Composable Architecture Primitives for the Era of Efficient Generalization

by

Michael E. Davies

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: 3/15/2024

The dissertation is approved by the following members of the Final Oral Committee:

Karthikeyan Sankaralingam, Professor, Computer Sciences

Vikas Singh, Professor, Biostatistics and Medical Informatics

Aws Albarghouthi, Professor, Computer Sciences

Eftychios Sifakis, Professor, Computer Sciences

Mark Hill, Professor Emeritus, Microsoft

© Copyright by Michael E. Davies 2024

All Rights Reserved

For Franklin and Darwin.

ACKNOWLEDGMENTS

It might be cliché, but perhaps the most important lesson I learned in these six years is people matter – more than anything – in finding your way in life. I'd like to acknowledge the people in my life who collectively made it possible for me to complete this thesis. In addition to people, I want to acknowledge the two fuzzy cats I've dedicated this dissertation to: Franklin and Darwin who are depicted in Figure 0.1. I adopted Franklin and Darwin in the summer of 2020 just after the pandemic started. They are full of personality and have been with me through the majority of this research work. Franklin would often sit on my desk helping me write and Darwin would bring me his little crinkle balls as a gift.

Figure 0.1: The author's two cats: Darwin (Left) and Franklin (Right).

As for humans, I'd like to first thank my advisor, Prof. Karu Sankaralingam, whose wisdom and kindness kept me going even in the face of confusing results, frustrating paper reviews, and everything else that goes along with being a graduate student. I truly believe would not have made it here without his help.

Next, my parents, James and Debbie Davies, who have given me love and support my whole life. Who taught me how to be curious, how to take things apart, and how to put them back together. From the bottom of my heart, thank you for all you've done for me.

I'd like to thank my brother Dr. Daniel Davies, who I still look up to, to this day. He has always been there when I've needed someone to talk to, and I will always admire his intelligence, talent, and wit. I'd also like to thank my brother Andrew Davies and sister Alexa Duda who have always given me sage advice when I need it.

Finally, I'd like to thank my best friends who I have spent so much time with here in Madison:

Brandon, Kristel, Erin, Nate and Hayley who have made my daily life orders of magnitude brighter through these years. They have shared in some of my highest and lowest moments and for that I will always be grateful. I'd also like to thank many others who have been a part of my life during my graduate career: Kristen, Lia, Tully, Zach, Jared, Marisa, Tom, Steve, Alek and many more.

There are so many who I'd like to thank I could fill this entire thesis with names alone. To acknowledge all these wonderful people, named or not, I will use "we" throughout this thesis to signify the collective group of people described here who made this thesis possible.

Thank you.

-Mike

P.S. The original draft of these acknowledgements were written on 10/17/2023 at 2am because I can't sleep.

Contents iv

List of Tables vi

List of Figures viii

1 Introduction 1

1.1 Contributions . . . 2

1.2 Organization . . . 4

1.3 Artifacts Developed and Used in this Research . . . 4

2 GPU Longitudinal Study 6

2.1 Background of DL & Related Work . . . 8

2.2 Methodology . . . 10

2.3 Software Perspective . . . 16

2.4 GEMM Deep Dive . . . 20

2.5 Application Informed Hardware Analysis . . . 24

2.6 Path Forward . . . 31

2.7 Conclusion . . . 35

3 Understanding the Architectural Implications of Deep Learning 36

3.1 Related Work . . . 39

3.2 Characterization of DL Applications . . . 41

3.3 Galileo Architecture . . . 43

3.4 Demonstrating Coverage . . . 47

3.5 Methodology . . . 50

3.6 Results . . . 51

3.7 Conclusion . . . 60

4 Benefits of Technology Scaling 61

4.1 Definitions and Modeling Equations 62
4.2 Model validation 63
4.3 Context and Timing 66

5 Spatial Fusion **68**
5.1 Program Behavior Motivation & Overview 72
5.2 Kitsune Design 75
5.3 Evaluation 83
5.4 Kitsune Software Prototype 90
5.5 Related Work 93
5.6 Conclusions 95

