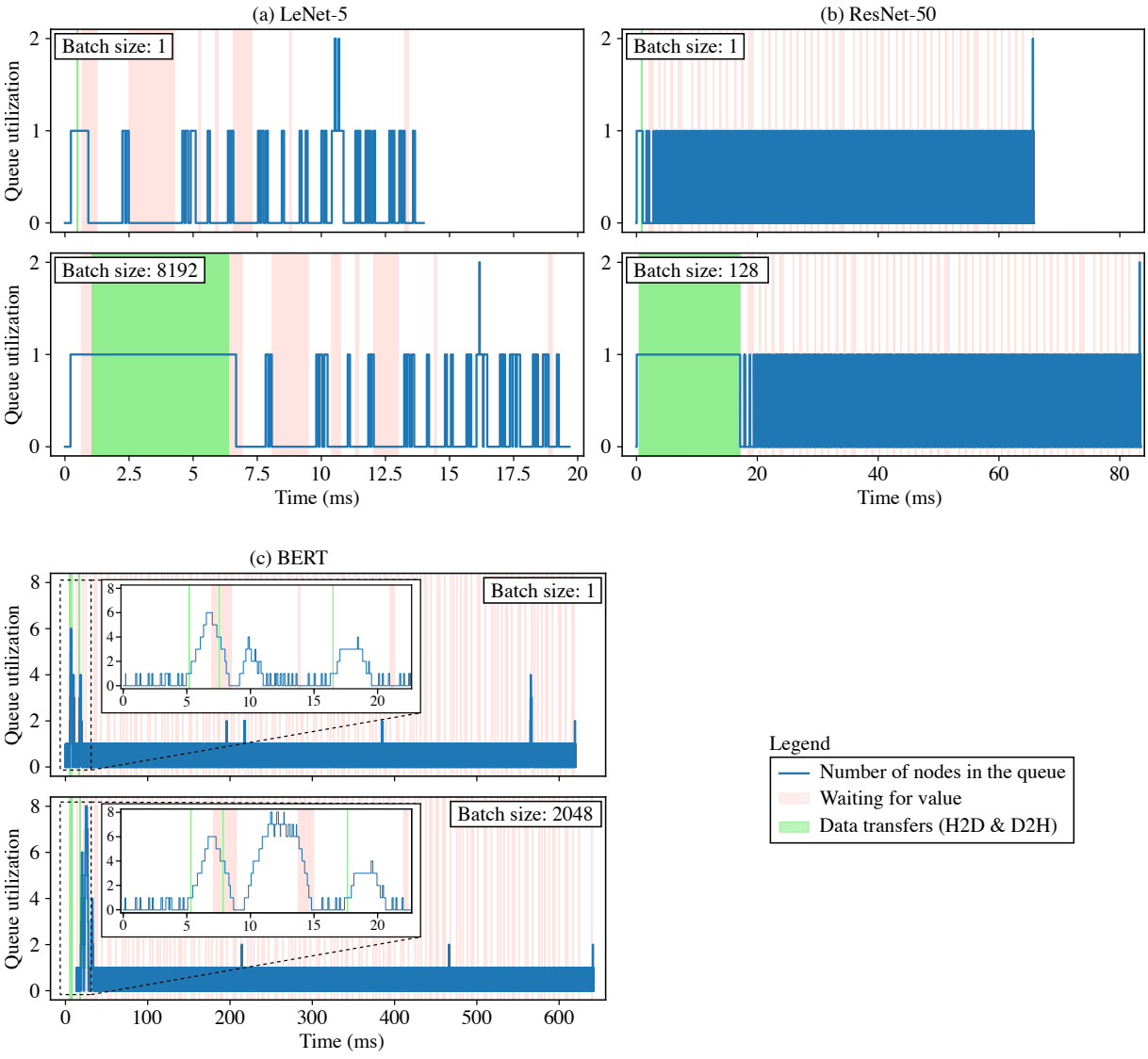


Figure 3.7: *Utilization of the scheduling queue over time for LeNet-5 (a), ResNet-50 (b), and BERT (c) inference on a CPU-GPU tandem*

To further understand the observed behaviors, Fig. 3.7 shows the utilization of the scheduling queue for LeNet-5 (a), ResNet-50 (b), and BERT (c) inference on a CPU-GPU tandem. We add annotations to the figures to indicate when the enqueuing thread is waiting for values to be computed and to show the data transfer timings. We make two additional observations.

First, the LeNet-5 execution spends a considerable amount of time on data transfers, which can explain the previously described behavior of spending a lot of time waiting when the queue is loaded. Indeed, when subtracting the data transfer time from the waiting time, the remaining waiting time represents 64% with a batch size of 1 and only 28% with a batch size of 8192 images.

Second, workloads spend most of the time waiting for values to be computed. This limits the opportunities for the host program to advance enqueuing in parallel with kernel executions. The zoomed part on Fig. 3.7c shows that the dequeuing thread empties the queue while the enqueuing thread waits for a value to be computed. Overall, while BERT uses the queue more than the two tested CNNs, the queue seems to be under-utilized most of the time during inference.

3.5 Related works

To the best of our knowledge, this work is the first to provide a detailed description and analysis of the TensorFlow eager execution runtime. In this section, we identify two areas of related works that are relevant to our contributions.

On the one hand, there have been efforts around providing ML profiling platforms. However, these works are either dedicated to specific workloads [58] or evaluate the framework overhead without analyzing the execution of the runtime in detail [90, 91]. Our work is complementary to these contributions, as it provides a clear description of an ML framework mechanism. The new insights can be used on top of the ones provided by current profiling platforms.

On the other hand, popular ML frameworks provide specific profiling tools for developers to analyze the performance of their model [100, 124, 149], and optimize the performance from a high level. However, these tools combine application-level analysis with low-level information, which is often difficult to interpret by an average user. They do not include an accessible analysis of the performance of the inner mechanism of their specific frameworks. In this chapter, we aim to bridge this gap and provide new accessible insights on the TensorFlow framework runtime.

3.6 Concluding thoughts and summary

In this chapter, we provide a detailed analysis of the TensorFlow eager execution runtime, we have reviewed the key concerns applicable to any execution mode (e.g., hiding the framework latency, maximizing utilization of the accelerator). Hence, the principles demonstrated through our analysis can also be applied to other execution modes. While variations exist between these different modes, the foundational concepts remain consistent. For example, different decisions could be made regarding which parts of the execution plan are resolved at design time instead of run time, or regarding the scope of the optimizations. However, the main optimization objectives will remain the same: the ML framework has to be as little invasive as possible regarding the execution time and execute faster to help maximize the utilization of the device (here, the GPU).

To this end, in this first chapter, we provide a detailed description of the TensorFlow eager execution runtime and propose new metrics to analyze its performance overhead. This contribution is presented in three main parts.

First, in section 3.2, we describe the main steps followed by the TensorFlow eager execution runtime to run code on a CPU-GPU tandem. We show how the runtime can leverage a scheduling queue to parallelize some of the long-running operations needed to run inference on a CPU-GPU tandem. We also use a simple illustrative example to show this mechanism in action.

Second, in section 3.3, we identify potential bottlenecks in the TensorFlow eager execution runtime and propose new metrics to analyze the performance overhead of the ML framework. These metrics help identify the time spent executing kernels, the distribution of the execution time between the main phases of the runtime execution, and the effectiveness of the scheduling queue to hide the runtime overhead.

Finally, in section 3.4, we use our described approach to conduct in-depth profiling of the inference process of two CNNs (LeNet-5 and ResNet-50) and a Transformer-based model (BERT) for different batch sizes. Our results show that the runtime overhead of the ML framework is reduced considerably when operating with larger CPU and GPU kernels. However, we also show that the overhead could become significant when GPU kernel execution is not long enough to hide the framework's runtime latency. We believe that this work highlights the need to better understand ML framework's bottlenecks.

CHAPTER 4

Multi-level Analysis of GPU Utilization in ML Training Workloads

Contents

4.1 Context and motivation **48**
4.2 Efficient ML training loop execution on GPU **50**
4.3 Proposed profiling methodology **53**
4.3.1 Profiling scope 53
4.3.2 Gathering high-level and low-level metrics 54
4.3.3 Coherent integration of traces 54
4.4 Experimental setup **55**
4.5 Results **57**
4.5.1 Overall performance vs. memory utilization 57
4.5.2 GPU compute resource utilization 61
4.5.3 Implications and Insights 66
4.6 Related works **66**
4.7 Summary **69**

4.1 Context and motivation

In the previous chapter, we presented a deep dive into how modern Machine Learning (ML) frameworks execute the inference process of ML models. While inference is the most frequent use case for users of ML applications, training is the main application driving the evolution of modern ML accelerators. Although extensive research is conducted in new specialized accelerator design [126], Graphics Processing Units (GPUs) remain the prevailing architecture when it comes to training ML models. As mentioned in Chapter 1, the GPU architecture has evolved over the years to improve latency and throughput for different types of workloads. For example, NVIDIA added a cache hierarchy to the Fermi architecture in 2010 [108], (enabling hardware-automated data reuse) and tensor cores to the Volta architecture in 2017 [109] (accelerating matrix multiplications [141]). However, throughput and latency improvements can be achieved through brute-force approaches. For example, increasing the number of cores or the memory bandwidth can improve the overall performance of the GPU (i.e., scaling up). Such scaling up can lead to a less well-balanced use of the computing resources across the architecture (e.g., underutilized cores). Thus, wasting area/cost and inducing energy overheads.

Designing more compute-efficient ML accelerators relies on an accurate understanding of how efficiently ML training workloads use the computing resources of modern architectures, such as GPUs. However, the design of modern GPUs is intricate and proprietary. In addition, as detailed in Chapter 3, the interactions between GPUs and ML frameworks rely on complex runtimes and optimized closed-source libraries [34]. This makes gathering performance metrics tedious as it requires using multiple profiling tools across different abstraction layers to get a complete picture of the execution. Hence, design decisions are often based on *high-level* or *low-level* metrics. On the one hand, high-level metrics (e.g., GPU utilization) can be misleading because they may not reflect the utilization of the internal components of the GPU architecture. On the other hand, low-level metrics (e.g., tensor cores utilization) cannot capture the efficiency of the host/device interactions happening at a higher level.

Using ML frameworks profilers, such as the TensorFlow Profiler [148], we can measure the GPU utilization and the throughput of the training process. GPU utilization is defined as the proportion of time the GPU is actively used (i.e., using GPU kernels) during training time. One could argue that a higher GPU utilization is desirable, as it signifies that the training process can consistently harness the acceleration potential of the GPU architecture. In practice, however, certain training processes can attain shorter training times by compromising GPU utilization. As an illustration, let's con-

sider one training loop of ResNet-50 running on a CPU-GPU tandem. Fig. 4.1a shows the GPU utilization and throughput when training ResNet-50 on the NVIDIA A100 GPU with different batch sizes, using Full Precision (FP) and Mixed Precision (MP) training. Here, we can observe that while FP achieves better GPU utilization with small batch sizes compared to MP, this difference gets lower when using large batches. Looking at throughput (normalized by highest achieved value), we can observe that while FP saturates throughput around 64 images per batch, MP scales much better and achieves higher throughput for large batches. Fig. 4.1b shows the relative gains in GPU utilization and throughput when using MP over FP for the same training loop and batch sizes. This second representation of the results makes it easier to see the relative gains that MP brings to GPU utilization and throughput. For GPU utilization, relative gains are always negative, down to around -15% for batch sizes of 16 and 32 images. However, looking at throughput, positive gains start to appear when using MP above 16 images per batch and rise to $+39\%$ for batches of 512 images. Hence, this shows that some software optimizations (e.g., MP) can sacrifice some GPU utilization to achieve far better throughput.

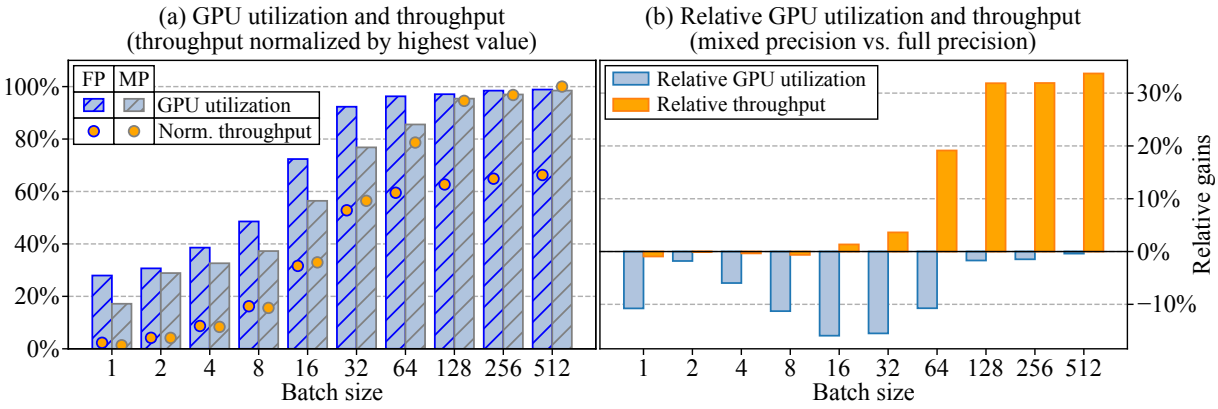
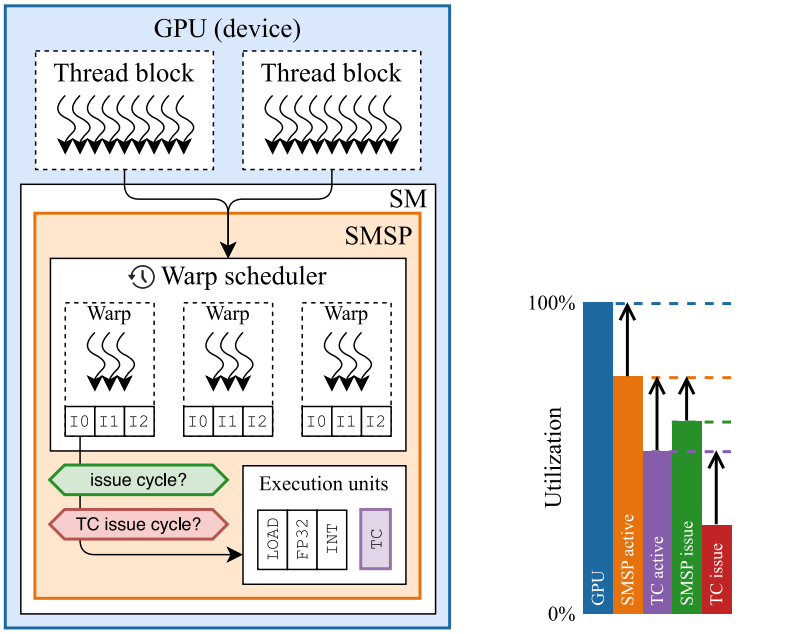


Figure 4.1: *GPU utilization and throughput training ResNet-50 with different batch sizes.*

Our **main goal** in this chapter is to evaluate how efficiently ML training workloads use the computing resources of modern GPUs.

To this end, we propose a multi-level analysis of GPU computing resource utilization for ML training workloads. We first describe an ideal reference execution of a GPU-accelerated ML training loop and identify relevant metrics that can be measured using existing profiling tools. Then, we propose a methodology that combines traces from these profiling tools to evaluate the utilization of the GPU computing resources at different levels of abstraction. This methodology circumvents the limitations of the state-of-the-art by gathering coarse-grain and fine-grain metrics to propose a multi-level view of GPU utilization. Hence, this methodology allows for new insights into the execution of ML training on modern GPU architectures. Thus, we implement this

methodology on two modern GPUs (i.e., NVIDIA V100 and A100). Finally, we analyze the impact of different software optimizations (e.g., mixed-precision, various ML frameworks, and execution modes) on throughput, memory usage, and multi-level utilization of the GPU resources.



(a) Different abstraction levels in the GPU architecture

(b) Utilization bounds for the different abstraction levels

Figure 4.2: *Multi-level utilization of the GPU computing resources*

4.2 Efficient ML training loop execution on GPU

In this section, we describe a reference execution as an ideal GPU-accelerated ML training loop execution. We analyze the execution at different levels of hardware abstraction (i.e., GPU, Streaming Multiprocessor (SM) and Tensor Core (TC)). For each abstraction level, we identify metrics that can be measured to evaluate the utilization of the GPU computing resources. Fig. 4.2 illustrates (a) the different abstraction levels considered in our analysis and (b) the boundaries of the utilization for each level.

GPU level. As defined by the ML framework profilers [122, 124, 148], *GPU utilization* is the ratio time executing GPU kernels over the total execution time. The time during which no kernel is being executed is the *ML framework overhead*. At runtime, for each GPU-compatible operation that has to be executed, the ML framework has to launch one or multiple kernels on the GPU. Before launching the kernels, the framework needs to execute other enabling tasks on the CPU, such as parsing the Python code, copying data between CPU and GPU, compiling the kernels (i.e., Just-In-Time (JIT) compilation), launching the kernels, synchronizing the GPU, etc. While the CPU can

also execute kernel operations for the ML model, we show in Chapter 3 that, when a GPU is available, the CPU is mainly used to run these enabling tasks. Hence, all CPU time not hidden by the GPU execution time can be considered as ML framework overhead. Ideally, this setup time would be entirely hidden by the parallel execution of previously launched GPU kernels. However, in practice, the setup time creates bubbles of GPU inactivity (*see Fig. 4.3a*), reducing GPU utilization.

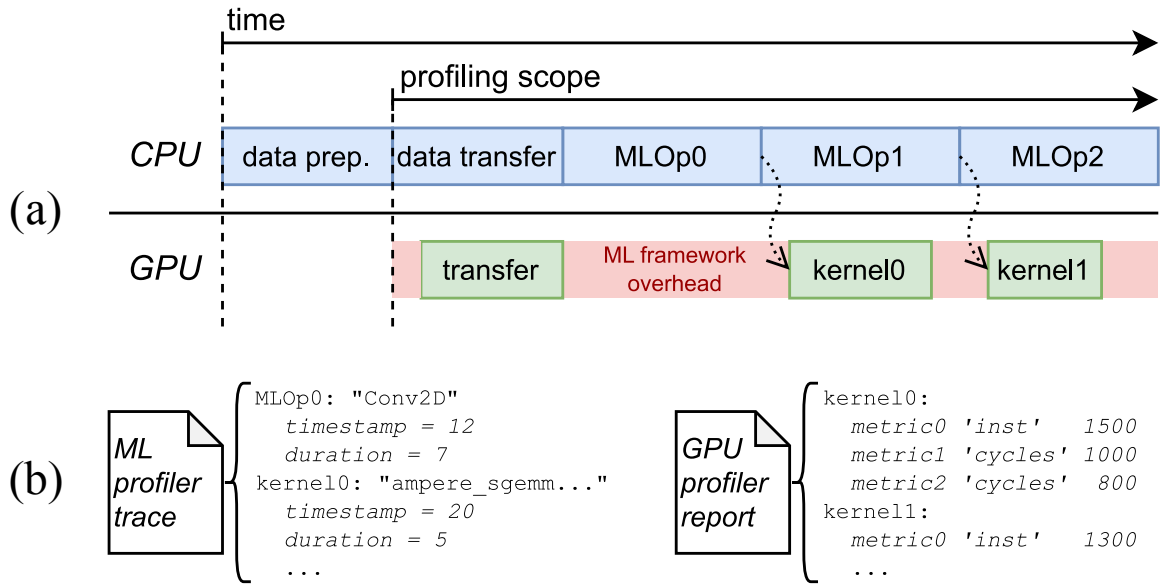


Figure 4.3: *(a) Typical eager execution trace; (b) CPU and GPU events are listed in the ML profiler trace file; GPU's performance counters are listed in the GPU profiler report*

GPU utilization can increase either by extending kernel execution time or reducing the ML framework overhead. In practice, a common and easy way to extend the kernel execution time without changing the model is to increase the batch size of the training workload. While changing the batch size can impact how fast the model converges during training, this can be mitigated by adjusting the learning rate accordingly [134].

However, an increased batch size results in GPU kernels operating on larger data. Thus, the available GPU memory limits the maximum batch size that can be used. Typically, increasing the batch size has a negligible impact on the framework overhead as the same number of kernels are launched. Reducing the ML framework's overhead can be done using more optimized modes of execution than eager (e.g., JIT compile). As described in Chapter 2, these modes leave the flexibility and debugging straightforwardness of Python [62] (e.g., line-by-line debugging, high-level code interpretation) to enable high-level code transformations or optimizations across multiple operations of the model (e.g., dead code elimination, constant folding, arithmetic simplification, kernel fusion).

As shown in Section 4.1, GPU utilization alone can be misleading when it comes to evaluating performance, as it only shows how well the setup time is hidden by the execution of GPU kernels. *Throughput*, defined as the number of input samples processed per second, is a better metric to compare different workloads, but it lacks specificity in pinpointing performance bottlenecks. By looking at GPU utilization together with the achieved throughput, we can compare the performance of different workloads from a high-level perspective. However, this high-level analysis does not provide insights into the utilization of the inner computing resources of the GPU architecture. Hence, metrics have to be gathered at a finer granularity, using performance counters for events happening at the microarchitectural level.