6 Conclusion **96**
6.1 Reflections, Implications and Open Questions 96
6.2 Concluding Thoughts 98

A Overview of Deep Learning **99**
A.1 Operators and Gradients 100

B GPU Hardware **103**

C Acronyms **105**

Bibliography **106**

LIST OF TABLES

1.1 Dissertation Organization and Relation to Author's Prior Work. 3
1.2 Artifacts developed for this work. 4
2.1 Mapping of insights to unifying execution behaviors. Definition of behaviors in Section 2.6 7
2.2 Applications in this study. Applications were primarily chosen based on diversity and citation count (as a proxy for popularity). 8
2.3 System configurations 12
2.4 Top Operator Contributions by Runtime 14
2.5 Number of Total and Unique CUDA Kernels 15
2.6 Kernel level breakdown of apps showing percentage of runtime spent in GEMM kernels, number of unique GEMM kernels, and the number of additional unique kernels to reach 80% and 90% runtime coverage. 17
2.7 Selected GEMMs sorted by AMI. 21
2.8 Selection of interesting kernels classified by hardware state and their associated framework operator. 26
2.9 Performance of MeshGraphNets GEMM shapes. 32
3.1 Summary of related work's solutions to DL acclerator challenges. 37
3.2 Summary of key behaviors in DL operator shapes. 39
3.3 Qualitative comparison of General Purpose processor, GPU and AI Accelerators. 40
3.4 Summary of Galileo's differences to related work. 40
3.5 Summary of different properties of workloads 41
3.6 Comparison of Specs. 53
3.7 Analysis of Top 3-5 layers for each app. by op count 55
3.8 Analysis of related academic DL accelerators on basis of their efficiency when observed behaviors are present. 57
4.1 Estimates of Transistor costs by wccftech [193] 61
4.2 Tech Scaling 61
4.3 Model validation 64

4.4 Top Operators in MLPerf 65
4.5 Fraction of runtime based on operator type 65

5.1 Description of selected applications and subgraphs for pipelining. Note: operator shapes elided in these diagrams. Different instances may have different shapes for each layer. . 72
5.2 Common patterns found across our applications. 72
5.3 Summary of fusions and traffic reductions. 84
5.4 Dataflow coverage of applications in terms of number of operators, time and traffic. . . 91
5.5 Measured traffic reduction from NVIDIA NCU. 92

LIST OF FIGURES

0.1 The author's two cats: Darwin (Left) and Franklin (Right). ii

2.1 Overview of the DL stack and tooling we used for profiling and extracting performance data. Thick border indicates instrumentation tooling. Dashed border indicated data that we gather. 9
2.2 Cumulative instruction count contribution for SASS opcodes for GEMM and Non-GEMM kernels. 16
2.3 Depiction of GEMM operation with NVIDIA CUTLASS (Adapted from [72]). This shows the tiling strategy for threadblocks (CTAs), Warps, and TensorCore instructions ("atoms"). 20
2.4 Histogram of GEMM Compute utilization on each GPU. 21
2.5 Scatter plot of GEMM shapes showing Utilization on A100 vs AMI. We show a sweep of shapes categorized by size of dimensions in addition to our observed shapes. 21
2.6 Hardware resource usage over fraction of runtime. 25
2.7 Per-application and average maximum resource usage by fraction of application runtime. Results ordered by increasing utilization. Red and Green points indicate bounds for average application utilization for 60% and 70% of runtime, respectively. 25
2.8 End-to-end speedup of each application by GPU generation. 27
2.9 Per-application timeline of hardware execution states. We observe many execution states are present with execution state changing rapidly (every $\approx 1-2$ kernels). 29
2.10 One GraphNetBlock from MeshGraphNets. 33
2.11 Performance opportunity by improving Compute-, Data-, and Dependence- Orchestration. Speedups over A100 as baseline. 33

3.1 Overview of the Galileo Architecture. 36
3.2 Comparison of Galileo and TDCC Architecture Design Space. 44
3.3 Organization of Galileo Tile. 44

3.4 Execution model and lower of graph to hardware primitives. (a) Shows how DL operators are lowered down to low-level primitives. (b) The execution model of an operator. 46

3.5 Mapping of Common Operators to the Galileo architecture 47

3.6 Use of transposed-loads in a small matrix-multiply. 48

3.7 Design Space Exploration of Galileo. Points of same color sweep from 1GHz-3GHz clock frequency with steps of 100MHz. 52

3.8 Galileo vs A100 across MLPerf at batch size = 16. Throughput in samp/sec, and Rel. Eff. is ratio of avg. pJ/op. 53

3.9 Overall performance sensitivity to Memory, Communication network, and Compute engine. 59

4.1 G3′ over G3 speedup with various values of r_l and γ. 64

4.2 World and China Semiconductor supply chain. Source [107]. 66

5.1 SM Utilization CDF for five popular DL applications. For a majority of the applications, greater than 50% of runtime is spent with a utilization less than 33% and 20% for inference and training, respectively. 69

5.2 Comparison of (a) Baseline bulk synchronous code with (b) Vertical fusion and (v) Spatial fusion. 69

5.3 Depiction of applications and the fusions we apply. 75

5.4 Depiction of the Kitsune spatial fusion compiler flow. 76

5.5 Running example of two subgraphs selected from (a) MeshGraphNets. (b) shows an MLP. (c) and (d) show the allocation and assignment. (e) shows a subgraph from the backward pass. 80

5.6 Queue design. Note: release routines are not shown for space reasons. They involve simple atomicAdd calls to update synchronization metadata and a CTA barrier with __syncthreads(). 82

5.7 Inference subgraph speedups including sensitivity to hardware resources. 85

5.8 Inference End-to-end Speedup over Bulk-Sync. 85

5.9 Training subgraph speedups including sensitivity. Dashed lines separate forward and backward passes. 86
5.10 Training End-to-end Speedup over Bulk-Sync. 89
5.11 All pairings of SIMT- and Tensor-heavy ops onto one SM. We show the individual and combined usage of each resource. Pairings are ordered by increasing combined usage for each resource / pairings are not the same ID across charts. 89
5.12 Performance of GPU atomics (top) and our synchronized queue (bottom). 90
5.13 Performance of KitsuneSW (software and hardware-accelerated) on A100 and H100 GPUs. Hatched bars are measured performance and all other values are modeled. . . . 93
5.14 Application level performance and traffic reduction with subgraphs implemented with KitsuneSW. Hatched bars are measured. 93

A.1 Operator Graph of Common DL Applications. 99

B.1 GPU Hardware overview. 103

COMPOSABLE ARCHITECTURE PRIMITIVES FOR THE ERA OF EFFICIENT GENERALIZATION

Michael E. Davies

Under the supervision of Professor Karthikeyan Sankaralingam
At the University of Wisconsin-Madison

We are in the age of AI, with rapidly changing algorithms and hardware. As Deep Learning (DL) applications have evolved, architects have turned to specialization to keep up with the insatiable compute that DL demands. Counter to this hardware evolution, DL demands that hardware support a vast number of diverse application behaviors and in addition, an inflexible *ossified* software interface in order to be viable. This dissertation motivates and adopts the guiding philosophy of "Efficient Generalization" – as opposed to "Specialization" – as a pathway for architects to support diverse DL application behavior while also catering to the ossified DL stack. We take a principled approach to identifying architecture and software primitives to enable performance and efficiency growth while paying careful attention to the restrictions that come with an ossified software stack and reaching the limits of technology scaling. We calibrate our understanding of modern architecture for DL by conducting a longitudinal study of how the DL revolution has panned out for GPUs – the dominant AI hardware platform of the day. Based on the insights from this study, we explore the fundamental architecture primitives needed to support DL's conflicting needs of programmability and high-performance. We use the knowledge of these architecture primitives to then tease apart the diminishing impacts of technology scaling. Combining all our findings about the role of technology, microarchitecture and architecture, we explore a novel execution model, spatial fusion, for DL on existing programmable architectures (GPUs) which can enable better utilization of on-chip resources leading to higher performance and power efficiency with only modest changes to the architecture.

1 INTRODUCTION

We are in the age of AI, with rapidly changing algorithms and hardware. As Deep Learning (DL) applications have evolved, the underlying hardware has evolved, too. Existing hardware platforms including graphics processors (GPUs) and conventional processors (CPUs) have added forms of specialized hardware blocks exemplified by NVIDIA's TensorCore [112], AMD's MatrixEngine [2], and Intel's Advanced Matrix Extensions (AMX) [59]. Exotic solutions include clean-slate architectures – many styles of which are being explored: GEMM acceleration including CGRA (many startups), systolic arrays (Google TPU), dataflow (Wave, Blaize, Graphcore and other startups), FPGA (LUT-based Xilinx, Versal, Systolic-array-based Lattice), spatial dataflow (SambaNova, Groq), other (Qualcomm AI-100, Inferentia - design details sparse) including some delineating of training vs inference, and within that support for particular types of DNNs (CNN, LSTM, GNNs etc.). Another cohort is inference optimized chips (with some architectural handicaps hampering training capability) like Tenstorrent (some form of spatial dataflow) [47], Groq (extreme slice-oriented spatial architecture, require extensive static scheduling) [45], Qualcomm AI-100 (8-core DSP with very wide vectors per core) [46], Baidu Kunlun [61], Xilinx Versal (wide vectors added to FPGA), and Alibaba's HanGuang [44].