SM level. When a GPU kernel is launched, its threads are grouped in thread blocks and automatically distributed across multiple SMs by the thread block scheduler. The scheduler keeps SM busy by distributing the thread blocks evenly (i.e., load balancing). However, during the beginning and end of the execution, some SMs may run out of thread blocks to execute. This can lead to SM underutilization [120]. At a given cycle, we consider an SM as active if it has at least one warp to execute (i.e., a set of 32 threads executing the same instruction). SMs of modern NVIDIA GPUs are composed of four subpartitions. For finer granularity, we choose to evaluate utilization at Streaming Multiprocessor Sub-Partition (SMSP) level. We define *SMSP active utilization* as the number of active SMSP cycles over the total number of elapsed cycles on the GPU. Hence, the difference between GPU utilization and SMSP active utilization evaluates the ability of the GPU schedulers (thread block scheduler and warp scheduler) to effectively parallelize threads across SMSPs.

An active SMSP does not necessarily imply that warp instructions are being issued. Indeed, the warp scheduler (internal to the SMSP) can stall the issuing of a warp instruction for different reasons (e.g., waiting for data to be fetched from memory, or waiting for a dependent instruction to finish execution). A cycle during which the warp scheduler is stalled (i.e., cannot issue any instruction to the GPU cores) is called *stall cycle*. Otherwise, the cycle is called *issue cycle* (i.e., the warp scheduler issues an instruction to the GPU cores). We define the *SMSP issue slot utilization* as the ratio of issue cycles over the total number of elapsed cycles on the SMSP. In modern GPUs, each SMSP can issue one warp instruction per active cycle. Hence, the SMSP active utilization is an upper bound for the SMSP issue slot utilization (see Fig. 4.2b).

Tensor core level. Tensor cores (TC) are specialized GPU cores designed to accelerate matrix multiplications. As a result, peak GPU FLOPS performance can only be achieved through the use of tensor cores. For example, the NVIDIA A100 can achieve 19.5 TFLOPS executing FP32 operations with classic FP32 cores, but when using ten-

sor cores, it can achieve 156 TFLOPS (and up to 312 TFLOPS when exploiting sparsity [104]).

Ideally, maximizing the *tensor core utilization* relies on three conditions. First, the number of TC active cycles should dominate the total number of active SMSP cycles (i.e., SMSP active utilization). As tensor cores are part of the SMSP, SMSP active utilization is an upper bound for TC active utilization. Second, the number of TC instructions should dominate the total number of issued instructions (i.e., SMSP issue slot utilization). As tensor cores are also limited to one instruction per cycle, TC active utilization is an upper bound for TC issue slot utilization. Finally, the issued TC instructions should achieve peak throughput of the tensor cores. Due to the latency and instruction bandwidth of the tensor cores, this last condition is frequently not met [141].

4.3 Proposed profiling methodology

In this section, we describe our profiling approach to evaluate the utilization of GPU computing resources at different abstraction levels. We first describe the scope of our profiling analysis and give insights into the expected profiling overhead. Then, we detail how to gather the previously described metrics using existing profiling tools. Finally, we propose a systematic methodology to combine the gathered metrics from the different profiling tools to analyze the execution of the ML training workload.

4.3.1 Profiling scope

Training an ML model can take hours or even days, depending on the complexity of the model and the size of the training dataset. While the first few iterations might have bigger variations in the operations to execute (i.e., due to initialization steps in the software), the greater proportion of the training is more stable and comprises thousands of identical iterations where the ML model is trained on different batches of data from the training dataset (see Chapter 2).

Thus, profiling the complete training workload is impractical due to the profiling overhead added by the profiling tools. For example, as described in Chapter 2, when gathering low-level metrics using NVIDIA Nsight Compute [114], kernels might have to be executed multiple times to gather all the required metrics (i.e., kernel replay). To circumvent this, we limit our scope to a single iteration. To select the target iteration, we preliminary run multiple iterations of the training workload. From this

preliminary run, we select the first iteration with a stable execution time, thus ignoring the iterations that include initialization overheads. These overheads are negligible in comparison with the overall training time and do not reflect the characteristics of an average training iteration.

Before starting the training, the ML framework could also execute data preprocessing steps on the Central Processing Unit (CPU) (*see Fig. 4.3a*). As we are interested in the efficiency of the GPU architecture, we ignore these steps. This way, our profiling scope still includes the CPU time spent orchestrating the GPU execution (i.e., the GPU utilization is not always at 100%), in addition to the GPU kernel execution time itself.

4.3.2 Gathering high-level and low-level metrics

We gather high-level metrics from ML framework profilers and low-level metrics from the GPU kernel profiler. The ML framework profilers (i.e., TensorFlow Profiler [148] or PyTorch Profiler [124]) provide the list of events as JSON files that we parse to gather our metrics of interest. We list the selected high-level metrics at the top of Table 4.1. Instead, the GPU kernel profiler (i.e., NVIDIA Nsight Compute [114]) provides a report file that lists all the executed kernels during the profiled iteration along with the values of the GPU performance counters. Hence, we aggregate the low-level metrics from the different kernels to evaluate the utilization of the GPU computing resources. For example, SMSP elapsed cycles are the sum of the cycles elapsed at the SMSP level for all the kernels. We list the selected low-level metrics at the bottom of Table 4.1.

Note that the tensor core metrics that we gather only cover two of the three conditions required to achieve peak tensor core utilization (described in Section 4.3). This limitation is because the GPU profiler only provides the total number of instructions that have been issued to the tensor cores, without distinguishing between their different types and achieved FLOPS. Hence, we evaluate tensor core utilization between two bounds: (1) the ratio of tensor core instructions over the number of issued instructions (i.e., TC issue slot utilization), and (2) the ratio of active tensor core cycles over the amount of active SMSP cycles (i.e., TC active utilization).

4.3.3 Coherent integration of traces

Analyzing the profiled metrics for the complete training iteration is straightforward using the selected metrics. In fact, by matching the GPU kernel execution time (i.e., high-level metric) with the SMSP elapsed cycles (i.e., low-level metric), we can eval-

uate the lower-level utilization of the GPU computing resources (i.e., SMSP and TC utilization) relative to the high-level utilization (i.e., GPU utilization). However, when it comes to analyzing the profiled metrics at a finer granularity of ML model architectures (e.g., per layer type), linking the high-level and low-level metrics becomes more challenging. On the one hand, the ML framework profiler does not gather the low-level metrics from the GPU performance counters for each kernel. On the other hand, the GPU kernel profiler cannot provide neither kernel timings, nor any information regarding the ML framework runtime. Hence, GPU kernels have to be matched between both profiler traces (i.e., ML profiler trace and GPU profiler report).

Previous works have done this matching by running a second tracer tool in parallel with the ML profiler [91]. This approach relies on manually implementing support for a tracing library (e.g., CUPTI [111]) with the target application. This allows to observe the profiling results in real-time but may introduce additional overheads on the profiled system. In contrast, our approach leverages existing profiling tools (i.e., Nsight Compute [114]) and can be performed without any adaptation to the target application. We propose to match the GPU kernels offline, after the gathering of the profiling traces. This is done by comparing kernel metrics that are common between both profiling tools (e.g., kernel name, block and thread dimensions, kernel duration, memory usage, etc.). Based on these metrics, we establish a correlation matrix between the kernels from the ML profiler trace and the kernels from the GPU profiler report. We then use this correlation matrix to identify the corresponding kernels and combine both high-level and low-level metrics. Hence, instead of matching kernels only using timestamps (which can be rendered useless in the case of kernel replay [40]), this approach provides a stronger matching between the kernels as it verifies multiple execution parameters for each kernel.

In addition to matching the kernels between the two traces, we also programmatically analyze the ML profiler trace to identify the different layers of the ML model and annotate the kernels accordingly. This allows us to group the kernels by layer type and evaluate the utilization of the GPU computing resources at a finer granularity.

4.4 Experimental setup

Workloads. We choose three well-known supervised Deep Learning (DL) workloads (i.e., ResNet-50, BERT, and DLRM) from the MLPerf training benchmark suite [95], which is a widely used benchmark suite for DL training workloads. These models

Table 4.1: *High-level (top) and low-level (bottom) metrics*

Name	Description
Total execution time	Total duration of the traced iteration
GPU kernel exec. time	Aggregate duration of all the GPU events
GPU utilization	GPU kernel exec. time / Total execution time
Achieved throughput	Batch size / Total execution time
SMSP elapsed cycles	Elapsed GPU cycles counted at SMSP level
SMSP active cycles	SMSP cycles with at least 1 warp active
SMSP active utilization	SMSP active cycles / SMSP elapsed cycles
SMSP issue cycles	SMSP cycles with an instruction issued
SMSP issue slot utilization	SMSP issue cycles / SMSP elapsed cycles
TC active cycles	SMSP cycles with at least 1 active tensor core
TC active utilization	TC active cycles / SMSP elapsed cycles
TC issue cycles	Number of tensor core instructions issued
TC issue slot utilization	TC issue cycles / SMSP elapsed cycles

are representative of common ML model architectures such as Convolutional Neural Network (CNN), Transformer-based, and recommendation models, respectively. ResNet-50 [63] and BERT [50], introduced in Chapter 3, are tried-and-tested models in the research community and are used for image classification and Natural Language Processing (NLP) tasks, respectively. DLRM [103] is a state-of-the-art model developed by Meta (Facebook) that is used as a personalization and recommendation system for applications such as search ranking, feed ranking, ads/video/content recommendation, etc. We use PyTorch (PT) and TensorFlow (TF) as reference ML frameworks, due to their maturity. ResNet-50 (PT), BERT (PT and TF), and DLRM (TF) implementations are downloaded from the NVIDIA repository [107]. DLRM (PT) implementation is downloaded from the MLPerf Training benchmarks reference models. ResNet-50 (TF) implementation is provided directly inside the TensorFlow library [158]. All of the training parameters are chosen following the MLPerf Training benchmarks rules to ensure a fair comparison between the workloads. Our results show less than 1% standard deviation in execution time across three runs, for each workload.

Software optimizations. We use PyTorch and TensorFlow with two different execution modes: eager execution and JIT compilation. For eager execution, we use the respective default eager execution runtimes of both ML frameworks. For JIT compilation, we use XLA JIT for TensorFlow workloads and the TorchScript JIT backend for PyTorch workloads (i.e., the respective default JIT compilation backends). We run each mode using both FP and MP [123, 147].

GPUs. We use NVIDIA V100 and A100 GPUs, from Volta (2017) and Ampere (2020) architectures, respectively. The GPUs are paired each with an AMD Milan EPYC 7543 CPU with 32 cores and a 2.8 GHz clock. Despite both of these GPU architectures being equipped with tensor cores, the V100's tensor cores can only execute half-precision

matrix-multiplications. Hence, the full precision workloads we run on the V100 GPU will not be able to take advantage of tensor cores. Table 4.2 lists the main GPU features.

Profiling tools. We use TensorFlow Profiler [148] and PyTorch Profiler [124] to gather high-level metrics and NVIDIA Nsight Compute [114] to gather low-level metrics. As described in Section 4.3.1, we profile one training iteration for each workload. We select the iteration when the training is in a steady state, without any one-time initialization overhead (e.g., data preprocessing, data loading, model initialization).

Table 4.2: *Main GPU features*

	NVIDIA A100	**NVIDIA V100**
Architecture	Ampere	Volta
SMSPs	432	320
DRAM Memory	80 GiB	32 GiB
Tensor Cores (Peak TFLOPS)	432 (312)	640 (125)

4.5 Results

In this section, we measure and analyze GPU compute efficiency from two perspectives. First, from a high-level analysis, we analyze *performance vs. memory utilization* (4.5.1). Then, leveraging the proposed profiling methodology that combines multiple tools, we analyze the *compute resource utilization* (4.5.2). Finally, we draw some general implications and insights considering both analyses (4.5.3).

4.5.1 Overall performance vs. memory utilization

Here, we evaluate the raw performance of the workloads by measuring training throughput and the allocated GPU memory. These two metrics can easily be obtained using the high-level profiling tools provided by the ML frameworks. We use these metrics to characterize the tested workloads and to understand the trade-offs between performance and memory utilization. We compare the performance of the workloads across different batch sizes, ML frameworks, and execution modes.

GPU memory & Batch size. We evaluate raw performance by measuring training throughput, defined as the amount of processed items per second. We use this metric as it is commonly used by industry and research-leading benchmarks. For example,

participants (i.e., companies or research labs) submit their training time (inversely proportional to throughput) results for reference ML models to MLPerf [95] to showcase the performance of their hardware/software stack.

Fig. 4.4 shows the normalized throughput of running ResNet-50, BERT, and DLRM with various batch sizes (annotated on the markers) and ML framework options. Fig. 4.4 also shows the corresponding GPU memory allocated (x-axis) for a given batch size, using both the NVIDIA A100 and V100 GPUs. For clarity, only batch sizes using $\geq 5\%$ of the respective GPU memory are shown (i.e., ≥ 4 GB for the A100 and ≥ 1.6 GB for the V100). In this chapter, we mainly comment on a subset of the tested workloads as an illustration (i.e., ResNet-50 using the A100 Fig. 4.4a and using the V100 4.4b). However, while the other workloads follow a similar trend, we comment on the major differences between them.

We can observe that both the amount of GPU memory allocated and the throughput increase with growing batch sizes. For example, training ResNet-50 using the A100 (see Fig. 4.4a), transitioning from batches of 128 to 2048 images, TF XLA JIT with mixed precision achieves a $3.8\times$ increase in throughput, while also allocating around $15\times$ more GPU memory. However, while the GPU memory allocated grows linearly with the batch size, throughput saturates when approaching the biggest batch sizes. For the same example, going from the second biggest batch size (1024) to the biggest batch size (2048) only increases throughput by 3% for mixed precision and 1% for full precision, while still doubling ($1.98\times$) the allocated GPU memory.

When using the V100 (see Fig. 4.4b), which has less memory capacity than the A100 (i.e., 32GB vs. 80GB, respectively), this throughput saturation is not as drastic. In comparison to the A100, using TF XLA JIT with the V100 still achieves a 58% increase in throughput when using full precision, going from the penultimate to the last batch size (128 to 256 images). Even when using mixed precision, the increase in throughput is still 40% going from the penultimate to the last batch size. These observations for ResNet-50 are consistent across all tested workloads. These results suggest that the addition of more memory capacity between the two GPU generations (V100 to A100) enables the A100 to saturate throughput, eliminating any bottleneck due to limited GPU memory.

Key takeaway 1. *The A100 provides enough memory to saturate throughput, eliminating utilization gaps caused by limited memory capacity and the push for larger batch sizes.*

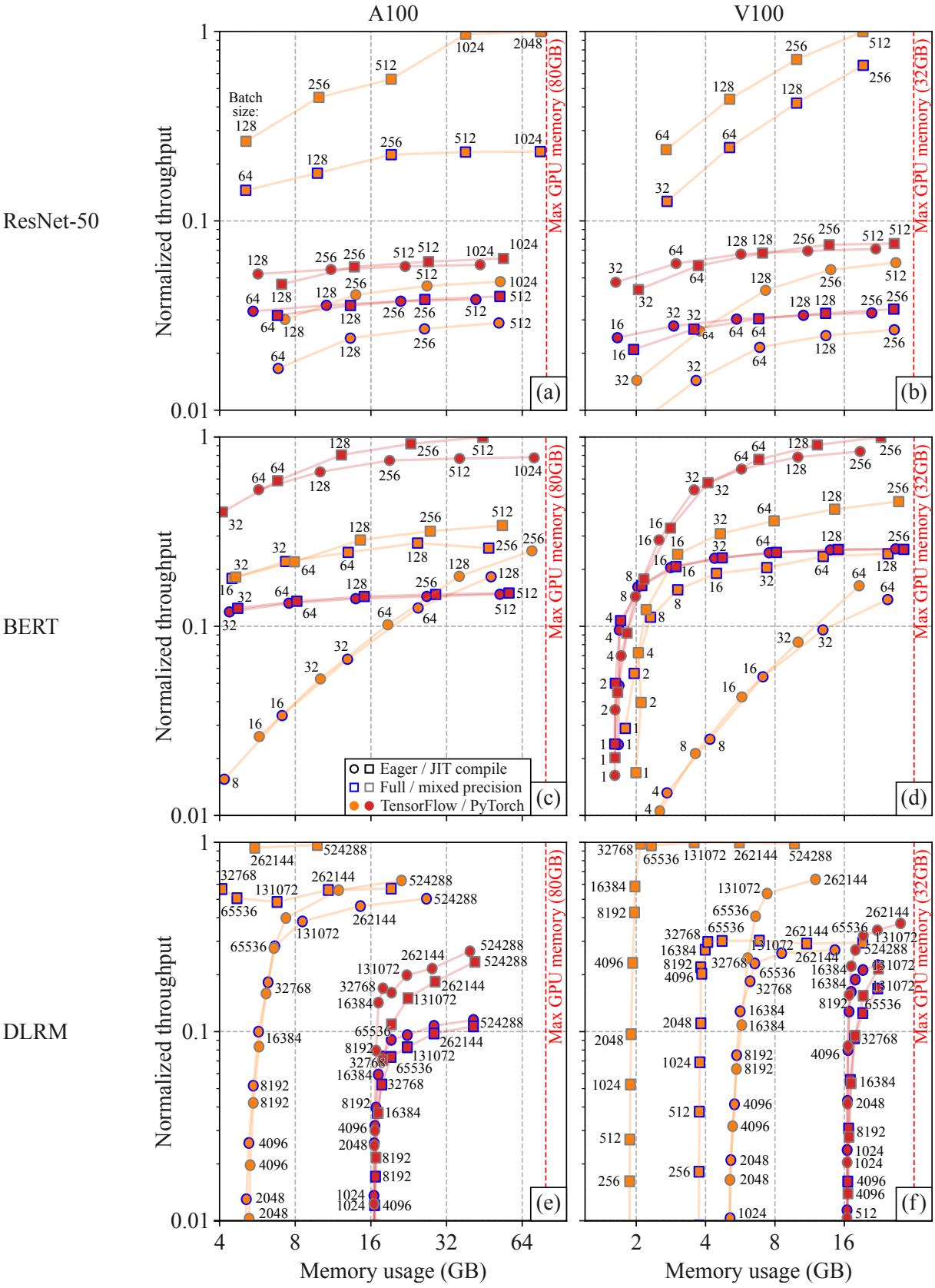


Figure 4.4: *Normalized throughput vs. GPU memory usage for ResNet-50, BERT, and DLRM on the NVIDIA A100 and V100 GPUs with different software optimizations. Batch size is annotated on the markers.*

ML framework execution modes. We compare memory usage and throughput across eager execution (i.e., default execution) and JIT compilation for both TF and PT. Fig. 4.4 shows different marker shapes for eager execution (round markers) and JIT compilation (square markers).

We make three observations. First, TF XLA JIT allocates less GPU memory than TF eager for the same batch size, which increases throughput (as discussed in our first takeaway). For example, running a TF eager execution requires 65% of the A100's memory for a batch size of 512, whereas XLA JIT only requires 48% of the memory for the same batch size. As a result, XLA JIT can fit larger batches within the GPU memory. Second, XLA JIT can achieve $8\times$ ($25\times$) higher throughput than TF eager execution in the A100 (V100). Finally, this reduction in memory usage and increase in throughput is not as drastic when comparing PT eager and PT TorchScript JIT. In fact, PT TorchScript JIT requires more memory for a given batch size and achieves higher throughput only for large batch sizes, compared to PT eager execution.

When comparing precision modes, we can make two observations. First, mixed precision (markers with gray outline) consistently uses around 50% of the memory allocated by full precision (markers with blue outline). This enables to fit larger batches in the same amount of memory, which increases throughput compared to full precision. In addition, mixed precision provides a boost in throughput for the same respective batch sizes as full precision. For example, using PT, mixed precision achieves around $1.5\times$ the throughput of full precision for the same batch sizes using the A100, and $2\times$ using the V100. This boost in throughput is highest when using TF XLA JIT on the A100 with the biggest batch sizes (i.e., 1024 and 2048), reaching $4.2\times$. These observations are consistent across all tested workloads.

Key takeaway 2. *JIT compilation and mixed precision increase throughput by fitting larger batch sizes in GPU memory.*

When comparing ML frameworks, we observe that PT eager execution uses less memory and achieves better throughput than TF eager, particularly with large batch sizes. For example, using full precision with a batch size of 512, PT eager achieves $1.3\times$ the throughput of TF eager and uses 12% less memory. However, when using JIT compilation, TF XLA JIT outperforms every other mode of execution, achieving the best throughput across all batch sizes and using less memory. For example, using full precision with a batch size of 512, TF XLA JIT achieves $5.8\times$ the throughput of PT TorchScript JIT and uses 17% less memory. While TF XLA JIT outperforms PT TorchScript JIT for ResNet-50 and DLRM, PT TorchScript JIT outperforms TF XLA JIT for BERT (see Fig. 4.4c and Fig. 4.4d).

Key takeaway 3. *Generally, TF XLA JIT significantly outperforms PT TorchScript JIT in both throughput and memory.*

While the high-level results presented here provide insights into the performance and memory utilization of the evaluated workloads, they do not provide a detailed view of the GPU compute resource utilization. Next, we delve deeper into GPU compute resource utilization using the profiling methodology proposed in Section 4.3.

4.5.2 GPU compute resource utilization

Here, we evaluate the utilization of the GPU's compute resources as GPU utilization, SMSP utilization, and TC utilization, as described in Section 4.3. Fig. 4.5 shows the GPU resource utilization at each level for a ResNet-50 TF eager mixed precision run on the A100 (Fig. 4.5a) and V100 (Fig. 4.5b). We use this run as an illustration of the measured utilization at different levels of the GPU hardware abstraction for different batch sizes. Instead, Fig. 4.6 shows the utilization results for all ML models but only for the batch size achieving the highest throughput in the A100 GPU.