In conjunction with this hardware evolution, the software "stack" that DL application developers use has become more mature with modern applications needing barely 1,000 lines of code. A byproduct of this maturity is the **ossification of the DL stack**, creating a heavy lift for new architectures to compete in the DL space – that is, new architectures must conform to the needs of the DL stack, not vice-versa. In parallel, the slowing and death of Moore's law has convinced computer architecture researchers to pursue *specialization* to maintain the performance and efficiency improvements previously afforded by technology scaling. *We observe for DL applications, these two influences run counter to each other. At best, this lends to a complex space of tradeoffs architects must navigate in order to provide performance wins while catering to the demands of DL software; and at worst, this becomes an insurmountable software lift for some highly specialized architectures.*

This dissertation identifies *"Efficient Generalization"* – as opposed to "specialization" – as a guiding philosophy for architecture research that is amenable to the trends in DL software and the death of Moore's law. We define "Efficient Generalization" as an approach to architecture that

acknowledges hardware doesn't exist in isolation from software and treats the demands of having vast behavior diversity as a first-class design constraint. In addition, the presence of an ossified software stack makes this approach more tenable than specialization. One concrete realization of efficient generalization is taking an existing programmable architecture with well understood execution semantics and augmenting it with architectural additions that retain programmability while providing high performance and efficiency. **This dissertation explores architecture-software primitives for the next generation of deep learning specialized accelerators to support performance and efficiency growth for this "Era of Efficient Generalization".**

1.1 Contributions

We first investigate the three-way intersection across domain, hardware features, and software stack complexity interactions by conducting a longitudinal study of state-of-art AI applications across three generations of GPU hardware to understand how the AI revolution has panned out so far, and how future AI hardware should evolve. Specifically, we answer three primary questions:

1. How does DL application behavior scale across hardware generations?
2. What are the software-compiler-hardware interactions responsible for enabling generational speedup or limiting performance? and
3. What can we learn from the state-of-art for where future architectures must focus? The closest related work is Google's TPUv4 inference retrospective [68] - which lacked detailed data on application, framework stack, or microarchitecture-level hardware utilization trends.

Motivated by our findings, we develop a clean-slate "minimal" architecture which eliminates much of the historical cruft of a GPU – I.e. mechanisms to facilitate and optimize performance for the SIMT execution model that was designed originally for graphics applications – and elucidate what is necessary in an architecture for DL. Through this, we elucidate the hardware and pure-architecture mechanisms responsible for both high performance DL and software amenability. Intriguingly, we find that a highly non-novel architecture belonging to a class of "Tiled Decoupled Control and Compute" accelerators called Galileo is sufficient for achieving high DL performance and bridging

Chap.	Topic	Author's Related Prior Work
1	GPU / DL Study	ASPLOS 2024
2	Architecture Primitives	In preparation for CACM
3	Contribution of technology scaling	In preparation for CACM
4	Spatial Fusion	OSDI 2024 in submission
N/A	DL Myths for Architects	In preparation for CACM

Table 1.1: Dissertation Organization and Relation to Author's Prior Work.

the ossified software stack gap. Galileo is only subtly different than other more complex designs, which we discuss in detail.

Moore's Law and Dennard scaling are dead from a practical standpoint of cost improvements [93], energy efficiency [192], and increasingly density improvements [188, 12, 133]. In light of this reality, we next study Galileo and GPU architectures with an eye towards disaggregating the contributions of technology scaling and architectural advancements. Through technology modeling, and detail simulation, considering the full MLPerf workload set, we show that a targeted AI chip (eliminating "cruft" like graphics, FP32, FP64, texture, the CUDA tax of bulk-synchronous execution) at 12nm, can easily match SOTA H100 built at 5nm and it's hypothetical successor which doubles performance.

We observe from our longitudinal study that modern DL accelerators – primarily GPUs – are substantially underutilized in some scenarios. From our study of Galileo, we find architecture alone can provide some benefits. To enable future improvements beyond what architecture alone can provide, software-architecture codesign must be considered. To this end, we study a software-architecture paradigm shift for modern GPUs based on the observation that the bulk-synchronous execution model currently employed is inefficient for modern DL applications. We propose a new synchronous dataflow-based execution model, Kitsune, which enables "Spatially Fusing" dependent operations. Kitsune draws on the lessons of minimalism we learn from our study of Galileo, showing that modest changes to the architecture combined with software-level changes can yield substantial speedup.

Chap.	Artifact	Public Link
1	GPU / DL Tools	https://github.com/VerticalResearchGroup/casio
2	Performance Model	https://github.com/VerticalResearchGroup/upcycle
3	Scaling Model	https://github.com/VerticalResearchGroup/upcycle
4	Spatial Fusion Prototype	https://github.com/medav/gpu-pipes
4	Spatial Fusion Model	Not publicly available

Table 1.2: Artifacts developed for this work.

1.2 Organization

The organization of this dissertation outlined in Table 1.1, along with the relation to the author's prior publications. Appendix C defines acronyms used throughout this document. We first discuss our longitudinal study of three GPU generations on DL workloads (Chapter 2). We then discuss what we learn from the insights of our study in relation to future AI accelerator architecture, and develop an efficient accelerator architecture, Galileo, which focuses on distilling out the hardware and architecture mechanisms which enable performance and programmability for DL (Chapter 3). After this, we use the Galileo architecture as the basis for teasing apart the contributions of architecture improvements and technology scaling for future DL hardware generations (Chapter 4). Finally, we study a hybrid software-architecture approach, called spatial fusion, to optimize DL on modern GPU platforms, introducing a new synchronous dataflow-based execution model for future GPUs (Chapter 5).

1.3 Artifacts Developed and Used in this Research

To conduct our research, we make use of several existing software tools, as well as develop new tools and performance models. Overall we use the A100 as the baseline SOTA GPU for our research which matches what was available as state-of-art for research use at the time this research was completed. Chapter 2 uses the CASIO Deep Learning suite which includes 10 recent highly cited DL applications, and uses three GPU hardware platforms. No simulators are used and the research is driven by source code, binary, and performance counter analysis. Chapter 3 uses applications from MLPerf. We develop a performance, power and area model for our proposed architecture on these workloads. We use MLPerf here because it is a popular set of benchmarks which can be used to demonstrate *coverage* of DL. We compare to similar platforms as Chapter 2 - specifically A100

and H100 from measured and extrapolated data when measurements are not available. Chapter 4 uses an analytical model validated against the same platforms as Chapter 3. Chapter 5 uses a set of applications which draws from both MLPerf and the CASIO suite. We based our selection primarily on the engineering effort required to implement spatial fusion for an application, as well as how well we felt that application would highlight the impact of spatial fusion. We develop a fast queue library that is evaluated in silicon and simulation and prototype several spatial fusions. This chapter also uses a proprietary NVIDIA simulator for the hardware platform matching Chapter 2. The software artifacts we develop for this work including public links are summarized in Table 1.2.

2 GPU LONGITUDINAL STUDY

We are in age of AI, with changing algorithms and hardware. Domains that AI has rapidly revolutionized include scientific simulation, computer vision, language processing, speech, and more. As AI applications have evolved, the underlying hardware has evolved from GPGPU to add forms of specialized hardware blocks targeted just for deep-learning (DL) – one notable example is NVIDIA's TensorCore to accelerate GEMM operations. In addition, entire chips designed just for DL include Google's TPU [68]. In between this, the software "stack" that application developers use has gotten more and more mature with modern AI applications needing barely 1,000 lines of code, running on frameworks like Tensorflow and PyTorch. *Amidst this three-way explosion across domain, hardware features, and software stack complexity, there is insufficient understanding on the multiway interactions and how modern hardware is performing in real use cases*[1].

MLPerf [148, 178] is a recent benchmark suite that serves as a way to compare and evaluate hardware. Choquette et al. [19], provide a detailed analysis of it's performance on the A100 GPU for example. However it has several drawbacks - it is excessively dominated by CNNs and does a poor job of capturing the diversity of AI use cases, and only represents a sliver of "production" AI use cases. We perform a longitudinal study of state-of-art AI applications across three generations of hardware to understand how the AI revolution has panned out, and how future AI hardware should evolve. The study's key goal is to answer three questions: i) How does application behavior scale across hardware generations? ii) What are the software-compiler-hardware interactions responsible for enabling generational speedup or limiting performance? and iii) What can we learn from the state-of-art for where future architectures must focus? The closest related work is Google's TPUv4 inference retrospective [68] - which lacked detailed data on application, framework stack, or microarchitecture-level hardware utilization trends.