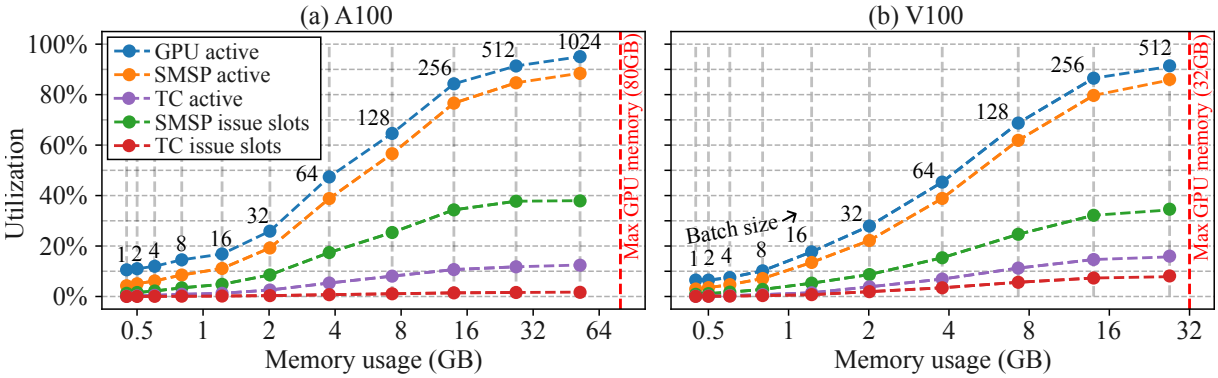


Figure 4.5: *Multi-level utilization vs. GPU memory usage. TensorFlow eager execution of a mixed precision ResNet-50 across various batch sizes on (a) A100 and (b) V100*

GPU utilization. We compare GPU utilization across all of our workloads for different batch sizes. In Fig. 4.5, as the batch size increases (annotated on the markers), we observe that GPU active utilization also increases, reaching a maximum of 95% and 91% for the A100 and V100, respectively. While SMSP active utilization follows a similar trend as GPU active utilization, SMSP issue slot utilization saturates at around 40% for the A100 and 35% for the V100. Looking at TC active and issue slot utilization, we also observe an increase with growing batch sizes, but for the A100 it always remains below 15% and 5%, respectively. For the V100, TC utilization is marginally higher, remaining below 20% and 10%, for active and issue slot utilization, respectively. We observed this

link between the increase in batch size and the increase in GPU utilization, as well as the saturations, for all of our workloads.

In Fig. 4.6, we can observe that BERT using TF eager execution achieves the lowest GPU utilization, which still tops at 76% using full precision and 78% using mixed precision. In general, PT achieves a slightly (2 to 6%) higher GPU utilization than TF, with the only exception being BERT using JIT compilation with mixed precision, where TF only achieves a 1% higher GPU utilization.

Despite this higher GPU utilization, TF generally achieves higher throughput. For example, DLRM full precision achieves 99.3% GPU utilization with PT eager compared to 87.4% with TF eager, but TF eager achieves $4.4\times$ higher throughput than PT eager. Furthermore, for the only workload where TF achieves higher GPU utilization than PT (BERT JIT mixed precision), PT achieves higher throughput. Hence, we observe no correlation between GPU utilization and throughput across the evaluated workloads and take our analysis one level deeper, looking at SMSP active utilization.

Key takeaway 4. *Higher GPU utilization does not correlate with higher throughput.*

SMSP active utilization. Fig. 4.5 shows that SMSP active utilization follows the same trend as GPU utilization across different batch sizes for ResNet-50 TF eager with full precision. We observed the same behavior for all tested workloads. However, as discussed in Section 4.3, GPU utilization is an upper bound for SMSP active utilization. We observe that SMSP active utilization is consistently 1% to 15% lower than GPU utilization. The highest differences between GPU utilization and SMSP active utilization are observed when training BERT with TensorFlow. Similarly to GPU utilization, we observe no correlation between SMSP active utilization and achieved throughput across workloads.

Key takeaway 5. *SMSP active utilization mirrors GPU utilization across batch sizes, staying* 1% *to* 15% *lower.*

SMSP issue slot utilization. We compare issue slot utilization across all of our workloads for different batch sizes. Fig. 4.5 shows that SMSP issue slot utilization follows the same trend as GPU and SMSP active utilization: increasing with the increase of batch size. We observed the same behavior for all the workloads. However, Fig. 4.6 shows that average SMSP issue slot utilization generally stays below 40%, even for the biggest batch sizes. We observe two exceptions with PyTorch training of BERT and DLRM using full precision, where it approaches a maximum of 54%. Once again, we observe no direct correlation between SMSP issue slot utilization and throughput across the evaluated workloads.

To better understand SMSP issue slot utilization, we profile this metric for each kernel and show the distribution in Fig. 4.7. In Fig. 4.7, we only show the ResNet-50 model using mixed precision XLA JIT compilation, for multiple batch sizes, for the A100 (Fig. 4.7a) and V100 (Fig. 4.7b) GPUs. Other workloads show similar trends. Average SMSP issue slot utilization (dotted line on Fig. 4.7) follows the same trend as seen in Fig. 4.5, and saturates near 40%. However, data distribution (violin plots and vertical bars in Fig. 4.7) shows maximum SMSP issue slot utilization ranging between 50% and 80%, generally increasing with batch size.

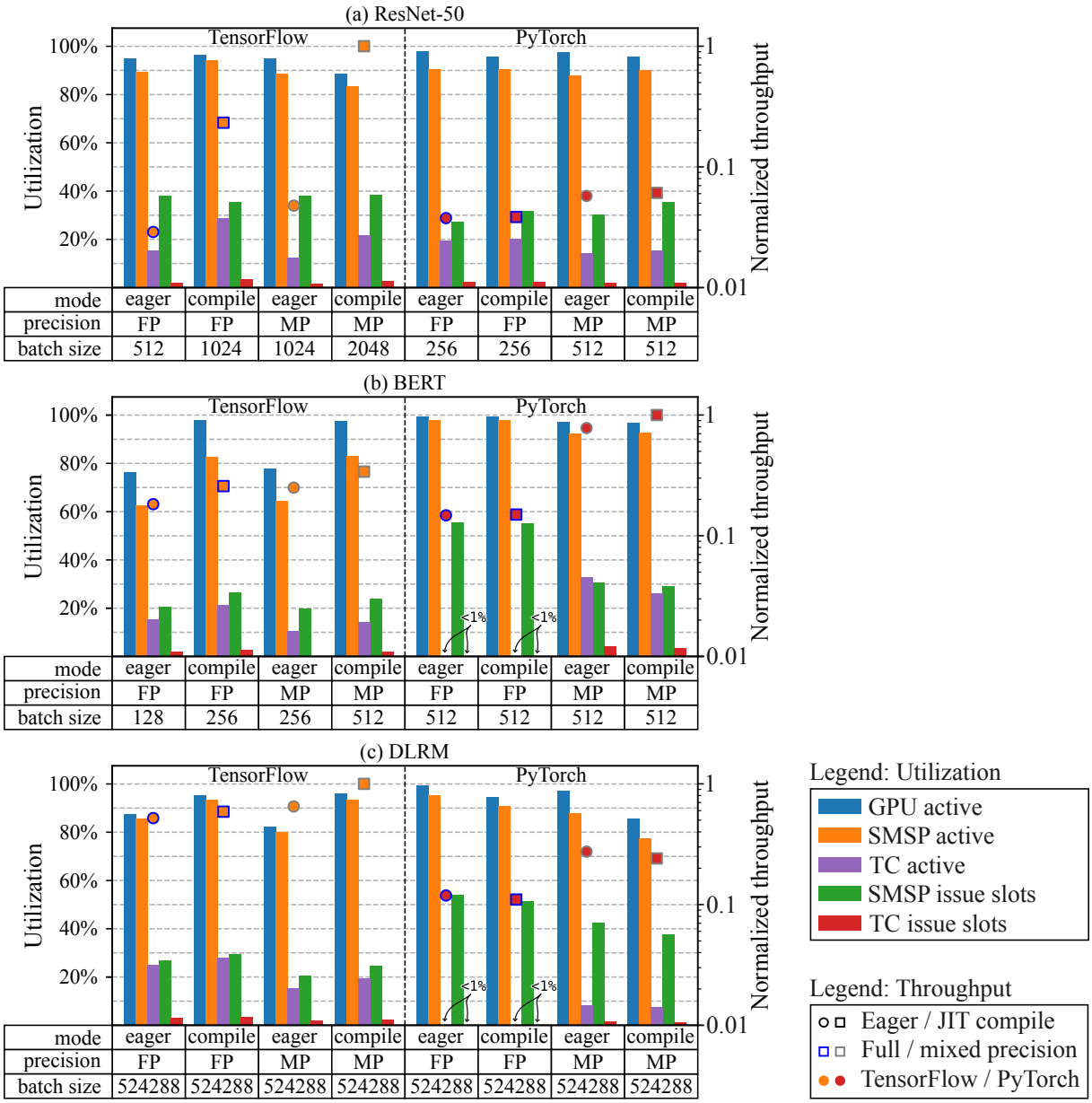


Figure 4.6: *Multi-level utilization (A100 workloads): GPU, SMSP (active & issue slots), and tensor core (active & issue slots); batch size chosen based on best throughput value for each workload; throughput normalized by best value for each model*

For example, Fig. 4.7 shows a maximum value of issue slot utilization of 50% when using no batches, compared to 80% when using a batch size of 2048 for this workload running on the A100. We also observe that although the average (and maximum) value is very similar for both GPUs, the A100 includes a higher population of kernels with an SMSP issue slot utilization above 60%. Despite this, the average remains low due to the majority of kernels saturating near 40%.

Key takeaway 6. *The average SMSP issue slot utilization increases with the batch size, but it seldom exceeds* 40%.

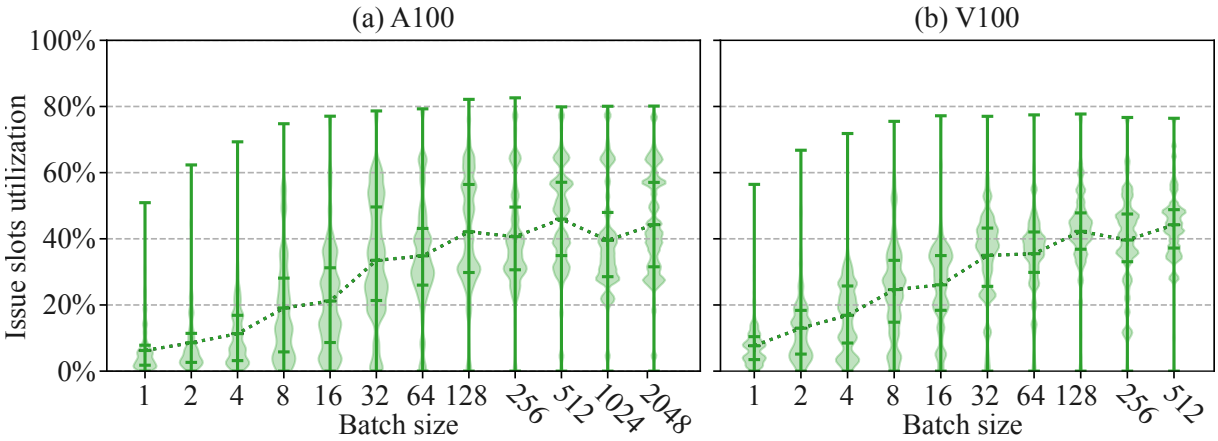


Figure 4.7: *Distribution of SMSP issue slot utilization for all kernels of ResNet-50 using XLA JIT with mixed precision for multiple batch sizes on (a) A100 and (b) V100 GPUs*

TC active and issue slot utilization. We compare tensor core utilization across all workloads for different batch sizes using two metrics: TC active utilization and TC issue slot utilization. Fig. 4.5 shows an increase in both metrics as the batch size increases for ResNet-50 using TF eager execution with full precision. We observed the same behavior for all the workloads. However, Fig. 4.6 shows that tensor cores are active for less than 35% of the total execution time.

Furthermore, less than 5% of the total execution cycles are spent issuing TC instructions. It is worth noting that training BERT and DLRM using PyTorch with full precision achieves a particularly low TC active and issue slot utilization (less than 1%). At the same time, training BERT with PyTorch using eager execution with mixed precision shows the peak TC active utilization, reaching nearly 35%. Similarly to the previous metrics, we observe no direct correlation between TC utilization and throughput across the evaluated workloads. Hence, we delve deeper into the distribution of TC issue slot utilization to gain more insights.

Fig. 4.8 shows the SMSP and TC utilization distributions for all kernels in the TF eager workloads, aggregated by layer. We observe that only convolution (i.e., Conv2D) and fully connected (i.e., MatMul) layers achieve TC issue slot utilization above 3%.

On the one hand, average TC active utilization can reach nearly 80% for pure MatMul operations, but it is limited to around 50% for Conv2D layers. On the other hand, the average TC issue slot utilization tops at 10%, with some kernels achieving 15% at peak. For reference, Fig. 4.8 also includes per-layer distributions of SMSP active and issue slot utilization. In contrast, the average SMSP active utilization is between 90% and 100% for the same layer types and the average SMSP issue slot utilization is between 25% and 40%. Hence, TC utilization does not seem to dominate the overall SMSP utilization, even for layers that use tensor cores extensively.

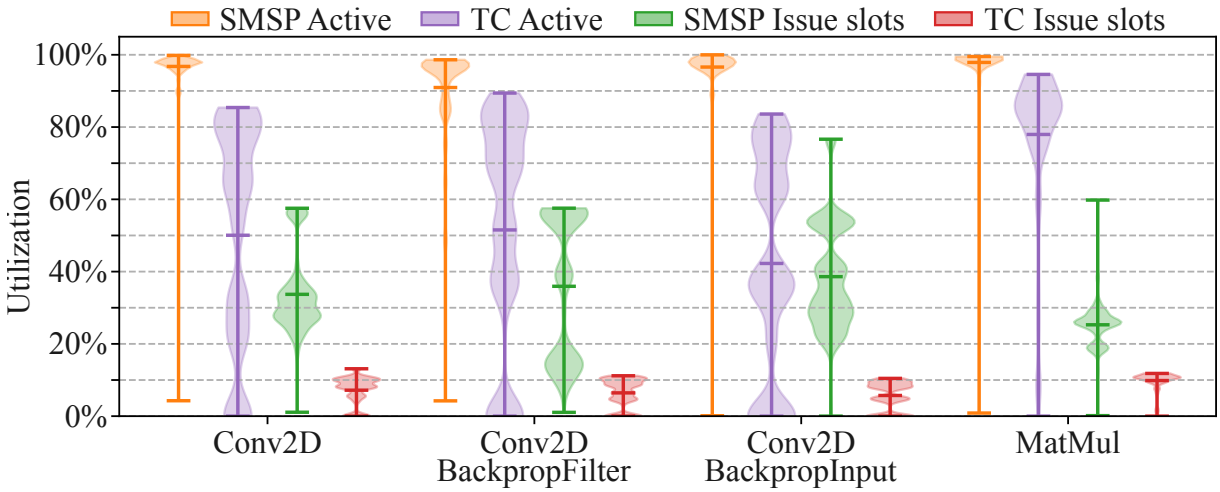


Figure 4.8: *Per-layer SMSP and TC utilization distributions (showing only layers with over 3% average TC issue slot utilization)*

To better understand how this relatively low issue rate of tensor core instructions impacts performance, Fig. 4.9 shows the cumulative percentage of GPU execution time as a function of TC issue slot utilization, for the eager workloads on TensorFlow (Fig. 4.9a) and PyTorch (Fig. 4.9b). Here, we observe that the kernels that do not issue any tensor core instructions (i.e., 0% on the x-axis) amount from 30% (training BERT with PyTorch using mixed precision) to 95% (training BERT with PyTorch using full precision) of the total A100 training time, depending on the workload. Furthermore, for both ML frameworks, kernels with less than 12% of TC issue slot utilization represent more than 99% of the total GPU kernel execution time of all workloads.

As previously observed in Fig. 4.6, BERT and DLRM using PyTorch with full precision achieve a particularly low TC issue slot utilization. This observation is also reflected in Fig. 4.9 where kernels with less than 2% of TC issue slot utilization represent more than 99% of the total GPU kernel execution time for these workloads. Despite these low utilization rates, the increased use of tensor cores when using mixed precision seems to be correlated with a significant increase in training throughput, as shown in Fig. 4.6. However, in Fig. 4.9, we observe that even the workloads benefiting

the most from tensor cores are now limited by the execution time of kernels that do not use tensor cores (Amdahl's law at play).

Key takeaway 7. *The majority of the kernels do not use tensor cores. Kernels that use tensor cores more extensively amount to a small proportion of the total GPU execution time.*

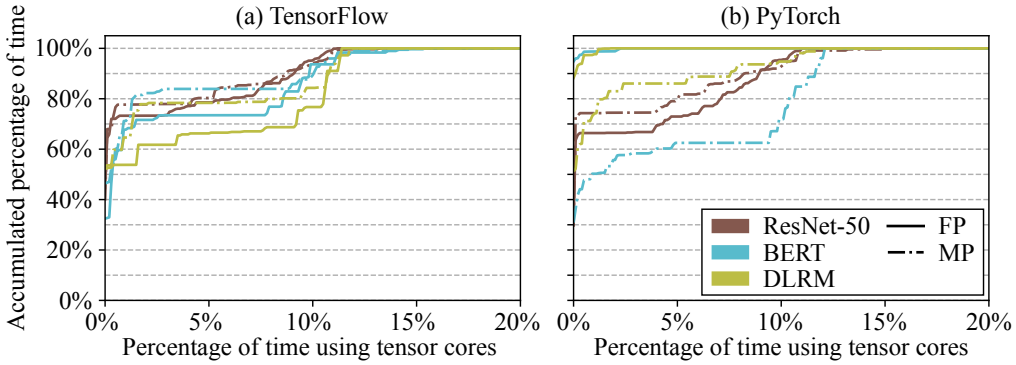


Figure 4.9: *Cumulative distribution of kernels' TC issue slot utilization for eager workloads (A100 GPU). Batch size is chosen based on the best throughput value for each workload*

4.5.3 Implications and Insights

To achieve sustainable performance improvements within ML training workloads, it is important to maintain a balanced utilization of key architectural resources. GPUs have increased their memory capacity in recent generations, enabling enhanced throughput and higher utilization through the support of larger batch sizes. However, our experiments on representative workloads suggest a plateau has been reached, and the additional memory in the A100 no longer leads to enhanced utilization ratios.

We also show that modern GPUs can achieve impressive acceleration but typically operate below 50% of their instruction-issuing potential. Furthermore, we observe that the tensor cores, which are the instructions delivering the highest raw computational power, are kept idle most of the time, and the evaluated ML training workloads are now constrained by kernels not using tensor cores. Thus, our results suggest that the current GPU paradigm is reaching a saturation point, and motivate further research into programmable architectures to sustainably accelerate ML training workloads.

4.6 Related works

To the best of our knowledge, no other studies have analyzed the computing efficiency of GPU execution of multiple ML workloads as thoroughly as we have in this paper.

In this section, we identify and detail two categories of the related works. Table 4.3 presents a summary of the different metrics covered by the related works.

On the one hand, profiling tools are provided by GPU manufacturers and ML frameworks. However, those are usually intended to analyze specific aspects of the hardware or software stack of ML applications. NVIDIA provides several profiling interfaces [110],[113],[114] for developers to evaluate GPU performance bottlenecks. ML frameworks provide specific profiling tools for developers to evaluate the performance of ML models [124, 148]. In this chapter, we gather metrics using both NVIDIA Nsight Compute [114] and ML frameworks profilers [124, 148]. When using application replay to gather a number of metrics, Nsight Compute [114] can match kernels between different runs by comparing their names. However, Nsight Compute can only perform this matching between its own reports and cannot match kernels from other profiling tools. Here, we implement a similar technique to match kernels between profiling tools. As described in Section 4.3, this approach relies on the identification of common metrics between the tools and the establishment of a correlation matrix.

On the other hand, previous works from the literature have also leveraged these existing profiling tools to provide analyses on ML workloads. Some directly use these tools to analyze ML workloads. For example, Verma et al. [151] propose a high-level performance study (using the Roofline model [153]) of the MLPerf Training benchmark suite [95] and compare it to DAWNBench [38] and DeepBench [102], two other ML benchmark suites. Others develop meta-tools that can identify bottlenecks by doing some automatic post-processing of the data reported by a single profiler [130, 159, 160], or aggregated data from multiple profilers [58, 91]. While most of these related works evaluate performance through metrics like throughput and latency, some works also evaluate GPU utilization from a high-level view [58, 91, 151], as described in Section 4.3. For example, Li et al. [91] proposed a profiling methodology to analyze the execution of ML inference on GPUs. This methodology is also based on leveraging multiple profiling tools to provide measurements at different levels of abstraction (i.e., model-level, layer-level, kernel-level). However, to the best of our knowledge, this is the first work to propose a finer-grain analysis of GPU utilization and systematically evaluate GPU resource utilization at multiple levels and across multiple training workloads, ML frameworks, and execution modes.