Answering Q1. The approach we take is to identify 10 real application use cases. This is in itself a hard problem - to avoid the problem of over-reliance of CNNs, and the somewhat "staleness" of benchmarks, we select applications based on their research impact (large # of citations), diversity across domains, and availability of code. We then study these applications (described in Table 2.2) from the perspective of the framework, software API, and underlying hardware across three

[1] Most academic works are restricted to run a few layers of convolutions and matmuls usually.

Insight	Behav.	Comments
1. Heavy Tail	All	All behaviors are brought out by the wide range of kernels needed
2. Shape Specialization	Ex.	Shape spec. needed for high perf. of irregular GEMM shapes
3. GEMM diminishing	Data	Decrease in GEMM contrib. makes low-AMI kernels more important
4. HW under-utilization	All	Algo-Arch. codesign could unlock better orch. of all behaviors
5. Behavior variation	Dep.	High temporal variation lends to "behavioral parallelism"

Table 2.1: Mapping of insights to unifying execution behaviors. Definition of behaviors in Section 2.6

generations of NVIDIA hardware (P100, V100, and A100). Overall generational gains across these applications is shown in Figure 2.8 (Pg. 27). We see the V100 achieves substantial gains over the P100 with its addition of the TensorCore, for MLPerf and our applications. The A100, often achieves only modest speedups, while maintaining the speedups for MLPerf. The curated applications, infrastructure we developed to run them, and data we gathered are provided for download. We call this collection with our application execution scaffolding the CaSiO suite. `https://github.com/VerticalResearchGroup/casio`

Answering Q2. We identify 5 insights which characterize the software-compiler-hardware interactions in these applications:

Insight 1: There is a heavy-tail of operators and device kernels a DL architecture much support – the compiler, architecture, and microarchitecture bear the responsibility to create and execute performant code.

Insight 2: Not only are there a large number of DL operations to support, but each individual operator may need many device kernels specialized to some aspects of the operator's semantics and/or microarchitectural parameters in order to extract high performance.

Insight 3: The lower contribution of GEMM-based operations on newer GPU hardware, and the likely lower future generational improvements in GEMM performance suggest the "era of GEMM acceleration" is over.

Insight 4: There is substantial underutilization of all hardware resources, providing an avenue to target for future speedups.

Insight 5: Diversity and temporal variation of execution state is high among device kernels in AI applications, suggesting hardware features must be able to support rapidly changing hardware execution state.

Answering Q3. Driven by these insights, we identify three behaviors that could be targeted

Application Name	Batch	# Param.	Description	Key Breakthrough, Citations, Year
MGN [136]	1	⩽3.6M	Mesh based physics sim.	100X speedup, 220, 2020
MuZero [154, 30]	1-1024	7.3M	Learning to play games.	Acc. and generality, 1172, 2020
NERF [102]	1024	600K	View synthesis	Speed and new capability, 1759, 2021
PINN [145]	1	⩽141K	Physical sim w/ NNs for PDEs	Acc. & generality, 3908, 2020
QDTrack [131]	1-2	57M	Multiobject tracking	Acc. & generality, 101, 2021
Swin-T [90]	1-32	⩽197M	Vision transformer	Vast improvement in acc., 3790, 2021
TabNet [5]	4-128	2.5K	Interpretable DL for tabular data	Self-supervised learning acc., 333, 2021
Tacotron2 [159]	1-64	28M	Speech recognition	State of art, 1972, 2018
WaveNet [127]	1	1.5M	Speech synthesis	State of art, 4790, 2016
GPT-3 [10, 71]	1	175B	Large Language Model	State of art, 13418, 2020

Table 2.2: Applications in this study. Applications were primarily chosen based on diversity and citation count (as a proxy for popularity).

for the next generation of speedups: 1) **compute orchestration** – how low arithmetic-intensity compute kernels such as irregular or small GEMM shapes is executed by the execution units, 2) **data orchestration** – how data-movement operations are handled by the memory system and 3) **dependence orchestration** – how diverse execution behavior and algorithmic dependence edges are leveraged by the accelerator to keep chip resources occupied. Table 2.1 maps our insights to these behaviors; Section 2.6 presents an empirical performance model and a case study which shows a 2.3X geo mean speedup can be achieved by targeting these behaviors (Figure 2.11).

The rest of this chapter is organized as follows. Section 2.1 provides DL background and related work. Section 2.2 describes methodology. Sections 2.3, 2.4 and 2.5 present our data analysis and insights. Section 2.5 and 2.5 revisit GPT-3 and MLPerf, Section 2.6 presents our behavior analysis and path forward, and Section 2.7 concludes.

2.1 Background of DL & Related Work

DL Stack for GPUs. Appendix A characterizes DL applications and details their execution characteristics. Deep Learning on GPUs requires many interacting components of a development stack as shown on the left-hand side of Figure 2.1. Frameworks like Tensorflow and PyTorch are DSLs that are designed for ease of programming, with applications written in 10s to a few hundred lines-of-

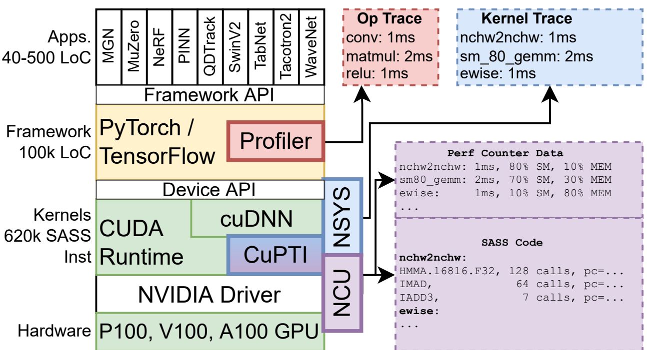


Figure 2.1: Overview of the DL stack and tooling we used for profiling and extracting performance data. Thick border indicates instrumentation tooling. Dashed border indicated data that we gather.

code (usually python)[2]. PyTorch and TensorFlow both employ the use of cuDNN, an optimized library of primitive functions which can be used to implement a common subset of DL operators. This *device API* allows GPU code to be abstracted away from the framework; allowing the framework to be accelerator agnostic. cuDNN contains CUDA kernels stored as SASS binaries compiled from a myriad of techniques – some hand coded, some generated from template frameworks such as cutlass [73]. In general, there are many academic works aimed at auto-compilation of kernels for Deep Learning such as MLIR and TVM [84]. One recent paper overviews the full complexity of building a "Deep Learning Compiler" [88]. Embedded inside the framework and device API is a massive software stack which is responsible for lowering high-level Python into performant object-code, including optimizations for different tensor-shapes and high-performance code for different operators. This underlying code-base is measured in 100,000s in terms lines-of-code. Backend developers, compiler writers, kernel writers, and hardware architects essentially view the framework API as the abstraction to target.

GPU Studies. Svedin et al. [166] provide a longitudinal study of 5 generations of GPUs on non-DL workloads, finding A100 provides less performance increase compared to previous GPU generations. Anzt et al. [4] study sparse matrix-vector multiplication with a focus on linear-system solvers on A100 and V100 GPUs and find the A100's higher memory bandwidth and larger cache allows for 1.45-1.8X performance over V100 on batched sparse matrix-vector product. Both of these studies focus on non-DL workloads and close-to-CUDA software stack. Many works analyze the TensorCore in depth from hardware perspective without considering whole applications effects, while covering

[2]MLPerf applications: torchvision's ResNet50 implementation is about 300 lines; the core BERT model file is 1100 lines.

numerical precision support [32], programmability [94], memory system bottlenecks [95, 190], it's core microarchitecture & simulation modeling [144], and microbenchmarking to expose micro-architectural features of the V100 and A100 [62, 63, 98, 185, 165]. Li et al. study sensitivity to interconnect and scale-out [86].

Other Studies. Sevilla et al [156] present a broad historical trend of DL growth considering model size and FLOPs trends across 5 decades. Hestness et al. present an empirical and analytical model on growth in the trends of DL model sizes and accuracy [51], with some studies focussed on sparsity [52, 25]. Jouppi describes the lessons learned on 4 generations of TPU hardware [68]. A hardware centric survey of machine learning hardware is presented in [103, 151], an FPGA survey is here [81], and a platform-application survey is here [50]. Fuchs et al. present a limit study of chip specialization separating out contributions of technology scaling and architectural gains [38].

2.2 Methodology

In this section, we first cover our list and rationale for applications and hardware targets chosen. We then describe the measurement tools, the execution environment of each of the applications including datasets and frameworks used. In all cases, our goal was to adhere to production uses cases or the intended environment from the authors of the applications.