Table 4.3: *Comparison to related works*

Solution	High-level metrics	Low-level metrics	Multi-level metrics
[110]	GPU freq. & temp.	Perf. counters	✗
[113]	GPU freq. & temp. (via [110])	✗	✗
[114]	✗	Perf. counters (via [110])	✗
[124, 148]	Total exec. time, per-kernel exec. time, GPU utilization, throughput, DRAM usage, etc.	✗	✗
[151]	Arithmetic and bandwidth intensity (based on [153])	✗	✗
[160]	✗	Instruction execution and memory depencies (based on [111])	✗
[130]	✗	Instruction Per Cycle, instruction stall reasons, warp efficiency	✗
[159]	✗	Program Counter (PC) sampling (based on [111])	✓ Calling context (reconstructed) to blame application variables or functions
[58]	Execution time (total), GPU utilization	✗	✗
[91]	Execution time (total, per-layer, per-kernel), DRAM usage, arithmetic and bandwidth intensity (based on [153])	DRAM reads/writes, kernel flops	✓ Low-level metrics aggregated by model, layer or kernel
Ours	Execution time (total, per-kernel, per-kernel), GPU utilization, throughput, DRAM usage (via [148] or [124])	SMSP elapsed, active, issue cycles and TC active, issue cycles	✓ Multi-level utilization (global, per-layer and per-kernel) (see Table 4.1)

4.7 Summary

In this chapter, we present a multi-level analysis of GPU computing resource utilization for ML training workloads. To this end, we propose a methodology that combines traces from existing profiling tools to compare the execution of various workloads and identify useful new insights.

First, in section 4.2, we describe an ideal reference execution of a GPU-accelerated ML training loop. With this reference, we identify relevant metrics that can be measured at different levels of abstraction, using multiple existing profiling tools. Second, in section 4.3, we propose a systematic methodology that combines traces from these profiling tools. This methodology allows us to evaluate the utilization of the GPU computing resources at different levels of abstraction: GPU level, SMSP level, and TC level. For the SMSP and TC levels, we focus on evaluating both the utilization of the GPU cycles and the utilization of the instruction issue slots. This allows us to evaluate if the computing resources are used often and if they are kept as busy as possible. Finally, implementing this methodology on two modern GPUs (i.e., NVIDIA V100 and A100), we analyze the impact of different software optimizations (i.e., full vs. mixed precision, eager vs. JIT compilation) using two ML frameworks (i.e., TF and PT) for various batch sizes, in section 4.5. While this study focuses on NVIDIA GPUs, the approach does apply to other GPU brands that give access to their performance counters. For example, AMD GPU performance counters can be easily accessed using the AMD GPUPerf API [9]. The proposed analysis methodology also has potential to improve performance prediction models for GPU-accelerated workloads [19], which could benefit from using new metrics such as multi-level utilization of the GPU resources.

We provide our analysis in two parts, providing new insights summarized through seven key takeaways. In the first part of our analysis, we focus on high-level metrics, such as throughput and memory usage. This allows us to evaluate the impact of different software optimizations on the overall performance of the GPU with different workloads. From this high-level analysis, we show that the A100 GPU saturates the throughput for all workloads, thanks to its larger memory, while the V100 GPU does not. We also show the high impact of software optimizations such as mixed-precision training and JIT compilation on the throughput and memory usage. In the second part of our analysis, we leverage the systematic methodology proposed in section 4.3 to evaluate the multi-level utilization of the GPU resources. From this analysis, we show that high utilization is typically achieved at the GPU level and SMSP level. Yet, the average SMSP issue slot utilization remains below 50%, with tensor core instructions reaching less than 5.2% of the issuing opportunities. We believe this work highlights the need for advanced profiling to unravel GPU limitations.

CHAPTER 5

Analyzing GPU Energy Consumption in Data Movement and Storage

Contents

5.1 Context and motivation **72**
5.2 Background **74**
5.2.1 GPU cache hierarchy 74
5.2.2 GPU performance counters and internal power sensors 76
5.3 Related work **76**
5.4 Energy model of GPU data movement **78**
5.4.1 Background on GPU energy models 78
5.4.2 Analytical energy model 79
5.4.3 Influence of parallel accesses 79
5.5 Methodology **80**
5.5.1 Parameter identification phase 80
5.5.2 Calibration phase 82
5.5.3 Evaluation phase 87
5.6 Implementation **88**
5.6.1 Experimental setup 88
5.6.2 Parameter identification 89
5.6.3 Calibration 89
5.6.4 Cross-validation of the calibration results 91
5.7 Evaluation of complete applications **92**
5.7.1 Matrix multiplication 92
5.7.2 ResNet-50 training iteration 93
5.8 Summary **95**

5.1 Context and motivation

In the previous chapter, we provide a new methodology to profile and analyze the performance of Machine Learning (ML) training on modern GPUs. In this analysis, we observe the impact of the architecture changes between the V100 (i.e., Volta architecture) and the A100 (i.e., Ampere architecture) on performance for ML training. But as the peak performance increases generation over generation, so does the Thermal Design Power (TDP). For example, the A100 has an increased TDP of 400W, compared to the V100's 300W, while the TDP of the upcoming NVIDIA Blackwell B200 is estimated at 1000W.

Despite this increase in performance and power consumption, the demand for computing power continues to grow. As we described in Chapter 2, the training of ML application is a major contribution to this demand. Furthermore, these higher computation requirements drive the design of future accelerators, GPUs included. Designing more powerful accelerators relies on a deep understanding of performance bottlenecks of current accelerators, such as GPUs (we propose profiling methodologies for performance in Chapters 3 and 4). However, analyzing performance is not sufficient if we want to design future accelerators to be energy-efficient, it is crucial to understand energy bottlenecks as well. GPU manufacturer-provided profiling tools do not offer a straightforward solution to evaluate energy bottlenecks or any energy consumption breakdown. In addition, the proprietary and complex design of modern GPUs makes it difficult to estimate such an energy breakdown using publicly available tools.

Estimating energy consumption breakdowns of a workload running on a given GPU can be done using different approaches. One approach that architecture designers have resorted to is to use simulators that use models of the GPU architecture to estimate its energy consumption [30, 73, 89]. However, the simulation approach can be time-consuming for heavy workloads. Another approach is to run the workload on the real GPU, measure its power consumption, and use similar models to attribute the power consumption to different components of the GPU. Both approaches are very dependent on the accuracy of the model and research around providing accurate power models for GPUs has been active in the past decade.

Researchers have proposed different techniques to establish power models for GPUs. All these techniques consist of running characterized workloads on the GPU and measuring the power consumption, then using these measurements to calibrate a model. The characterized workloads can be typical GPU benchmarks [27, 65, 140], i.e., kernels that are known to be representative of real-world GPU workloads; or microbench-

marks [71, 96, 141], i.e., small programs that target specific components of the microarchitecture to evaluate their performance or power consumption.

However, previously proposed techniques and power models have limitations that make them unsuitable for estimating the energy bottlenecks of modern GPUs. For example, some of these approaches are either: being performed on older GPU architectures [64, 94, 101], which have evolved substantially in the last year (see Chapter 2); relying on external power measurement tools [13, 17, 64, 94, 101], which is impractical when the GPU is not physically accessible (e.g., cloud server); analyzing PTX assembly code [6, 20, 27, 61, 94, 97, 106], which is not applicable when using vendors-provided precompiled libraries (e.g., cuDNN [34]); using Deep Learning (DL) models to predict power consumption [20, 61, 97, 98, 135, 154] which obfuscates the power consumption details of the GPU component behind trained weights; ignoring crucial kernel parameters that can influence the energy consumption per memory transaction [17] (i.e., type of transaction, kernel dimensions, access pattern); ignoring memory operations altogether [13]; or, focusing on the software abstractions of the GPU memory hierarchy [8], which do not provide a detailed breakdown of the energy consumption at the microarchitectural level. We detail these limitations further in Section 5.3.

Our **main goal** in this chapter is to address the shortcomings of the state-of-the-art by providing an energy model and a calibration method to estimate a breakdown of the energy consumption of data movement in modern GPU architectures.

As previous works point to memory as one of the main contributors to power consumption in modern GPUs [73], we focus on modeling data movement and storage. However, unlike previous works, the proposed model details different components of the memory hierarchy (e.g., shared memory, L1, L2, DRAM) and allows us to evaluate their energy consumption independently. In the next sections, we first describe a basic energy model for modern GPUs, building from the existing literature [64, 68] (Section 5.4) and focusing on data movement and storage. Then, we propose a methodology to calibrate this model using microbenchmarks (targeting different levels of the memory hierarchy) and using performance counters as a safety check for the calibration results (Section 5.5). Based on performance counters and internal power sensors (included in modern GPUs), our method uses the microbenchmarks to stress the different levels of the memory hierarchy with a varying number of threads to evaluate the minimum amount of energy consumed (i.e., lower bound) by a given memory access. We implement our methodology on the NVIDIA A100 GPU using its internal power sensor (Section 5.6). Finally, we evaluate energy consumption breakdowns for multiple workloads of increasing complexity, up to a training of the ResNet50 ML model (Section 5.7).

5.2 Background

5.2.1 GPU cache hierarchy

As mentioned in Chapter 2, modern GPUs have cached memories to reduce the latency (and energy consumption) of memory accesses (i.e., LOADs or STOREs). Modern GPUs' memory hierarchy is composed of several levels of data cache: L1 cache (private to each Streaming Multiprocessor (SM)) and L2 cache (common to multiple SMs). Both L1 and L2 caches are managed by the hardware and are used to cache data from the DRAM (i.e., global memory or main memory). Beyond the main memory, data transfers between the host CPU and the GPU device are managed by software, outside the execution of GPU kernels. While these communications could also involve energy overhead (e.g., due to page faults), we do not address the storage layers in this contribution and focus on data movement within the GPU architecture (i.e., through the DRAM and data caches).

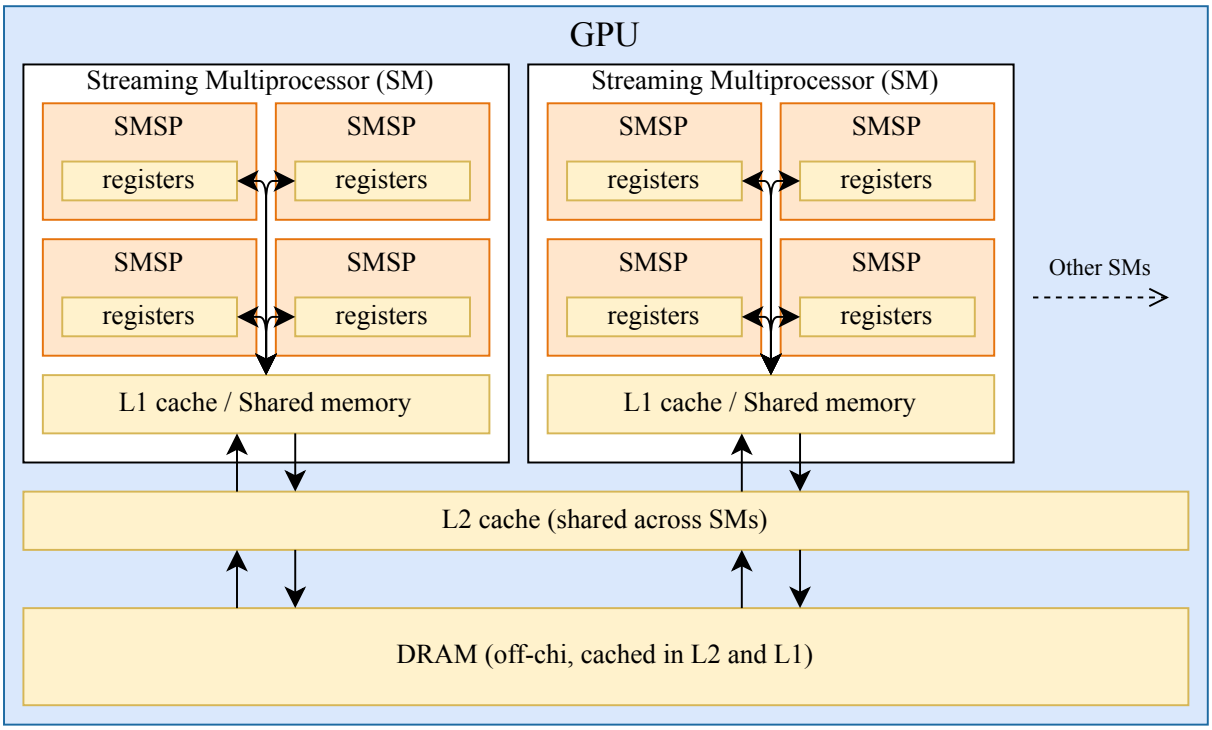


Figure 5.1: *NVIDIA A100 DRAM and data caches hierarchy*

Accesses to these hardware caches are tagged as *HIT* or *MISS* depending on whether the data targeted by a given access is present in the cache or not. When a *MISS* occurs, the data is fetched from the next level of the memory hierarchy (e.g., L1 cache to L2 cache, L2 cache to DRAM). These hardware-managed caches are complemented by software-managed caches (e.g., texture cache, constant cache, shared memory) from (to) which a programmer can explicitly load (store) data. Each access to a memory preloads a complete *cache line* to be accessed by the threads before the next access replaces it. The cache line size is known as the *access granularity* of the memory. The minimum access granularity for the NVIDIA A100 GPU is 32 bytes (i.e., a sector). In this chapter, the proposed methodology covers both types of data caches (hardware-managed and software-managed), along with the DRAM.

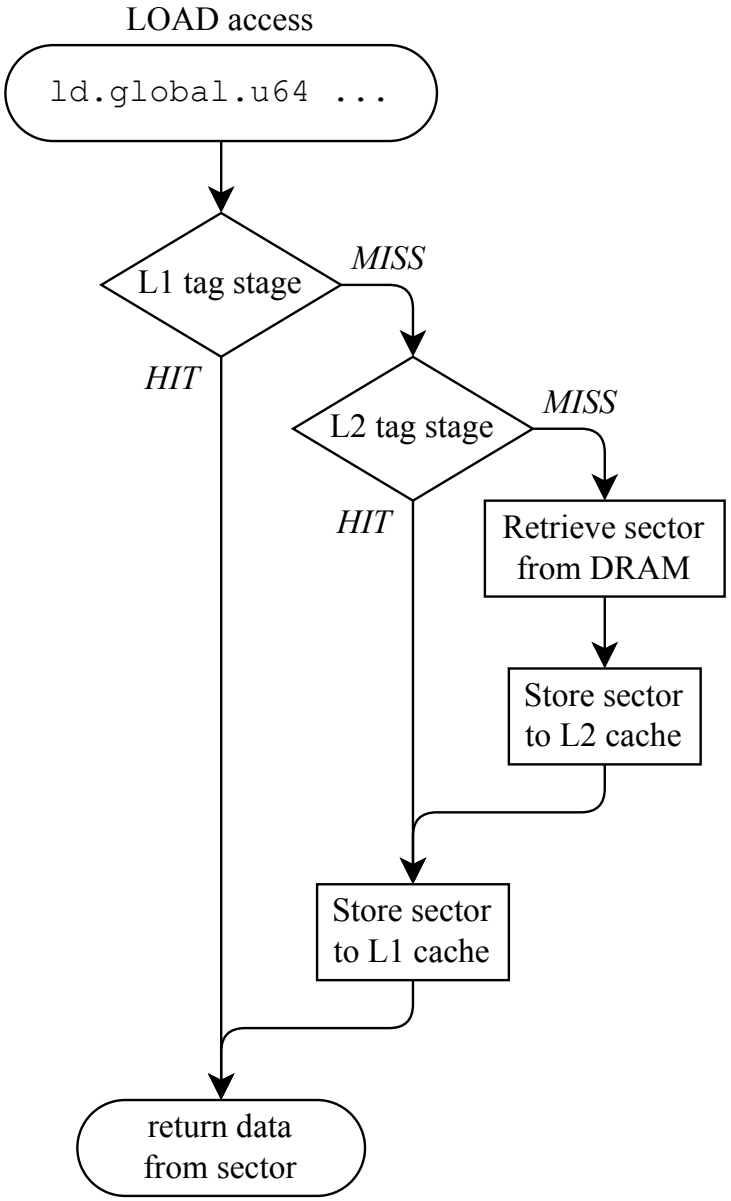


Figure 5.2: *NVIDIA A100 LOAD memory access process flow*

5.2.2 GPU performance counters and internal power sensors

As mentioned in Chapter 5.2, NVIDIA GPUs provide access to a set of performance counters through the NVIDIA Nsight Compute (ncu) profiling tool [114]. Performance counters are exposed to the user through a set of *metrics*. A limited number of metrics can be requested during a profiling run before the GPU driver has to replay kernels multiple times to gather all needed performance counters, adding profiling overhead (i.e., kernel replay). In this chapter, we select specific metrics to evaluate the number of memory transactions at each level of the GPU memory hierarchy. Modern NVIDIA GPUs also provide access to an internal power sensor using the NVIDIA Management Library (NVML). Limitations of such sensors were detailed in previous works [23] and evaluated against external power sensors [13, 23]. These works show that the error introduced by using the internal power sensor can be minimized with basic post-processing techniques, making it a reliable source of power measurements. We build upon the previous works to collect power measurements and provide more details on our methodology in Section 5.5.

5.3 Related work

Numerous GPU power models were proposed in previous works, some of which use microbenchmarks to calibrate their models [8, 13, 17, 61, 64] and/or provide estimates for either the total GPU power consumption [6, 20, 61, 94, 97, 135, 154] or a detailed power breakdown [6, 24, 64, 101, 127, 138]. In this section, we discuss the limitations of these works and how our methodology addresses them. We identify seven main limitations in the existing literature.

Application-based energy breakdown. Some works base their energy breakdown on different execution phases of ML applications. For example, showing the energy consumed by running a specific layer of an ML model [127] or predicting the energy consumption based on the architecture of the ML model [24, 138]. While these breakdowns and predictions are of great value for ML developers to optimize their applications, they do not provide detailed insights on the energy consumption of inner components in the GPU architecture. In contrast, in this chapter, we propose a technique to evaluate energy breakdowns based on hardware performance counters, and show the energy consumption of specific GPU memory accesses.

Dated GPU architectures. Some works rely on dated GPU architectures [64, 94, 101]. While the proposed methodologies are using similar approaches to ours (e.g., mi-

crobenchmarks, performance counters), they are not directly applicable to modern GPUs with hardware-managed cache hierarchies. In contrast, we base our energy model on modern GPU architectures. We implement and test our methodology on an NVIDIA A100 GPU, from the Ampere generation (2020), which includes hardware-managed cache hierarchies.

Ignore memory operations. Some works evaluate the energy cost for compute instructions but ignore memory instructions altogether [13] (i.e., only evaluating the energy consumption of compute operations). Similarly, some works focus only on the software abstractions of the GPU memory hierarchy [8], without distinguishing between the physical implementations. This can lead to inaccurate evaluations if performed on modern GPUs, which include hardware-managed cache hierarchies. In this chapter, we propose a complementary contribution to these works by characterizing the energy consumption of transactions across the different levels of the GPU memory hierarchy.

Ignore crucial kernel parameters. Some works ignore crucial kernel parameters which can lead to overestimations of the energy of memory transactions [17] (i.e., type of transaction, kernel dimensions, access pattern). Our microbenchmark-based methodology takes into account the aforementioned kernel parameters to provide a tight lower-bound estimation of the energy consumption of memory transactions. We give a detailed description of the methodology and its design choices in Section 5.5.

Require access to PTX code. Some works rely on analyzing the PTX assembly code of GPU benchmarks [6, 20, 27, 94, 97, 106] or microbenchmarks [61]. These approaches are not applicable when the PTX code of the targeted GPU workloads is not accessible, which is the case in vendors-provided precompiled libraries often used by ML applications (e.g., cuDNN [34] is used by TensorFlow [2] and PyTorch [121]). In contrast, our methodology does not rely on the PTX code of the targeted GPU workloads but uses performance counters as input to the energy model.

DL-based models. Some works rely on using DL-based models [20, 61, 97, 98, 135, 154] trained to infer power predictions based on performance counter values. While these models can provide accurate power predictions, using DL models obfuscates the power consumption details of the GPU component behind trained weights. Hence, these models cannot provide a detailed power breakdown of the GPU architecture. We propose a more simple and interpretable energy model, which can provide a detailed power breakdown of the GPU energy consumption.

Require physical access. Some works [13, 17, 64, 94, 101] use external power measurement tools [15], building testbeds to measure the power consumption of the GPU.

This makes the methodology less relevant when physical access to the GPU device is not available (e.g., in a cloud server). Furthermore, previous works have compared the internal power sensor readings to external power measurements [13, 23], showing that the error introduced by the internal power sensor can be made negligible with using simple post-processing of the power measurements. While we base our calibration methodology on the GPU's internal power sensor and performance counters, to make it relevant even without physical access to the GPU device, the methodology still is applicable using external power measurement tools.

5.4 Energy model of GPU data movement

In this section, we propose an energy model to characterize the energy consumption of data movement in the GPU memory hierarchy, noted E_{MEMORY}. We first provide background on the different high-level components of the GPU energy model, previously described in the literature. Then, we enrich the model with more details on data movement and storage, based on our analysis of modern GPU architectures. The proposed analytical energy model covers both software-addressed memory spaces and hardware-managed caches. Finally, we discuss the influence of parallelized transactions on the energy consumption of the memory hierarchy.

5.4.1 Background on GPU energy models

The energy consumption of a GPU running a kernel can be divided between static and dynamic energy components:

$$E_{\text{TOTAL}} = E_{\text{STATIC}} + E_{\text{DYNAMIC}}. \tag{5.1}$$

The static energy is the part of the total energy consumed by the GPU, which is constant (i.e., independent of the kernel being executed). This static energy component is specific to a given GPU model and influenced by the temperature, the voltage, and the frequency of the GPU. The dynamic energy is the additional energy (i.e., on top of the static energy) consumed by the GPU when it is executing a kernel. This dynamic energy component has been modeled in the past as the sum of the energy consumed by the memory (E_{MEMORY}) and the energy consumed by the SMs (E_{COMPUTE}) [64, 68]:

$$E_{\text{DYNAMIC}} = E_{\text{MEMORY}} + E_{\text{COMPUTE}}. \tag{5.2}$$

Static energy (E_{STATIC}) and compute energy (E_{COMPUTE}) have been modeled further in the past [13, 64]. However, in this chapter, we focus on providing a tight lower-bound estimation of the energy consumed in data movement and storage (E_{MEMORY}).

5.4.2 Analytical energy model

Energy spent by moving data across the GPU cache hierarchy is the sum of energy spent loading (storing) data from (to) the different memory levels. Hence,

$$E_{\text{MEMORY}} = \sum_{\text{MEM}} E_{\text{MEM}}. \tag{5.3}$$

We define the energy spent by reading data from a given memory level (E_{MEM}) as the number of accesses (noted $\#$accesses) multiplied by the typical energy cost of one access to this memory level (noted ε_{MEM}). However, accessing a given memory level also requires the activation of the inner components of this memory level (e.g., clock tree), which we evaluate as an offset energy (noted ΔE_{MEM}). In Section 5.5, we propose a methodology to identify and isolate the energy cost of one HIT access from the offset energy using linear regression:

$$E_{\text{MEM}} = \#\text{accesses} \times \varepsilon_{\text{MEM}} + \Delta E_{\text{MEM}}. \tag{5.4}$$

In the case of hardware-managed data caches (L1 to LLC), a MISS will be matched to either a HIT at a lower cache level or access to the main memory. Hence, for hardware-managed data caches, we restrict the count of the number of accesses only to HIT accesses (i.e., when the data actually moves from one level to another). During such cache accesses, data moves from one level to the registers, to be processed by the cores when executing further compute instructions.

5.4.3 Influence of parallel accesses

GPUs are intended to achieve better energy efficiency in two ways: by accessing a given memory using multiple threads at a time (i.e., parallel accesses) and/or by amortizing the energy cost of loading a cache line with subsequent memory accesses (i.e., coalescence). Hence, the energy cost of one memory access (noted ε) to a given memory level is influenced by three parameters: the number of threads accessing the memory at a time (noted N_T), the coalescence of the accesses and the access granularity of the memory (i.e., size of the cache line, noted C) [70]. Ideally, memory operations are sub-

sequent and fully coalesced and the number of accesses can be defined as the number of entire cache lines that a given amount of threads will load:

$$\#\text{accesses} = \left\lceil \frac{N_T}{C} \right\rceil \times C. \tag{5.5}$$

In this case, better energy efficiency per access can only be achieved by having enough threads to occupy all memory ports in parallel, capitalizing even more on the energy cost of activation of the memory,

$$\varepsilon = \frac{E_{\text{MEM}} - \Delta E_{\text{MEM}}}{\left\lceil \frac{N_T}{C} \right\rceil \times C}. \tag{5.6}$$

In Section 5.5, we increase the number of threads when accessing a memory level to find the lower limit of Eq. 5.6.

5.5 Methodology

In this section, we describe a 3-step methodology to evaluate a lower bound for the energy consumption of data movement and storage in the GPU memory hierarchy (illustrated in Fig. 5.3), based on the proposed energy model in Section 5.4.

First, we identify important parameters of the device, such as the sizes of the different cache levels and the performance counters that count the number of accesses at each cache level (5.5.1). Second, we calibrate the energy model using microbenchmarks using the previously identified parameters (5.5.2). We describe the design of the microbenchmarks, the considerations regarding access granularity and kernel dimensions, the gathering of performance counters and power measurements, and the use of linear regression to identify the energy cost of each memory access. Finally, we evaluate the energy breakdown for new applications, using the calibrated energy model (5.5.3). We demonstrate this methodology through an example implementation (5.6) and results on different workloads (5.7).

5.5.1 Parameter identification phase

Calibrating the proposed energy model relies on two specific sets of parameters, specific to the target GPU model. The first input parameter is the list of performance

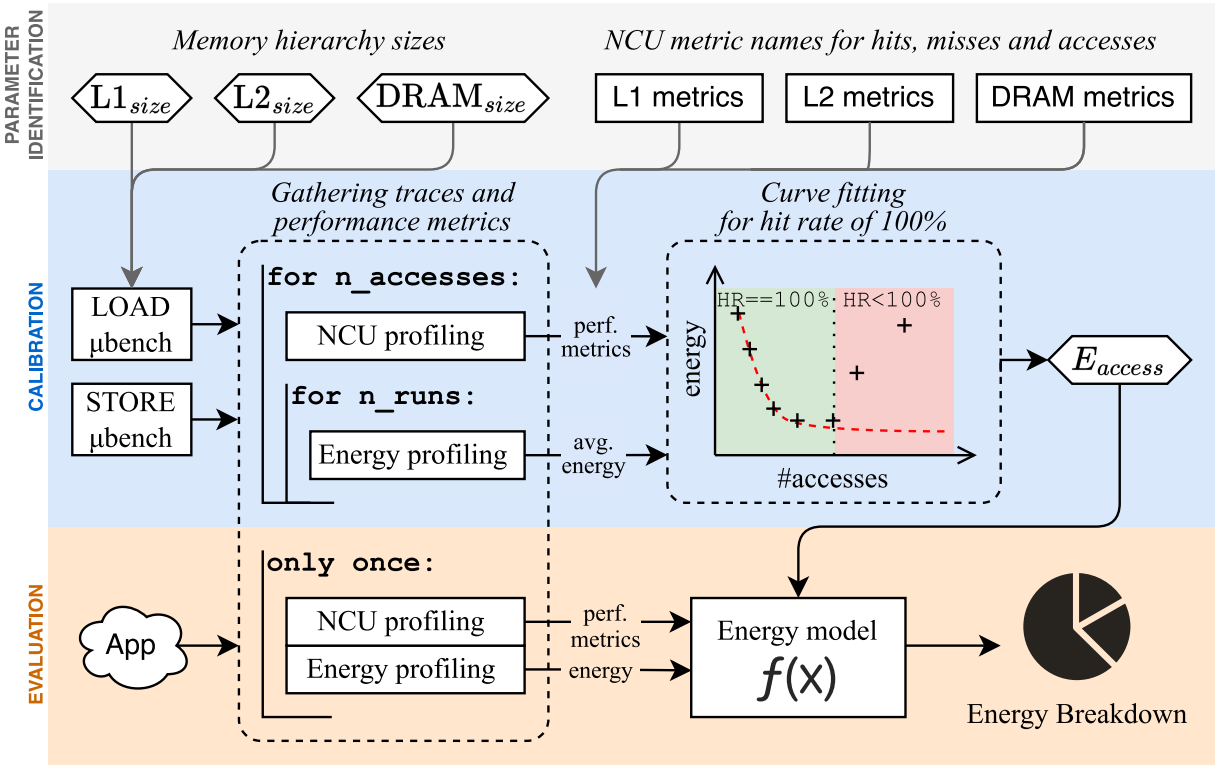


Figure 5.3: *Different phases of the methodology: (1) parameters identification, (2) calibration, and (3) evaluation*

counters that count the number of accesses at each cache level (along with the proportion of *HIT*s and *MISS*es). This parameter is needed as a safety check to ensure that the microbenchmarks are targeting the right cache level during the calibration phase. The second parameter is the list of sizes of each GPU cache level. The size of the targeted GPU cache is needed as a direct input for the microbenchmarks when calibrating the energy model. The sizes of the GPU caches are typically shared by the manufacturer in the GPU documentation (e.g., NVIDIA's whitepapers [104, 109]). However, the sizes of the GPU caches can also be identified by evaluating the hit/miss rate of the cache over a range of array sizes [66, 96].

For example, Fig. 5.4 shows the miss rate of the A100 GPU L1 and L2 caches over a range of array sizes. L1 cache miss rate increases as the array size passes 150kB and misses 100% of the time for an array size of 190kB. L2 cache miss rate increases as the array size reaches 15MB and misses 100% of the time for an array size of 22MB. NVIDIA's whitepaper [104] states that shared memory and L1 cache share a configurable 192kB memory space. L2 cache is 40MB, divided into 2 partitions of 20MB each. The results of this evaluation are consistent with the manufacturer's documentation.

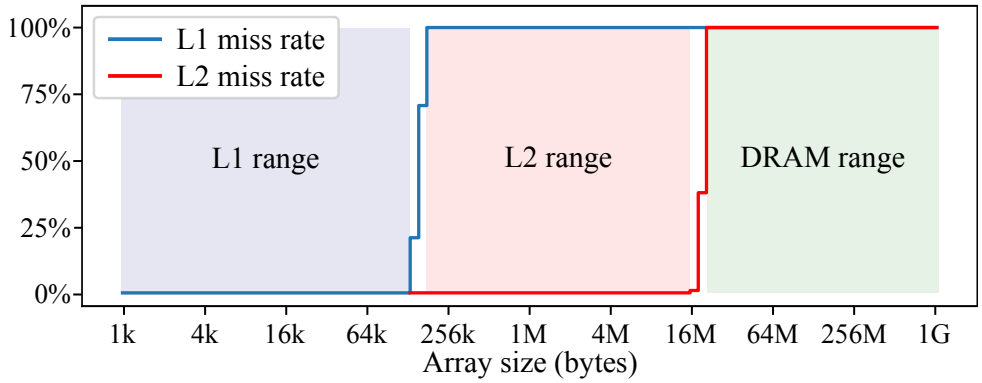


Figure 5.4: *Miss rate of the L1 and L2 caches of the A100 GPU, over a range of array sizes between 1 kB and 1 GB*

5.5.2 Calibration phase

Once the parameters of the device are identified, we calibrate the energy model by running the microbenchmarks on the target GPU, sweeping the number of memory accesses while gathering the performance counters and power measurements, for each memory level. We detail this calibration phase by describing the general objective of the microbenchmarks (1) and the considerations for the access granularity and the dimensions of the kernel (2). Then, we describe how we gather performance counters (3) and power measurements (4). Finally, we explain how we use linear regression to identify the energy cost of each memory access (5).

5.5.2.1 Microbenchmarks

We divide the proposed microbenchmarks into two codebases (written in CUDA with inline PTX assembly code): one for *LOAD* operations and one for *STORE* operations. A simplified version of the *LOAD* microbenchmark is shown in Listing 5.1. Both microbenchmarks are based on the pointer-chasing algorithm used in previous works [17, 96], which consists of a loop that iterates over an array of pointers, each pointing to another element of the array, separated by a given *stride*. The objective of the microbenchmarks is to perform a large number of memory transactions on an array of 64-bit unsigned integers (i.e., `uint64_t`), the size of the array is chosen to fit on the targeted cache level according to the identified parameters (*see Fig. 5.4*). Using pointer-chasing allows for a higher ratio of memory transactions versus other instructions when executing the microbenchmarks.

Listing 5.1 shows a typical microbenchmark containing two loops: a *warm-up* loop and a *measurement* loop. The warm-up loop (lines `13` to `17`) is used to move the pointer-chasing addresses from the GPU global memory to the targeted cache. During the ex-

```
1  __global__ void load(uint64_t *array) {
2   // Get thread ID and block ID
3   uint64_t tid = threadIdx.x;
4   uint64_t bid = blockIdx.x;
5   // Compute thread start address
6   uint64_t *start = array + tid + (bid * SUBTAB_SIZE / sizeof(
       uint64_t));
7
8   asm volatile(
9   [...] // Init registers
10  "{.reg .pred %p;\n"
11  ".reg .u64 %tmp;\n"
12  "mov.u64 %tmp, %0;\n\n"
13  // Warm-up loop
14  "$warmup:\n"
15  "ld.global.u64 %tmp, [%tmp];\n"
16  "setp.ne.u64 %p, %tmp, %0;\n"
17  "@%p bra $warmup;\n"
18  // Measurement loop
19  [...] // Reset counter and address
20  "\n$measurement:\n"
21  "ld.global.u64 %tmp, [%tmp];\n" // 1st LOAD
22  [...] // unrolling X more LOADs
23  "setp.ne.u64 %p, %tmp, %0;\n"
24  "@%p bra $measurement;\n" // for SIZE/STRIDE
25  "add.u32 %k, %k, 1;\n"
26  "setp.lt.u32 %p, %k, %1;\n"
27  "@%p bra $measurement;\n" // for N_ITER
28  "}" : "+l"(start)
29  : "n"(N_ITER)
30  );
31 }
```

Listing 5.1: *Simplified LOAD microbenchmark kernel code*

ecution of this loop, all memory accesses to the targeted cache level will be tagged as *MISS*es, as the data is not yet present in the cache. The microbenchmark of Listing 5.1 is designed to target hardware-managed caches (e.g., L1, L2 for the NVIDIA A100). However, it can easily be adapted to target other memory spaces (e.g., shared memory) by manually programming the filling of the targeted memory space instead of the warm-up loop. In the case of the DRAM, the warm-up loop is not needed as the data is already in the DRAM memory and the microbenchmark can directly start with the measurement loop.

The measurement loop (lines `18` to `27` in Listing 5.1) follows the same data movement pattern as the warm-up loop. However, we unroll some iterations from the measurement loop to increase the ratio of memory instructions per iteration. This helps

to reduce the contribution of other instructions (e.g., updating loop counters, checking loop conditions) to the total energy consumption. In addition, we still execute multiple iterations of the measurement loop to increase the duration of the microbenchmark, reducing the contribution of possible ramp-up and ramp-down of the power consumption of the GPU. Thanks to the warm-up loop, all memory accesses to the targeted cache level should be tagged as *HIT*s during the execution of the measurement loop.

Thus, for each measurement point, we execute the microbenchmarks twice: once with only the warm-up loop, and once with both the warm-up and measurement loops. Then, we isolate the energy consumption and performance counter values of the measurement loop by subtracting the corresponding values of the warm-up loop.

5.5.2.2 Access granularity and kernel dimensions

Ideally, optimized kernels should exploit the cache line locality by accessing the memory in a coalesced and parallel way across multiple threads. Hence, using a single thread at a time to access memory can lead to overestimation of energy consumption per access. The proposed methodology assumes such ideal memory usage and is designed to assess a *lower bound* energy consumption for memory accesses. As mentioned in Section 5.4.3, to evaluate this lower bound, we have to take into account two effects: multiple threads accessing the same cache line in *parallel* and/or each thread accessing the elements of the cache line in a *subsequent* coalesced way.

Different memory levels can have different access granularities. For example, the granularity of the A100's L2 cache can be configured to 32, 64, or 128 bytes. This must be taken into account to prevent the generation of undesired HITs when targeting hardware-managed caches. To have consistently-strided accesses, we set the minimum stride of the pointer-chasing algorithm to the access granularity size of the targeted memory level. For example, the A100 GPU has a LOAD access granularity of 32 bytes (i.e., a *sector*, 4 `uint64_t` elements). Hence, we run the microbenchmarks with a stride of 32 bytes to access the array.

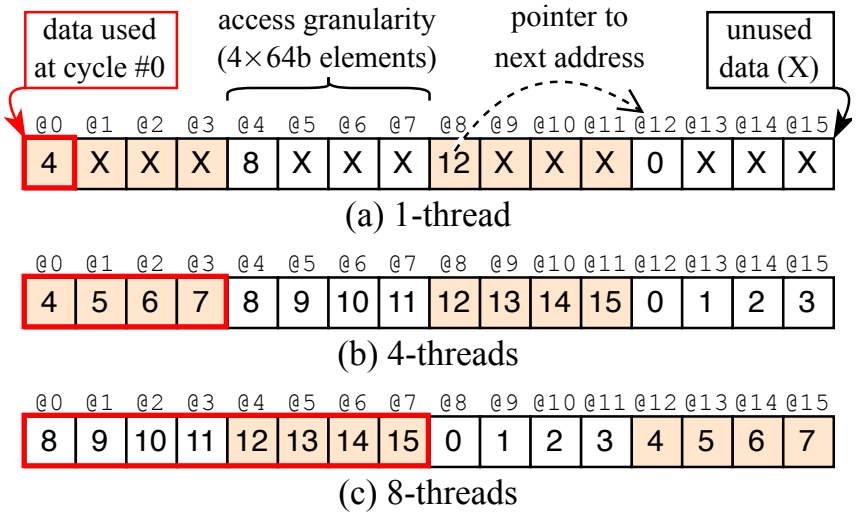


Figure 5.5: *Thread accesses of the microbenchmarks for different numbers of threads per block, adapting the stride accordingly*

However, with this stride, using only 1 thread per thread block would underuse the sector, loading only a fraction of the accessed elements (*see Fig. 5.5a*). As each LOAD operation of one thread will trigger a different memory access, this could lead to an unrealistically high energy consumption per access. Typically, multiple GPU threads should access the same sector at the same time (in our case, 4 threads to access a sector of 32 bytes to amortize the energy cost of accessing a sector, *see Fig. 5.5b*). Past this point, increasing the number of threads per block (*see Fig. 5.5c*) could still help reduce the energy consumption per memory transaction for two reasons: (1) it provides greater opportunities to parallelize memory transactions, thereby using multiple memory ports and reducing the execution time of the microbenchmark, and (2) it facilitates leveraging the fixed energy cost of using the targeted cache level (i.e., activating the necessary components to access the cache level).

The increase in the number of parallel accesses can also be achieved by increasing the number of thread blocks, leading to a further reduction in energy consumption per memory transaction. From a practical point of view, using more threads also increases the dynamic energy of the GPU, ultimately increasing the signal-to-noise ratio of the power measurements. Hence, in our experiments, we sweep the number of blocks and the number of threads per block until we reach a low saturation point for the energy per access. This evaluates the lower bound of the energy consumption of memory transactions.

5.5.2.3 Performance counters

The theoretical number of accesses during one iteration of the microbenchmarks is the array size divided by the stride size and the number of threads executed. Practically, we evaluate it using the following formula:

$$\#\text{accesses} = \frac{\text{arraysize} \times \#\text{blocks} \times \#\text{threadsperblock}}{\text{stride} \times \text{access granularity}}. \tag{5.7}$$

As described in Section 5.4.3, the total energy spent increases linearly with the number of accesses but also hides a static offset energy resulting from the activation of auxiliary components (e.g., memory controllers, SMs, etc.). Hence, we evaluate the total energy for multiple amounts of memory accesses and use linear regression to separate this energy offset from the energy cost of each memory transaction. For each measurement point, we run the profiling once and use the performance counters as a safety check to ensure that two conditions are met. First, we verify that the memory accesses are performed on the targeted memory level, by checking that the hit rate of the target cache level is 100% during the measurement loop. Second, we verify that the counted number of memory accesses is consistent with Eq. 5.7. Any difference in the evaluated number could indicate changes in the microarchitecture. Thus, parameters would have to be changed accordingly.

5.5.2.4 Power measurements

We manually add delays in the CPU code before and after the execution of the kernel to ensure we measure the static power consumption of the GPU during a period of inactivity (i.e., idle). While we use the internal power sensor of the GPU in our implementation of the methodology, external power tools could also be used with this alignment technique using delays.

In Fig. 5.6, we show a typical power trace of the A100 GPU, using base clock (1065 MHz), executing a kernel that is representative of the proposed microbenchmarks in its power consumption (i.e., multiply-add operations on a vector with millions of values), during 10 seconds. The power trace is annotated with markers (`t0` to `t5`), separating the different phases of execution.

1. Between `t0` and `t1`, no kernel is executing and no memory is allocated on the GPU, the GPU is in a *sleep* state ($\simeq 55$ W for the A100).
2. Between `t1` and `t2`, the memory is allocated on the GPU global memory and

the GPU enters an *idle* state. During this phase, we measure the static power consumption of the GPU.

3. Between `t2` and `t3`, during the 10s of *kernel execution*, the power consumption of the GPU increases depending on the kernel's load. During this phase, we identify the energy consumption above the idle power level as the *dynamic energy* (i.e., green zone in Fig. 5.6). We identify the rest of the energy consumption as the *static energy* (i.e., red zone in Fig. 5.6). When executing the microbenchmarks, the duration of this phase can be influenced by increasing or decreasing the number of iterations executed during the measurement loop.

4. Kernel execution ends at `t3`, and the GPU returns to an idle state.

5. Finally, at `t4`, the memory is deallocated and the GPU goes back to a sleep state until the end of the run at `t5`.

Experimentally, we use the NVIDIA Management Library (NVML) to collect power measurements from the GPU and run each measurement point multiple times (at least 3). Using base clock, we observe no difference in power consumption between the sleep and idle phases. However, this is not the case when using boost clock, where we observe a rise in the idle phase to around $80W$ and a *cooldown* phase at `t4`, where the power gradually lowers down to the power value observed in the sleep phase. Hence, we lock the GPU clock at base clock (1065 MHz) to help prevent power fluctuations that can arise from thermal throttling or changes in power modes.

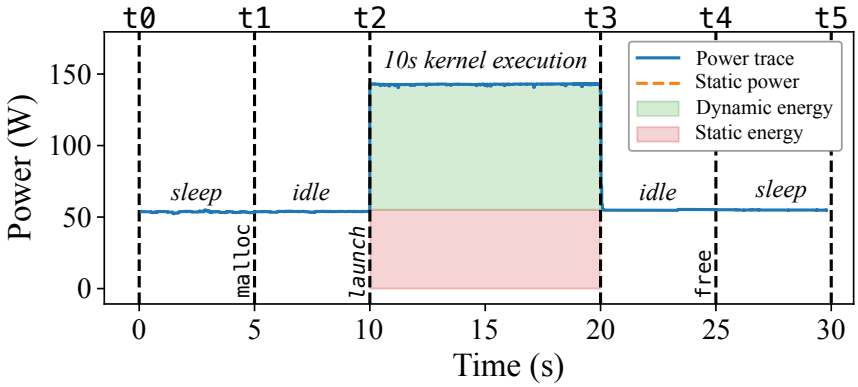


Figure 5.6: *Power trace of a 10s kernel execution on the A100 GPU with base clock, annotated with markers*

5.5.3 Evaluation phase

Once the energy consumption per memory access is isolated for each level of the memory hierarchy, the calibrated energy model can be used to provide energy breakdowns

for new applications. To this end, the new application has to be profiled with the same tools used during the calibration phase. However, while the calibration phase needs multiple runs of profiling (i.e., `for n_accesses` and `for n_runs` in Fig. 5.3), the evaluation phase only needs two runs, one for each profiling tool (i.e., NCU profiling and energy profiling). Our calibration methodology provides a lower-bound evaluation of the energy consumption and covers only the energy consumption of the data movement and storage. Hence, the measured dynamic energy will be higher than the sum of the energy consumption of the different memory levels. This difference includes the energy of the compute instructions (i.e., E_{COMPUTE} in Section 5.4) and additional energy from memory accesses exceeding the lower bound. In Section 5.7, we identify this difference as the remaining component of the dynamic energy (i.e., "Dyn. (rest)" in Figs. 5.11 and 5.12).

5.6 Implementation

In this section, we use the methodology described in Section 5.5 to calibrate the proposed energy model for the NVIDIA A100 GPU. We present the calibration results for the L1 cache, L2 cache, DRAM, and shared memory. Then, we challenge the consistency of our calibration results by evaluating the energy consumption of a *DIV* instruction using the same methodology.

5.6.1 Experimental setup

We use the NVIDIA CUDA Toolkit 12.1 to compile our microbenchmarks and run our experiments on a cloud server equipped with A100 paired with AMD EPYC 7343. We use the NVIDIA Management Library (NVML) to read the internal power sensor measurements and lock the clock frequency of the GPU to 1065 MHz (i.e., base clock). We use NVIDIA Nsight Compute CLI (ncu) to read the performance counters. We calibrate the energy model for the L1 cache, L2 cache, DRAM, and shared memory, using only the LOAD microbenchmark. We consider the energy of a STORE access energy consumption as the same as a LOAD access to a given memory. This respects our lower bound assumption, as the energy consumption of a STORE access is usually higher than a LOAD access [117]. For each measurement point, we run at least 3 iterations (i.e., `n_runs=3` in Fig. 5.3) and report the average energy consumption, standard deviation, and minimum/maximum values.

5.6.2 Parameter identification

According to the manufacturer's whitepaper [104], the A100 GPU is equipped with 108 SMs (each with 4 Streaming Multiprocessor Sub-Partitions (SMSPs)), 80GiB of HBM2 memory, 40MiB of L2 cache, and 192 kiB of combined shared memory and L1 cache. In Section 5.5, we show that MISSes can occur with smaller array sizes than the cache size. Hence, we choose the array sizes for our microbenchmarks small enough to fit in the targeted cache: 150 kB for the L1 cache, 250 kB for the L2 cache, and 50 MB for the DRAM. For shared memory, we use a 48 kB array size, as it is the maximum static allocation size per block on the A100 GPU [105].

5.6.3 Calibration

Fig. 5.7 shows the total energy consumption of the A100 GPU when running the *LOAD* microbenchmark with a varying number of memory accesses, targeting the L1 cache (i.e., 150 kB array size), using (a) one thread per block and (b) 1024 threads per block. We observe that the energy consumption increases linearly with the number of memory accesses. We use linear regression to calibrate the energy model and evaluate the energy consumption of each memory access. Linear regressions achieved r-squared scores of 0.96 and 0.99 for 1 and 1024 threads per block, respectively. The standard deviation of our measurements shows a maximum value of around 6J, which is acceptable with respect to the total energy consumption. Thus, we observe a higher precision of the calibration when using a higher number of threads per block.

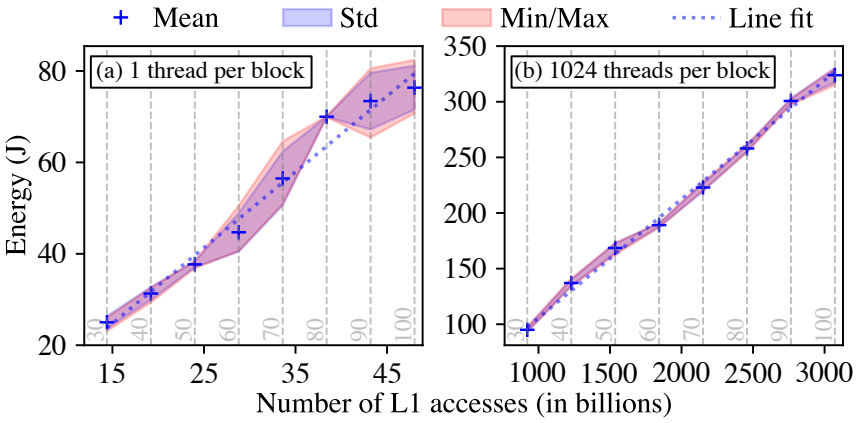


Figure 5.7: *Calibration traces of the L1 cache LOAD microbenchmark with (a) one thread per block, (b) 1024 threads per block*

Using the slope of the linear fit, we can estimate the energy consumption of each memory access, which is around 1600 pJ for the L1 cache using one thread per block

and 107 pJ for 1024 threads per block. As described in Section 5.5, the energy consumption of one memory access can vary depending on the number of threads per block. Hence, we repeat this calibration process for the L1 and L2 caches using multiple threads per block, up to 1024 threads per block (i.e., the maximum number of threads per block on the A100 GPU for 1D grids [105]). We present the results of this calibration in Fig. 5.8a, along with the L2 cache calibration in Fig. 5.8b. We can see that below 4 threads per block, the evaluated energy per access is similar (nearly 1600 pJ). With a low number of threads per block, we observe high signal-to-noise, hence the variation of the projections for the L2 cache (ranging between 8490 and 4260 pJ). As mentioned in Section 5.5.2.2, due to the access granularity, using less than 4 threads per block the proposed microbenchmark will always prompt the same number of sector accesses. When increasing the number of threads per block, we observe a plateau in the evaluated energy consumption of each memory access decreasing to around 107 pJ for L1 and 378 pJ for L2. These results show that ignoring multithreaded memory accesses leads to near 15× overevaluated energy per access (from 1600 to 107 pJ). We consider the lower bound of the energy consumption (i.e., 107 pJ for L1, 378 pJ for L2).

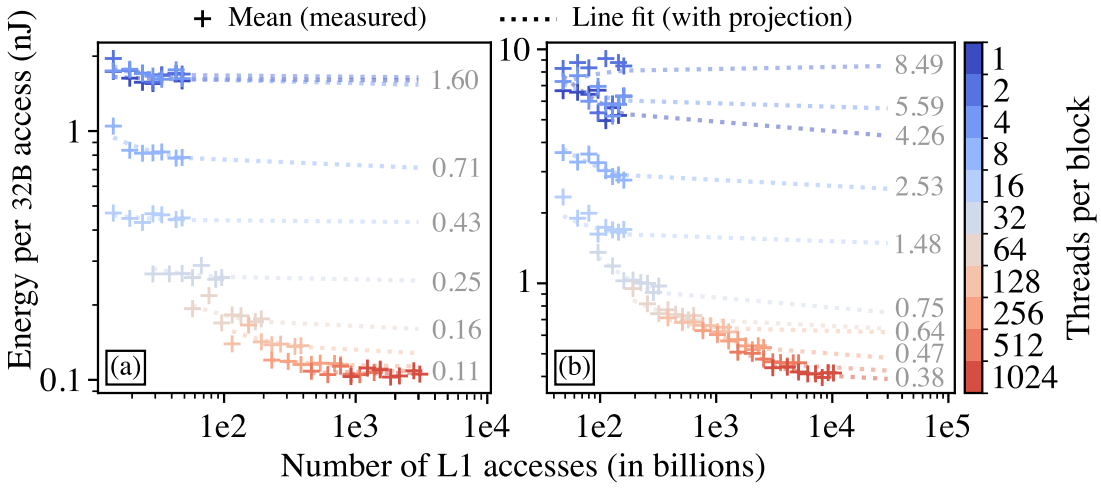


Figure 5.8: *L1 (a) and L2 (b) cache LOAD calibration results*

We repeat this calibration process for DRAM and shared memory (i.e., 50 MB and 48 kB array size, respectively) using multiple threads per block. We present the calibration results in Fig. 5.9, showing the projection of the energy consumption of each memory access for different numbers of threads per block.

Similarly to the L1 cache calibration, we observe that the energy consumption per access decreases when increasing the number of threads per block, for all memory levels. This shows the importance of considering the number of threads per block when evaluating the energy consumption of memory accesses on modern GPUs. We also

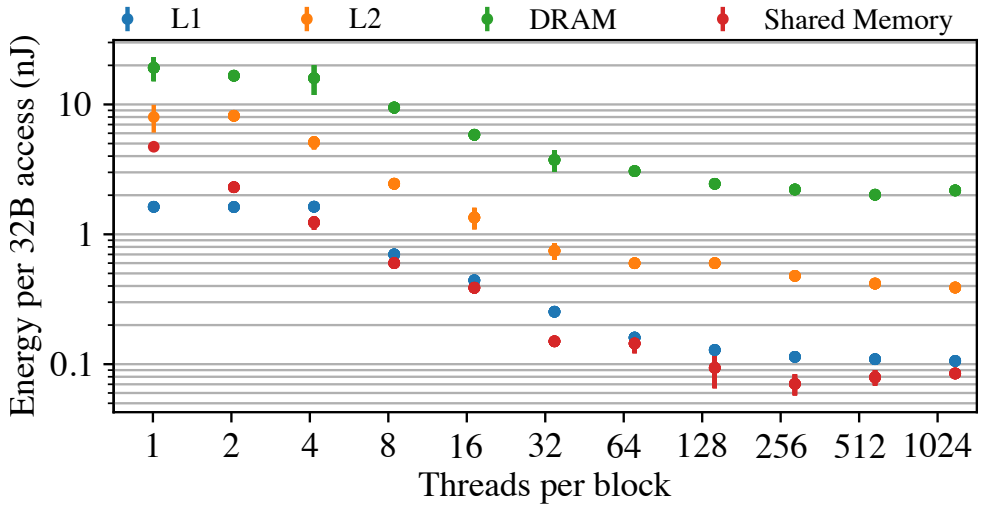


Figure 5.9: *L1, L2, DRAM and shared memory calibration results*

confirm that a saturation point is reached for the energy consumption of each memory access for all memory levels. This supports our lower-bound evaluation. Hence, we evaluate the lower bound of the energy consumption of one access at a given memory level by taking the minimum value across all evaluated numbers of threads per block. Hence, we evaluate the minimum energy consumption of a sector access (i.e., 32 bytes) to the shared memory, L1 cache, L2 cache, and DRAM to be: 82.1, 107, 368 and 2090 pJ, respectively. The difference in power consumption between L1 and shared memory (located on the same physical memory) can be explained by the difference in how the memory is addressed (i.e., shared memory is directly addressed, L1 cache has a tag stage checking if the data is present in the cache).

5.6.4 Cross-validation of the calibration results

The DRAM memory of the A100 uses HBM2 technology. Previous works estimate HBM2 memory energy consumption at 500 pJ per 32B access [117]. Using the proposed calibration methodology, we evaluate A100 DRAM energy consumption per access around 2090 pJ. Taking into account the additional hardware overhead of the DRAM and cache controllers, we consider this value to be consistent with the literature.

To challenge the consistency of our results, we use a modified version of the *LOAD* microbenchmark, with a supplemental *DIV* instruction after each memory access (i.e., *LOAD+DIV*), and perform the same calibration. We select the DIV instruction as it consumes significant energy compared to other compute instructions [13], making it simpler to isolate from the noise of the measurements. We evaluate the difference in energy consumption between the *LOAD* calibration and the *LOAD+DIV* calibration for the L1, L2, and DRAM memory levels. We present the results in Fig. 5.10.

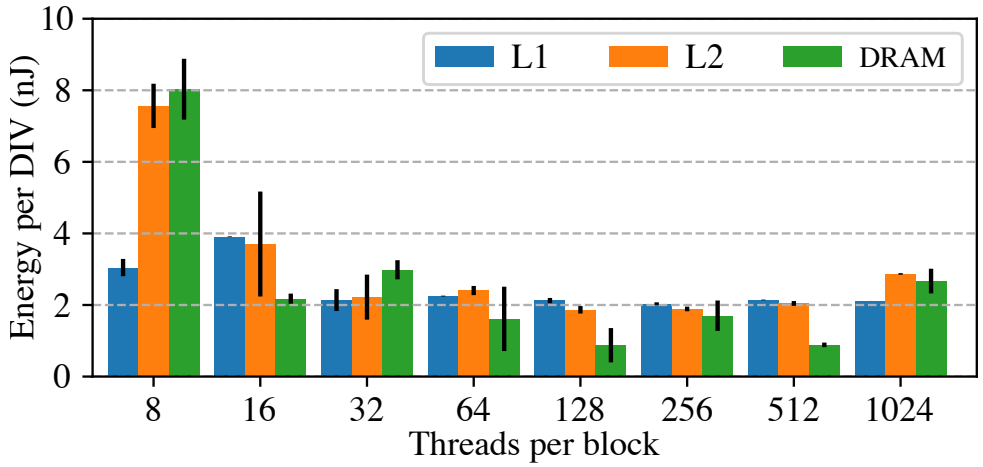


Figure 5.10: *DIV instruction energy evaluation based on the difference with the LOAD calibration results (L1, L2, and DRAM)*

We observe some variations in the evaluated energy consumption of the *DIV* instruction when using a small number of threads per block (i.e., with the highest signal-to-noise ratio). Above 8 threads per block, we observe that the energy consumption of the *DIV* instruction is consistent at around 2 nJ per instruction. This value is consistent with the literature, as the energy consumption of an unsigned division instruction was evaluated around 3.9 nJ on the previous generation of NVIDIA GPUs [13].

5.7 Evaluation of complete applications

We use the calibrated energy model to evaluate energy breakdowns for two real-world applications with increasing complexity: (1) a matrix multiplication (MatMul), varying the sizes of the multiplied matrices, and (2) a training iteration of a deep learning model (i.e., ResNet-50), using TensorFlow with different software optimizations. We perform 10 runs for each application and report less than 1% of standard deviation.

5.7.1 Matrix multiplication

We use a reference MatMul implementation from the NVIDIA CUDA samples (which does not use the tensor cores [106]). We evaluate the energy consumption for multiple square matrix sizes (from 512 to 4096) and show the breakdowns in Fig. 5.11.

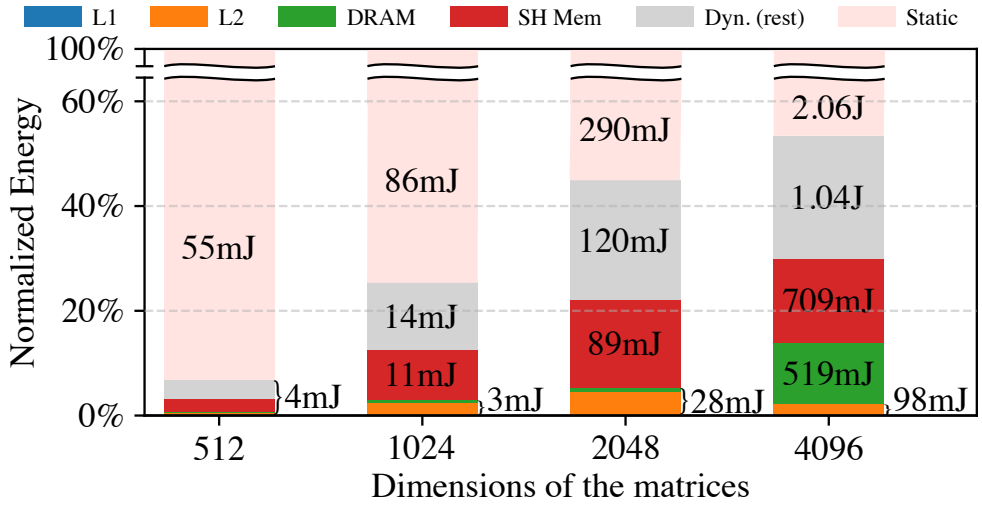


Figure 5.11: *Energy breakdown of MatMul for multiple sizes*

We observe that while the energy consumed during the execution of the kernel increases with the size of the matrices, it is dominated by the static energy consumption of the GPU (i.e., $> 50\%$). When ignoring the static energy consumption, we observe that a major part of the energy is consumed by moving data across the memory hierarchy (i.e., $> 50\%$ of the dynamic energy), with the shared memory being the most energy-consuming memory component.

The largest tested matrix multiplication (i.e., 4096×4096) has an arithmetic intensity of 1365 FLOPS/byte, which makes it compute-bound for the A100 GPU (i.e., arithmetic intensity $\gg 1$ FLOP/byte). Nevertheless, we observe that the energy consumption of data movement is still significant, representing around 30% of the total energy consumption of the kernel. This simple example shows the significant energy consumed in data movement and storage in the A100, even for a lower-bound evaluation of compute-bound workloads.

5.7.2 ResNet-50 training iteration

We use a reference implementation of ResNet-50 from the TensorFlow model garden [158] and evaluate the breakdown of the energy consumption of one training iteration using the A100 GPU. Similarly to Chapter 4, to select the training iteration, we run the training loop for a few iterations and select the first iteration with a stable execution time. Thus, avoiding the warm-up phase of the training loop and profiling a representative iteration of the rest of the training process. We run the training iteration comparing multiple software optimizations (i.e., full precision (FP32) vs. mixed precision (FP16), and eager execution vs. Just-In-Time (JIT) compilation using XLA) with batches of 512 images. We present the results of this evaluation in Fig. 5.12.

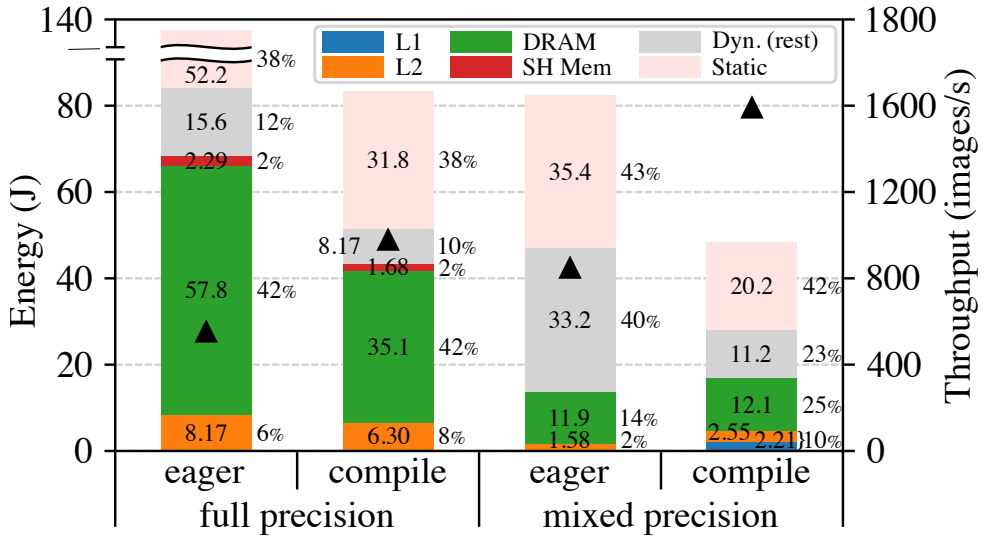


Figure 5.12: *Energy breakdown and throughput of ResNet50 using multiple software optimizations, executed using TensorFlow*

We make four observations from these results. First, the kernel's static energy is around 40% of its total energy consumption, regardless of the software optimization used. Second, data movement is the most energy-consuming component, representing the majority of the dynamic energy consumption (i.e., between 60% and 84%), except for the Mixed Precision (MP) eager execution (i.e., only 29%). Third, both of the tested software optimizations reduce the energy consumption of the kernel, showing a potential drop of 90 J going from full-precision eager to mixed-precision XLA JIT compilation. However, these optimizations influence the energy consumption breakdown differently. On the one hand, going from full-precision to mixed-precision greatly reduces the energy of data movement and storage. When using eager execution it represents a drop from 80% to 29% of the dynamic energy, and a drop from 83% to 60% when using XLA JIT compilation. On the other hand, XLA JIT compilation increases the proportion of memory energy consumption. It represents an increase from 29% to 60% of the dynamic energy consumption with mixed-precision, and from 80% to 84% when using full precision. Finally, we observe that DRAM greatly dominates the energy consumption of the memory hierarchy (above 70%). In contrast, the L2 cache represents between 2% and 8% of the total energy consumption. The L1 cache is negligible, representing less than 1% of the total energy consumption, except for the MP XLA JIT compilation, where it represents around 5%. Shared memory is also negligible, representing less than 2% of the total energy consumption for all tested configurations.

While the proposed methodology evaluates a lower bound of the data movement and storage energy, our results show that it still represents a significant part of the total energy consumption of the GPU, even for compute-bound workloads. On the one

hand, the calibration results highlight the importance of exploiting data locality when aiming for energy-efficient GPU applications. On the other hand, the energy breakdown results show that even a well-optimized deep learning model, exploiting data locality and reduced precision, is still dominated by the energy consumption of data movement. These results suggest that the long data paths of modern GPU architectures represent a significant energy bottleneck.

5.8 Summary

To conclude, this chapter addresses the limitations of existing methodologies to estimate an energy consumption breakdown for data movement and storage in modern GPUs. We propose an energy model along with a calibration methodology that can be used to estimate the energy per access to different levels of the GPU memory hierarchy.

First, in section 5.4, we present the energy model based on previous works [64, 68]. The model goes further into detailing the energy consumption of modern GPUs' memory hierarchy and takes into account parallel accesses to the memories, estimating a lower bound of the energy consumed by each memory access.

Second, in section 5.5, we propose a calibration methodology for the model using microbenchmarks. We describe the design choices of the microbenchmarks and the performance counters used as a sanity check for the calibration process. We also explain how to use the internal power sensor of the GPU with the proposed methodology.

Third, in section 5.6, we implement the proposed methodology, estimating the lower bound energy consumption of memory accesses on an NVIDIA A100 GPU. Our implementation evaluates the energy consumption of accesses to shared memory, L1 cache, L2 cache, and DRAM memory. While our implementation is restricted to an NVIDIA GPU, the general concepts can apply to other GPU brands that provide access to performance counters of their architecture. Furthermore, while we use an internal power sensor in our specific implementation, the methodology is also applicable using external power measurement tools [15]. For example, AMD GPU performance counters can be easily accessed using the AMD GPUPerf API [9] and the AMD GPU driver exposes power measurements to standard Linux tools (i.e., `hwmon sysfs` interface [78]).

We also challenge the calibration results by adding complexity to our microbenchmarks and adding DIV instructions to our LOAD microbenchmark. With this change, we reiterate the calibration process and compare the results with the original microben-

chark. With this comparison, we observe that the difference between the two calibrations is nearly constant, validating the consistency of our methodology.

Finally, in section 5.7, we evaluate the energy breakdowns for increasingly complex GPU workloads. We first evaluate the energy consumption of a matrix multiplication kernel with varying matrix sizes, showing that the majority ($> 50\%$) of the energy consumption of the GPU comes from its static power consumption. Furthermore, data movement across the memory hierarchy remains the dominant part of the GPU dynamic energy consumption, even for the compute-bound workload that this size matrix multiplication represents. Our last evaluation is the energy consumption of a training iteration of a Machine Learning model (i.e., ResNet-50) with different software optimizations. While software optimizations can lower the total energy consumption, we show that data movement remains the dominant part of the GPU dynamic energy consumption, with DRAM accesses being the main contributor (up to 84%).

CHAPTER 6

Conclusion

Contents

6.1 Summary of key contributions and findings **98**

6.1.1 Analysis of Machine Learning (ML) frameworks runtime for ML inference 98

6.1.2 Multi-level analysis of Graphics Processing Unit (GPU) utilization for ML training 99

6.1.3 Energy breakdown of data movement in the GPU architecture 99

6.2 Future work **100**

6.2.1 Profiling methodologies for GPU-accelerated ML workloads . 100

6.2.2 Software optimizations 102

6.2.3 Beyond the GPU architecture 102

6.3 Concluding remarks **104**

In summary, this thesis tackles performance analysis of the GPU architecture for ML workloads. Going top-down from the application level to the microarchitecture level, this thesis provides new methodologies and insights to understand the limitations of the current GPU architecture for both ML inference and training workloads. The provided analyses focus on evaluating how efficiently these workloads utilize the GPU resources and how the energy is spent in the GPU architecture.

In this chapter, we summarize the main contributions and findings, provide directions for future work, and finalize this thesis with some concluding remarks.

6.1 Summary of key contributions and findings

This thesis provides new methodologies to analyze the performance and energy of the GPU architecture running ML workloads. In this section, we summarize the three main contributions and findings of this thesis.

6.1.1 Analysis of ML frameworks runtime for ML inference

The first contribution of this thesis is an in-depth analysis of the TensorFlow eager execution runtime for ML inference on a Central Processing Unit (CPU)-GPU tandem. We provide a detailed description of the main steps followed by the TensorFlow eager execution runtime to run code on the CPU and GPU. With this description, we identify important metrics to analyze the overhead of the ML framework's runtime.

In addition, we propose a new approach to conduct an in-depth performance analysis of the inference process of three ML models (i.e., LeNet-5, ResNet-50, and BERT) for different batch sizes. The results of this analysis show that the runtime overhead of the TensorFlow eager execution is reduced considerably when operating with larger GPU kernels. However, this overhead could become significant when GPU kernel execution is not long enough to hide the ML framework's runtime latency, thus decreasing the proportion of total time when the GPU is in use. This work highlights the need to better identify bottlenecks in the runtime execution of ML frameworks.

This work resulted in a contribution to an international conference paper: Paul Delestrac, Lionel Torres, and David Novo. "Demystifying the TensorFlow Eager Execution of Deep Learning Inference on a CPU-GPU Tandem". In: *25th Euromicro Conference*

on Digital System Design (DSD). 2022. The source code and data used in this work are available on GitLab [144].

6.1.2 Multi-level analysis of GPU utilization for ML training

The second contribution of this thesis focuses on the analysis of the utilization of GPU computing resources for ML training workloads. From a description of an ideal GPU-accelerated ML training process, we identify relevant performance metrics across different levels of abstractions of the GPU architecture. We propose a new methodology to produce a coherent integration of the traces from multiple profiling tools.

We compare the execution of multiple ML training workloads to the ideal process and present our findings with seven key takeaways. We show that high utilization is typically achieved when looking at the GPU as a whole. However, our results show that the average instruction issue slot utilization remains below 50%, with tensor core instructions reaching less than 5.2%. This work highlights the need for advanced profiling to unravel the limitations of the GPU architecture for ML training workloads.

This work resulted in a contribution to an international conference paper: Paul Delestrac et al. "Multi-level Analysis of GPU Utilization in ML Training Workloads". In: *28th IEEE International Conference on Design, Automation & Test in Europe (DATE)*. 2024. The source code and the data used in this work are open-source and available on GitLab [99].

In addition, we plan to submit an extended version of this work to the ACM Transactions on Architecture and Code Optimization (TACO) journal in the coming months. This extension will include a new methodology to gather detailed power traces of GPU kernels for ML training workloads. This addition to the second contribution of the thesis is still a work in progress.

6.1.3 Energy breakdown of data movement in the GPU architecture

The third contribution of this thesis tackles the evaluation of the energy of data movement and storage in modern GPU architectures. We establish a basic energy model for the GPU architecture that focuses on the energy consumed by memory accesses. We propose a methodology to calibrate the proposed energy model by evaluating a lower bound of the energy consumed by memory accesses. The proposed methodology uses specific microbenchmarks, and performance counters, and leverages the internal

power sensor available in modern GPUs. We implement the proposed methodology and calibrate the energy model for an example GPU (i.e., NVIDIA A100), identifying the energy consumption of accesses to shared memory, L1 cache, L2 cache, and DRAM memory. We challenge the consistency of the calibration results by cross-validating with modified microbenchmarks to which we add additional compute instructions.

Finally, using the calibrated energy model, we evaluate breakdowns of the energy consumption for increasingly complex GPU workloads, up to a training iteration of an ML model (i.e., ResNet-50) with different software optimizations. While the proposed methodology evaluates a lower bound of the energy of data movement and storage, our results show that it still represents a significant part of the total energy consumed by the GPU, even for compute-bound workloads. On the one hand, the calibration results highlight the importance of exploiting data locality when aiming for energy-efficient GPU applications. On the other hand, the energy breakdown results show that even a well-optimized deep learning model, exploiting data locality and reduced precision, is still dominated by the energy consumption of data movement (up to 84%), with DRAM accesses being the main contributor.

This work resulted in a contribution to an international conference paper: Paul Delestrac et al. "Analyzing GPU Energy Consumption in Data Movement and Storage". In: *35th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2024. The source code and data used in this work are open-source and available on GitLab [4].

6.2 Future work

In this section, we provide some directions for future work based on the contributions and findings of this thesis.

6.2.1 Profiling methodologies for GPU-accelerated ML workloads

In this thesis, we propose new methodologies to analyze the execution of ML workloads on the GPU architecture. With the implementation of these profiling methodologies, we extract new insights that could not be obtained with traditional profiling tools. The results we present are a first step towards a more comprehensive profiling workflow for ML workloads. Here, we give directions for future work in this area:

ML frameworks runtime. While profiling tools provided by ML frameworks are useful for ML model development, we show in Chapter 3 that they do not provide a complete view of the runtime execution of the ML framework. Advanced profiling tools that can provide insights into the runtime execution of ML frameworks, such as the proposed TensorFlow eager runtime profiler [47, 144], could help improve both the ML models and the ML frameworks runtime. While the proposed profiler is limited to TensorFlow eager execution for inference, future work could extend this profiling methodology to training workloads, other runtime execution modes, and other ML frameworks.

GPU architecture metrics. In Chapter 4, we show the limits of existing profiling tools to provide a precise view of the GPU architecture utilization for ML training workloads. We propose a new methodology to integrate multiple profiling tools to analyze the GPU architecture utilization. Future work could extend this methodology to provide insights on additional metrics.

For example, Nsight Compute provides an estimation of the instruction issue stall reasons using sampling. This sampling methodology is limited to a small number of samples gathered from a small proportion of the GPU's Streaming Multiprocessors (SMs). Using assembly code instrumentation or simulation may provide paths for other profiling approaches which could result in a more precise view of the instruction issue stall reasons across the GPU architecture.

Another example would be to integrate power and energy analyses into the profiling methodology to compare the energy efficiency of different ML training workloads. To provide further insights energy consumption values could be provided at the kernel level. On the one hand, this could be measured using multiple power traces of a given workload to gather enough samples to provide a reliable energy evaluation. On the other hand, such energy value could be evaluated using a calibrated energy model, as proposed in Chapter 5 for data movement energy and in related work for compute energy [13]. This would also provide insights into how the energy is spent in the GPU architecture when running such kernels.

Other metrics could also be explored, specific to components inside the GPU architecture. For example, for a more memory-focused analysis, the methodology from Chapter 4 could be extended to monitor the memory controller modes. Thus, providing more fine-grained insights into the utilization of a specific component of the architecture during execution.

6.2.2 Software optimizations

While this is not explored in this thesis, we believe that the proposed profiling methodologies and new metrics could be used as guidance for software optimizations. In Chapters 4 and 5, we show that Just-In-Time (JIT) compilation can help achieve higher GPU utilization and lower energy consumption. Recent efforts on domain-specific compilers, such as Multi-Level Intermediate Representation (MLIR) [87], have shown promising results when used to optimize the execution of ML workloads. Hence, future work could use metrics such as the instruction issue slot utilization and the energy breakdown when exploring software optimizations with such compilers.

6.2.3 Beyond the GPU architecture

While software optimizations can help reach the high-performance potential of the modern GPU architecture, we show in this thesis (see Chapters 4 and 5) that this paradigm is reaching a saturation point. The scientific community has been exploring new architectures that could better serve ML workloads for years. Here, we give an overview of some of the architectures explored in the literature. Finally, we conclude this thesis by describing our first efforts to develop a vector accelerator in collaboration with international research groups from imec, KTH, and EPFL.

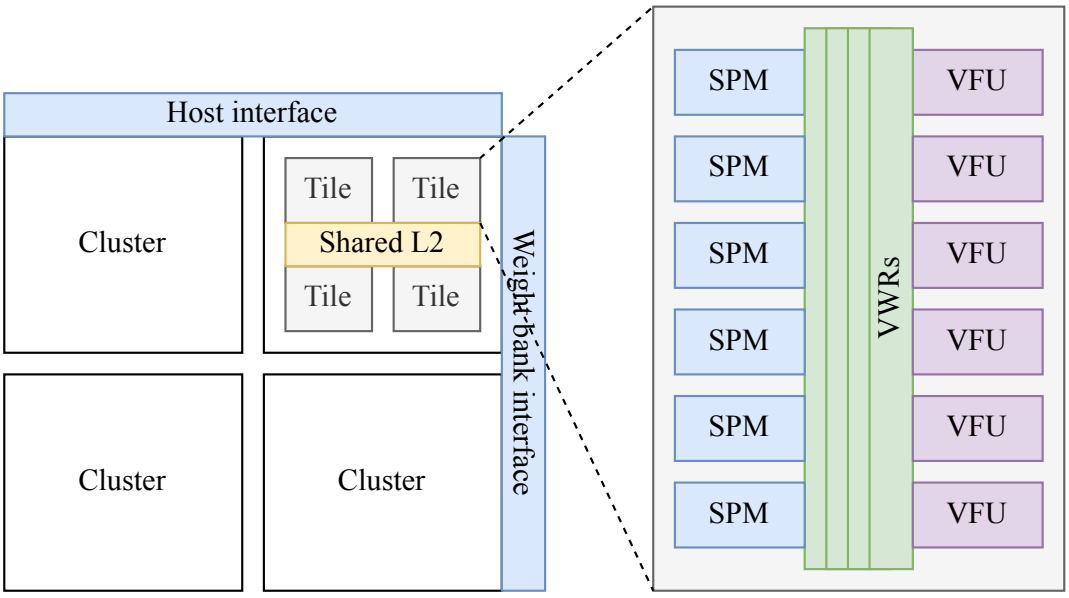
State-of-the-art ML accelerator architectures. In Chapter 5, we show that data movement remains the dominant part of the GPU dynamic energy consumption. Recent approaches try to build more efficient architectures by reducing the data movement that has to occur during execution. For example, architectures such as Google's Tensor Processing Units (TPUs) [72] bet on systolic execution units to reduce reads and writes to the main memory (i.e., *systolic arrays* [29, 33, 46, 53, 57, 112, 119]). Other architectures aim to reduce the distance between compute and memory units (i.e., *near-memory* processing architectures [67, 77, 84, 131]) or merge computation and memory units into a common block (i.e., *in-memory* processing architectures [11, 12, 35, 92, 132, 157]).

First efforts on developing a vector accelerator. In conjunction with this thesis work, we collaborate with international research groups from imec, KTH, and EPFL to explore the design of a vector accelerator. The first version of this design was published as a journal article in 2022 [156], where the imec team proposed a Design Space Exploration (DSE) framework for Convolutional Neural Network (CNN) mapping on tile-based accelerators. This contribution explored the integration of Analog In-Memory

Compute (AIMC) blocks with digital Vector Functional Units (VFUs) to build a hybrid architecture through a template Tiled ANalog In-memory Accelerator (TANIA).

However, our recent efforts focus on exploring a fully digital version of the template architecture presented in this paper. This new variant of the architecture leaves the analog AIMC blocks behind and integrates Very Wide Registers (VWRs) in-between the scratchpad memory units (SRAM) and the VFUs (illustrated in Fig. 6.1). VWRs use wide memory interfaces to reduce the number of memory accesses, thus aiming to reduce the energy consumption of the architecture [25].

LIRMM's contribution to the early steps of this project is to provide a system-level simulator that can be used to explore different configurations of the architecture (e.g., number of VFUs, VWRs, word width). We built the first version of this simulator in early 2024 using the Structural Simulation Toolkit (SST) [137]. While this work has not resulted in a publication yet, the collaboration between imec, KTH, EPFL, and LIRMM is ongoing. We are confident that this research path will result in major contributions to the community.



(a) Accelerator composed of multiple clusters, each with multiple tiles and a shared L2 cache

(b) Tile composed of multiple SPMs, VWRs and VFUs

Figure 6.1: *All-digital implementation of the TANIA architecture template*

6.3 Concluding remarks

In this thesis, we provide new profiling methodologies, metrics, and insights on the performance and energy consumption of GPU-accelerated ML inference and training workloads. First, we analyze an ML framework's runtime execution for inference workloads, using TensorFlow eager execution as a case study. We show that the runtime overhead of the ML framework can become significant when the GPU kernel execution is not long enough to hide the runtime latency. Second, we analyze the utilization of the GPU computing resources for ML training workloads at multiple levels of abstraction. Our results show that the average instruction issue slot and tensor core utilization remain low, despite high GPU utilization reported by traditional profiling tools. Finally, we evaluate the energy consumption of data movement in the GPU architecture and show that data movement remains the dominant part of the GPU dynamic energy consumption for ML training workloads.

We believe our results uncover some of the limitations of the current GPU architectures and motivate the need for more advanced profiling to further unravel these limitations. We hope that the methodologies and insights provided in this thesis will help the community in building more efficient ML workloads and new architecture paradigms to support them.

Bibliography

[1] International Energy Agency (IEA). *Electricity 2024*. 2024. URL: https://www.iea.org/reports/electricity-2024 (cit. on p. 2).

[2] Martín Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 2016 (cit. on pp. 3, 13, 26, 38, 77).

[3] Hamdy Abdelkhalik et al. "Demystifying the NVIDIA Ampere Architecture through Microbenchmarking and Instruction-level Analysis". In: *Proceedings of HPEC*. IEEE. 2022.

[4] *ADAC GPU Power Experiments*. 2024. URL: https://gite.lirmm.fr/adac/gpu-power-experiments (cit. on p. 100).

[5] Akshay Agrawal et al. "TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning". In: *Proceedings of Machine Learning and Systems*. 2019 (cit. on p. 26).

[6] Gargi Alavani et al. "Program Analysis and Machine Learning–based Approach to Predict Power Consumption of CUDA Kernel". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 8.4 (2023), pp. 1–24 (cit. on pp. 6, 73, 76, 77).

[7] *All symbols in TensorFlow 2*. URL: https://www.tensorflow.org/api_docs/python/tf/all_symbols (cit. on p. 29).

[8] Tyler Allen and Rong Ge. "Characterizing power and performance of GPU memory access". In: *Proceedings of E2SC*. IEEE. 2016 (cit. on pp. 73, 76, 77).

[9] AMD. *AMD GPU Open: GPUPerf API*. 2024. URL: https://gpuopen.com/gpuperfapi/ (cit. on pp. 69, 95).

[10] *An in-depth look at Google's first Tensor Processing Unit*. URL: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[11] Shaahin Angizi et al. “Cmp-pim: an energy-efficient comparator-based processing-in-memory neural network accelerator”. In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6 (cit. on p. 102).

[12] Shaahin Angizi et al. “MRIMA: An MRAM-based in-memory accelerator”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.5 (2019), pp. 1123–1136 (cit. on p. 102).

[13] Yehia Arafa et al. “Verified instruction-level energy consumption measurement for NVIDIA GPUs”. In: *Proceedings of CF*. ACM, 2020 (cit. on pp. 73, 76–79, 91, 92, 101).

[14] The Linux Kernel Archives. *perf*. URL: https://perf.wiki.kernel.org/index.php/Main_Page (cit. on p. 4).

[15] Daniel Bedard et al. “Powermon: Fine-grained and integrated power monitoring for commodity computer systems”. In: *Proceedings of SoutheastCon*. IEEE. 2010 (cit. on pp. 77, 95).

[16] *BERT base model (uncased)*. URL: https://huggingface.co/bert-base-uncased (cit. on p. 38).

[17] Nicola Bombieri et al. “MIPP: A microbenchmark suite for performance, power, and energy consumption characterization of GPU architectures”. In: *Proceedings of SIES*. IEEE, 2016 (cit. on pp. 73, 76, 77, 82).

[18] Amirali Boroumand et al. “Google workloads for consumer devices: Mitigating data movement bottlenecks”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 2018, pp. 316–331 (cit. on p. 8).

[19] Halima Bouzidi et al. “Performance prediction for convolutional neural networks on edge gpus”. In: *Proceedings of the 18th ACM International Conference on Computing Frontiers*. 2021, pp. 54–62 (cit. on p. 69).

[20] Lorenz Braun et al. “A simple model for portable and fast prediction of execution time and power consumption of GPU kernels”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 18.1 (2020), pp. 1–25 (cit. on pp. 6, 73, 76, 77).

[21] Robert A Bridges, Neena Imam, and Tiffany M Mintz. “Understanding GPU power: A survey of profiling, modeling, and simulation methods”. In: *ACM Computing Surveys (CSUR)* 49.3 (2016), pp. 1–27.

[22] Tom Brown et al. “Language models are few-shot learners”. In: *Proceedings of NeurIPS*. 2020.

[23] Martin Burtscher, Ivan Zecena, and Ziliang Zong. "Measuring GPU power with the K20 built-in sensor". In: *Proceedings of GPGPU Workshop*. ACM, 2014 (cit. on pp. 76, 78).

[24] Ermao Cai et al. "Neuralpower: Predict and deploy energy-efficient convolutional neural networks". In: *Asian Conference on Machine Learning*. PMLR. 2017, pp. 622–637 (cit. on p. 76).

[25] Francky Catthoor et al. *Ultra-low energy domain-specific instruction-set processors*. Springer Science & Business Media, 2010 (cit. on p. 103).

[26] Anantha P Chandrakasan, Samuel Sheng, and Robert W Brodersen. "Low power techniques for portable real-time DSP applications". In: *The Fifth International Conference on VLSI Design*. IEEE Computer Society. 1992, pp. 203–204 (cit. on p. 2).

[27] Shuai Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *Proceedings of IISWC*. IEEE. 2009, pp. 44–54 (cit. on pp. 72, 73, 77).

[28] Yu-Hsin Chen et al. "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2019).

[29] Yu-Hsin Chen et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks". In: *IEEE journal of solid-state circuits* 52.1 (2016), pp. 127–138 (cit. on pp. 2, 102).

[30] Jianmin Chen et al. "Statistical GPU power analysis using tree-based methods". In: *Proceedings of IGSC*. IEEE. 2011 (cit. on p. 72).

[31] Tianqi Chen et al. "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *arXiv preprint arXiv:1512.01274* (2015) (cit. on p. 26).

[32] Tianqi Chen et al. "TVM: end-to-end optimization stack for deep learning". In: *arXiv preprint arXiv:1802.04799* 11.2018 (2018), p. 20 (cit. on p. 4).

[33] Yunji Chen et al. "Dadiannao: A machine-learning supercomputer". In: *Proceedings of the International Symposium on Microarchitecture*. 2014 (cit. on pp. 2, 102).

[34] Sharan Chetlur et al. *cuDNN: Efficient primitives for deep learning*. 2014 (cit. on pp. 3, 14, 31, 48, 73, 77).

[35] Ping Chi et al. "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory". In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 27–39 (cit. on p. 102).

[36] Steven WD Chien et al. "tf-Darshan: Understanding fine-grained I/O performance in machine learning workloads". In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2020, pp. 359–370 (cit. on p. 22).

[37] Google Chromium. *Catapult Project*. URL: https://chromium.googlesource.com/catapult (cit. on p. 22).

[38] Cody Coleman et al. "Dawnbench: An end-to-end deep learning benchmark and competition". In: *Training* 100.101 (2017), p. 102 (cit. on p. 67).

[39] NVIDIA Corporation. *Nsight Compute Occupancy Calculator*. URL: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator (cit. on p. 19).

[40] NVIDIA Corporation. *Nsight Compute Replay*. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#replay (cit. on pp. 24, 55).

[41] NVIDIA Corporation. *NVIDIA CUDA Programming Guide: Compute Capabilities*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities (cit. on p. 18).

[42] NVIDIA Corporation. *NVIDIA Visual Profiler*. URL: https://developer.nvidia.com/nvidia-visual-profiler (cit. on p. 23).

[43] *CUDA, release: 10.2.89*. URL: https://developer.nvidia.com/cuda-toolkit.

[44] Scott Cyphers et al. "Intel ngraph: An intermediate representation, compiler, and executor for deep learning". In: *arXiv preprint arXiv:1801.08058* (2018) (cit. on p. 4).

[45] William J Dally, Stephen W Keckler, and David B Kirk. "Evolution of the graphics processing unit (GPU)". In: *IEEE Micro* 41.6 (2021), pp. 42–51 (cit. on p. 2).

[46] Saptarsi Das et al. "A systolic dataflow based accelerator for CNNs". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2020, pp. 1–5 (cit. on p. 102).

[47] Paul Delestrac, Lionel Torres, and David Novo. "Demystifying the TensorFlow Eager Execution of Deep Learning Inference on a CPU-GPU Tandem". In: *25th Euromicro Conference on Digital System Design (DSD)*. 2022 (cit. on pp. 98, 101).

[48] Paul Delestrac et al. "Analyzing GPU Energy Consumption in Data Movement and Storage". In: *35th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2024 (cit. on p. 100).

[49] Paul Delestrac et al. "Multi-level Analysis of GPU Utilization in ML Training Workloads". In: *28th IEEE International Conference on Design, Automation & Test in Europe (DATE)*. 2024 (cit. on p. 99).

[50] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv preprint arXiv:1810.04805* (2019) (cit. on pp. 38, 56).

[51] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. "TOP500 supercomputer sites". In: (1997) (cit. on p. 2).

[52] Nan Du et al. "Glam: Efficient scaling of language models with mixture-of-experts". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 5547–5569 (cit. on p. 12).

[53] Zidong Du et al. "ShiDianNao: Shifting vision processing closer to the sensor". In: *Proceedings of the 42nd annual international symposium on computer architecture*. 2015, pp. 92–104 (cit. on p. 102).

[54] Radwa Elshawi et al. "DLBench: a comprehensive experimental evaluation of deep learning frameworks". In: *Cluster Computing* (2021) (cit. on p. 26).

[55] Wu-chun Feng and Kirk Cameron. "The green500 list: Encouraging sustainable supercomputing". In: *Computer* 40.12 (2007), pp. 50–55 (cit. on p. 2).

[56] Eva García-Martín et al. "Estimation of energy consumption in machine learning". In: *Journal of Parallel and Distributed Computing* 134 (2019), pp. 75–88 (cit. on p. 2).

[57] Hasan Genc et al. "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2021, pp. 769–774 (cit. on p. 102).

[58] James Gleeson et al. "RL-Scope: Cross-stack Profiling for Deep Reinforcement Learning Workloads". In: *Proceedings of MLSys*. 2021 (cit. on pp. 6, 24, 44, 67, 68).

[59] Google. *Chrome Tracing*. URL: https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview (cit. on p. 22).

[60] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org (cit. on p. 31).

[61] João Guerreiro et al. "GPU static modeling using PTX and deep structured learning". In: *IEEE Access* 7 (2019), pp. 159150–159161 (cit. on pp. 6, 73, 76, 77).

[62] Horace He. *Making Deep Learning Go Brrrr From First Principles*. 2022. URL: https://horace.io/brrr_intro.html (cit. on p. 51).

[63] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016 (cit. on pp. 38, 56).

[64] Sunpyo Hong and Hyesoon Kim. “An integrated GPU power and performance model”. In: *Proceedings of ISCA*. ACM, 2010 (cit. on pp. 6, 73, 76–79, 95).

[65] Bodun Hu and Christopher J Rossbach. “Altis: Modernizing gpgpu benchmarks”. In: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2020, pp. 1–11 (cit. on p. 72).

[66] Quentin Huppert et al. “Memory hierarchy calibration based on real hardware in-order cores for accurate simulation”. In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021 (cit. on p. 81).

[67] Ranggi Hwang et al. “Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 968–981 (cit. on p. 102).

[68] Canturk Isci and Margaret Martonosi. “Runtime power monitoring in high-end processors: methodology and empirical data”. In: *Proceedings of MICRO*. IEEE. 2003 (cit. on pp. 73, 78, 95).

[69] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Vol. 1. Wiley New York, 1991 (cit. on p. 4).

[70] Jim Jeffers and James Reinders. *High performance parallelism pearls volume two: multicore and many-core programming approaches*. Morgan Kaufmann, 2015 (cit. on p. 79).

[71] Zhe Jia et al. *Dissecting the NVIDIA volta GPU architecture via microbenchmarking*. Tech. rep. arXiv preprint arXiv:1804.06826, 2018 (cit. on p. 73).

[72] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12 (cit. on pp. 2, 102).

[73] Vijay Kandiah et al. “AccelWattch: A power modeling framework for modern GPUs”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 738–753 (cit. on pp. 72, 73).

[74] Svilen Kanev et al. “Profiling a warehouse-scale computer”. In: *Proceedings of the 42nd annual international symposium on computer architecture*. 2015, pp. 158–169 (cit. on p. 8).

[75] Stefanos Kaxiras and Margaret Martonosi. *Computer architecture techniques for power-efficiency*. Springer Nature, 2022.

[76] Liu Ke et al. "Near-memory processing in action: Accelerating personalized recommendation with axdimm". In: *IEEE Micro* 42.1 (2021), pp. 116–127.

[77] Liu Ke et al. "Recnmp: Accelerating personalized recommendation with near-memory processing". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 790–803 (cit. on p. 102).

[78] Linux Kernel. *`hwmon sysfs` interface*. 2024. URL: https://www.kernel.org/doc/Documentation/hwmon/sysfs-interface (cit. on p. 95).

[79] Mahmoud Khairy et al. "Accel-Sim: An extensible simulation framework for validated GPU modeling". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 473–486.

[80] David Kirk. "NVIDIA CUDA Software and GPU Parallel Computing Architecture". In: *Proceedings of the International Symposium on Memory Management*. 2007.

[81] Yuriy Kochura et al. "Batch size influence on performance of graphic and tensor processing units during training and inference phases". In: *Advances in Computer Science for Engineering and Education II*. Springer. 2020, pp. 658–668 (cit. on p. 41).

[82] Jack Kosaian et al. "Boosting the throughput and accelerator utilization of specialized cnn inference beyond increasing batch size". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 5731–5741 (cit. on p. 41).

[83] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects". In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 461–475 (cit. on p. 2).

[84] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 740–753 (cit. on p. 102).

[85] Paul Eric Landman. *Low-power architectural design methodologies*. University of California, Berkeley, 1994 (cit. on p. 2).

[86] Rasmus Munk Larsen and Tatiana Shpeisman. *TensorFlow graph optimizations*. 2019 (cit. on p. 14).

[87] Chris Lattner et al. "MLIR: Scaling compiler infrastructure for domain specific computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 2–14 (cit. on p. 102).

[88] Y. LeCun et al. “Backpropagation Applied to Handwritten ZIP Code Recognition”. In: *Neural Computation* (1989) (cit. on p. 38).

[89] Jingwen Leng et al. “GPUWattch: Enabling energy optimizations in GPGPUs”. In: *ACM SIGARCH computer architecture news* 41.3 (2013), pp. 487–498 (cit. on p. 72).

[90] Cheng Li et al. “The design and implementation of a scalable deep learning benchmarking platform”. In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 2020 (cit. on pp. 6, 44).

[91] Cheng Li et al. “XSP: Across-stack profiling and analysis of machine learning models on GPUs”. In: *Proceedings of IPDPS*. 2020 (cit. on pp. 6, 44, 55, 67, 68).

[92] Shuangchen Li et al. “Drisa: A dram-based reconfigurable in-situ accelerator”. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 2017, pp. 288–301 (cit. on p. 102).

[93] Jan Lucas and Ben Juurlink. “Mempower: Data-aware gpu memory power model”. In: *Architecture of Computing Systems*. Springer, 2019, pp. 195–207.

[94] Cheng Luo and Reiji Suda. “A performance and energy consumption analytical model for GPU”. In: *International Conference on Dependable, Autonomic and Secure Computing*. IEEE, 2011, pp. 658–665 (cit. on pp. 6, 73, 76, 77).

[95] Peter Mattson et al. “MLPerf training benchmark”. In: *Proceedings of MLSys*. 2020 (cit. on pp. 55, 58, 67).

[96] Xinxin Mei and Xiaowen Chu. “Dissecting GPU memory hierarchy through microbenchmarking”. In: *IEEE Transactions on Parallel and Distributed Systems* 28 (2016), pp. 72–86 (cit. on pp. 73, 81, 82).

[97] Christopher A Metz, Mehran Goli, and Rolf Drechsler. “Pick the Right Edge Device: Towards Power and Performance Estimation of CUDA-based CNNs on GPGPUs”. In: *Proceedings of DATE SLOHA Workshop*. 2021 (cit. on pp. 6, 73, 76, 77).

[98] Diksha Moolchandani, Anshul Kumar, and Smruti R Sarangi. “Performance and power prediction for concurrent execution on GPUs”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 19.3 (2022), pp. 1–27 (cit. on pp. 73, 77).

[99] *Multi-Level Analysis of GPU Utilization in ML*. 2024. URL: https://gite.lirmm.fr/adac/delestrac2024multilevel (cit. on p. 99).

[100] *MXNet documentation: Profiling MXNet Models*. URL: https://mxnet.apache.org/versions/master/api/python/docs/tutorials/performance/backend/profiler.html (cit. on pp. 4, 44).

[101] Hitoshi Nagasaka et al. "Statistical power modeling of GPU kernels using performance counters". In: *Proceedings of IGSC*. IEEE. 2010 (cit. on pp. 6, 73, 76, 77).

[102] S Narang and G Diamos. *DeepBench: Benchmarking deep learning operations on different hardware*. 2017 (cit. on p. 67).

[103] Maxim Naumov et al. "Deep Learning Recommendation Model for Personalization and Recommendation Systems". In: *CoRR* abs/1906.00091 (2019). URL: https://arxiv.org/abs/1906.00091 (cit. on p. 56).

[104] NVIDIA. *NVIDIA A100 GPU whitepaper*. URL: https://www.nvidia.com/en-us/data-center/a100/ (cit. on pp. 53, 81, 89).

[105] NVIDIA. *NVIDIA Ampere GPU Architecture Tuning Guide*. URL: https://docs.nvidia.com/cuda/ampere-tuning-guide/ (cit. on pp. 89, 90).

[106] NVIDIA. *NVIDIA CUDA Samples*. Version Version 12.4. URL: https://github.com/NVIDIA/cuda-samples (cit. on pp. 73, 77, 92).

[107] NVIDIA. *NVIDIA Deep Learning Examples*. URL: https://github.com/NVIDIA/DeepLearningExamples (cit. on p. 56).

[108] NVIDIA. *NVIDIA Fermi Architecture Whitepaper*. 2010. URL: https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf (cit. on p. 48).

[109] NVIDIA. *NVIDIA Volta Architecture Whitepaper*. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (cit. on pp. 3, 48, 81).

[110] *NVIDIA CUDA Profiling Tools Interface (CUPTI)*. 2024. URL: https://developer.nvidia.com/cupti (cit. on pp. 67, 68).

[111] *NVIDIA CUPTI*. URL: https://docs.nvidia.com/cupti (cit. on pp. 22, 23, 35, 55, 68).

[112] *NVIDIA Deep Learning Accelerator (NVDLA)*. 2024. URL: https://nvdla.org (cit. on p. 102).

[113] *NVIDIA Management Library (NVML)*. 2024. URL: https://developer.nvidia.com/nvidia-management-library-nvml (cit. on pp. 6, 67, 68).

[114] *NVIDIA Nsight Compute*. 2024. URL: https://developer.nvidia.com/nsight-compute (cit. on pp. 4, 23, 53–55, 57, 67, 68, 76).

[115] *NVIDIA Nsight Compute CLI*. URL: https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html (cit. on p. 23).

[116] *NVIDIA Nsight Systems*. 2024. URL: https://developer.nvidia.com/nsight-systems (cit. on p. 23).

[117] James Michael O'Connor et al. "Energy efficient high bandwidth DRAM for throughput processors". PhD thesis. UT Austin, 2021 (cit. on pp. 88, 91).

[118] Nathan Otterness and James H Anderson. "AMD GPUs as an alternative to NVIDIA for supporting real-time workloads". In: *32nd Euromicro conference on real-time systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020 (cit. on p. 2).

[119] Angshuman Parashar et al. "SCNN: An accelerator for compressed-sparse convolutional neural networks". In: *ACM SIGARCH computer architecture news* 45.2 (2017), pp. 27–40 (cit. on p. 102).

[120] Soyeon Park, Kyungwoon Cho, and Hyokyung Bahn. "Analysis of Thread Block Scheduling Algorithms for General Purpose GPU Systems". In: *Proceedings of CSDE*. 2021 (cit. on p. 52).

[121] Adam Paszke et al. "PyTorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* (2019) (cit. on pp. 3, 13, 26, 77).

[122] PyTorch. *Kineto Docs GPU Utilization*. URL: https://github.com/pytorch/kineto/blob/aac412e6e7be846062add5c84be0c3fb501c4378/tb_plugin/docs/gpu_utilization.md (cit. on p. 50).

[123] *PyTorch Automatic Mixed Precision*. URL: https://pytorch.org/tutorials/recipes/recipes/amp_recipe.html#automatic-mixed-precision (cit. on p. 56).

[124] *PyTorch Profiler*. 2024. URL: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html (cit. on pp. 4, 22, 44, 50, 54, 57, 67, 68).

[125] James Reinders. *VTune performance analyzer essentials*. Vol. 9. Intel Press Santa Clara, 2005 (cit. on p. 4).

[126] Albert Reuther et al. "AI and ML Accelerator Survey and Trends". In: *Proceedings of HPEC*. 2022 (cit. on p. 48).

[127] Crefeda Faviola Rodrigues, Graham Riley, and Mikel Luján. "SyNERGY: An energy measurement and prediction framework for Convolutional Neural Networks on Jetson TX1". In: *Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer ... 2018, pp. 375–382 (cit. on pp. 2, 76).

[128] Nadav Rotem et al. "Glow: Graph lowering compiler techniques for neural networks". In: *arXiv preprint arXiv:1805.00907* (2018) (cit. on p. 4).

[129] Amit Sabne. *XLA : Compiling Machine Learning for Peak Performance*. 2020 (cit. on pp. 4, 14).

[130] Alvaro Saiz et al. "Top-Down Performance Profiling on NVIDIA's GPUs". In: *Proceedings of IPDPS*. 2022 (cit. on pp. 6, 67, 68).

[131] Fabian Schuiki et al. "A scalable near-memory architecture for training deep neural networks on large in-memory datasets". In: *IEEE Transactions on Computers* 68.4 (2018), pp. 484–497 (cit. on p. 102).

[132] Ali Shafiee et al. "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars". In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 14–26 (cit. on p. 102).

[133] John Smith and Jane Doe. "A completely made-up title for testing purposes". In: *Journal of Fictional Research* 1.1 (2023), pp. 1–10.

[134] Samuel L Smith et al. "Don't decay the learning rate, increase the batch size". In: *arXiv preprint arXiv:1711.00489* (2017) (cit. on p. 51).

[135] Shuaiwen Song et al. "A simplified and accurate model of power-performance efficiency on emergent GPU architectures". In: *Proceedings of IPDPS*. IEEE, 2013 (cit. on pp. 6, 73, 76, 77).

[136] Mani B Srivastava, Anantha P Chandrakasan, and Robert W Brodersen. "Predictive system shutdown and other architectural techniques for energy efficient programmable computation". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4.1 (1996), pp. 42–55 (cit. on p. 2).

[137] *SST: The Structural Simulation Toolkit*. 2024. URL: https://sst-simulator.org (cit. on p. 103).

[138] Dimitrios Stamoulis et al. "Hyperpower: Power-and memory-constrained hyper-parameter optimization for neural networks". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 19–24 (cit. on p. 76).

[139] Rico van Stigt, Stephen Nicholas Swatman, and Ana-Lucia Varbanescu. "Isolating gpu architectural features using parallelism-aware microbenchmarks". In: *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 2022, pp. 77–88.

[140] John A Stratton et al. "Parboil: A revised benchmark suite for scientific and commercial throughput computing". In: *Center for Reliable and High-Performance Computing* 127.7.2 (2012) (cit. on p. 72).

[141] Wei Sun et al. "Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors". In: *IEEE Transactions on Parallel and Distributed Systems* 34 (2022) (cit. on pp. 3, 48, 53, 73).

[142] *TensorBoard.dev*. URL: https://tensorboard.dev/ (cit. on pp. 22, 35).

[143] TensorFlow. *TensorFlow Lite: ML for Mobile and Edge Devices*. URL: https://www.tensorflow.org/lite (cit. on p. 15).

[144] *TensorFlow Eager Runtime Profiler Repository*. URL: https://gite.lirmm.fr/adac/tensorflow-eager-runtime-profiler (cit. on pp. 27, 35, 99, 101).

[145] *TensorFlow Guide: Create an op*. URL: https://www.tensorflow.org/guide/create_op (cit. on p. 29).

[146] *TensorFlow Keras Applications: ResNet50*. URL: https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet50 (cit. on p. 38).

[147] *TensorFlow Mixed Precision*. URL: https://www.tensorflow.org/guide/mixed_precision (cit. on p. 56).

[148] *TensorFlow Profiler: Profile model performance*. 2024. URL: https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras (cit. on pp. 4, 22, 48, 50, 54, 57, 67, 68).

[149] *TensorFlow Profiler: Profile model performance*. URL: https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras (cit. on pp. 34, 44).

[150] *TensorFlow, Large-scale machine learning on heterogeneous systems*. URL: https://github.com/tensorflow/tensorflow (cit. on p. 27).

[151] Snehil Verma et al. "Demystifying the MLPerf training benchmark suite". In: *Proceedings of ISPASS*. 2020 (cit. on pp. 3, 6, 67, 68).

[152] Oreste Villa et al. "NVBit: A dynamic binary instrumentation framework for nvidia gpus". In: *International Symposium on Microarchitecture*. IEEE/ACM, 2019, pp. 372–383.

[153] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76 (cit. on pp. 67, 68).

[154] Gene Wu et al. "GPGPU performance and power estimation using machine learning". In: *Proceedings of HPCA*. IEEE. 2015 (cit. on pp. 6, 73, 76, 77).

[155] Charlene Yang, Thorsten Kurth, and Samuel Williams. "Hierarchical Roofline analysis for GPUs". In: *Concurrency and Computation: Practice and Experience* 32.20 (2020) (cit. on p. 3).

[156] Simei Yang et al. "Aero: Design space exploration framework for resource-constrained cnn mapping on tile-based accelerators". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 12.2 (2022), pp. 508–521 (cit. on p. 102).

[157] Shihui Yin et al. "XNOR-SRAM: In-memory computing SRAM macro for binary/ternary deep neural networks". In: *IEEE Journal of Solid-State Circuits* 55.6 (2020), pp. 1733–1743 (cit. on p. 102).

[158] Hongkun Yu et al. *TensorFlow Model Garden*. 2020. URL: https://github.com/tensorflow/models (cit. on pp. 56, 93).

[159] Hui Zhang and Jeffrey Hollingsworth. "Understanding the performance of GPGPU applications from a data-centric view". In: *Proceedings of ProTools*. 2019 (cit. on pp. 6, 67, 68).

[160] Keren Zhou et al. "GPA: A GPU performance advisor based on instruction sampling". In: *Proceedings of CGO*. 2021 (cit. on pp. 6, 67, 68).