Application Selection

Applications. We used three metrics to guide our application selection: recency of publication in top-ranked AI publications, large number of citations, diversity of use case. A practical requirement was that the code needed to be runnable on our target hardware (NVIDIA family of GPUs) - as we explain in the next sub-section, this study's focus is three generations of **practical**, **usable** and **available** commercial hardware. Another guiding factor, was an informal one that the applications must reflect diversity in application domains. Table 2.2 summarizes the applications, their number of citations, and which domain they operate in, and potential or current commercial use cases. Their variants are used in several commercial settings including Google's Speech and Speech synthesis API, Tabular API, video tracking and vision transformers are used in several AV settings, the elements of PINN and Meshgraphnets are used in commercial physics simulations like Omniverse.

Some of these application indeed have sub-variants, which is driving the growth in DNN literature - KiloNERF [150], and InstantNERF [104] in photogrammetry for example. LLMs including OpenAI's GPT-3, LLaMA 2 and Phi-2 have recently exploded in popularity [128, 171, 101]. The majority of LLMs are structurally similar to the attention-based transformer architecture of BERT with massively scaled model sizes. GPT-3 specifically uses 96 transformer layers which include 96 attention heads and a hidden dimension of 12,288, giving a total of 175B (≈300 GB of FP16 weights) parameters compared to BERT-Large's 336M parameters (520X larger). While there is no officially released source-code for GPT-3, we use publicly available software [71] which can be used to recreate GPT-3 by using its model hyperparameters.

Training. In this study we narrow our focus to training, since it is a more contained (and arguably more fundamental) problem. Inference introduces additional figures of merit that include tradeoffs between latency vs throughput, power vs energy, physical form factor constraints, and software/algorithmic optimizations that allow for quantization/re-training. Understanding the training space can serve as useful pre-cursor to understanding inference. We also did some optimizations and assumption to eliminate "straight-forward" software/algorithm inefficiencies: even if the applications were not yet fully made stable for fp16 numerical computation, we forced all applications to run in fp16 (by modifying source code) for all ops except ones which would cause gradient over/underflow to get a sense for the limits of the hardware capability in terms of using hardware features like the TensorCore. Others have explored the algorithmic issues of mixed precision in depth [100]. Due to the fact GPT-3 cannot fit in a single GPU's device memory, we study the performance and behavior of a single transformer layer of GPT-3 (≈3.4 GB of FP16 weights). Running a single transformer layer is representative of a distributed training scenario using pipelined model-parallelism [57, 146]. Since all of GPT-3's layers have identical hyperparameters, this captures all of the operators and shapes in GPT-3. Additional techniques such as banded-sparse attention is used in production GPT-3 at OpenAI.

Region-of-Interest. The unit of work itself needs careful attention, since DNNs have at least three distinct phases from the perspective of *single* GPU (excluding initialization with data-loaders, system-level schedulers, and inter-node communication): forward pass, loss calculation, backward pass. For this study, a unit of work or region-of-interest (ROI) is defined as one forward pass, loss

GPU Platform	TFLOPS FP16 / 32	Mem (GB/s)	CPU
P100	21 / 10.6	720	Xeon 4114 x2, 192GB
V100	125 / 15.7	900	Xeon E5-2667 x2, 128GB
A100	312 / 19.5	1,555	EPYC 7282 x2, 512GB

GPU	# SMs	Major Changes over Previous Generation
P100	56	HBM2 Added
V100	80	TensorCore Added
A100	108	7X L2, better L2 cache control, async-copy

Table 2.3: System configurations

computation[3], and one backward passes. Furthermore, for each application, we ran the applications with as large a batch size as tolerated until the device ran out of memory. As mentioned by [157] there is abundant data/batch-level parallelism, which makes more efficient use of hardware resource for GPUs. Others have discussed how small-batch is desirable for other hardware platforms [163, 96].

Hardware Selection

There are many hardware platforms to consider, with many startups' chips, Google TPU, AMDs GPUs, and NVIDIAs GPUs. To practically perform a longitudinal study of multiple generations of hardware rules out startups. Second, a requirement to run real applications from our list rules out startup chips and AMD GPUs (none support the level of operator diversity needed; no other platform has full MLPerf submissions [37]). That left us with Google TPUs and NVIDIA GPUs. We decided to focus on generational study across **one** hardware platform. The public availability of the hardware, easily accessible low-level tools like profilers (from NSIGHT Compute suite), and publicly available information on the API level and hardware level ISA made NVIDIA GPUs the only viable choice.

We selected the A100, V100, and P100 as our targets going back 3 generations, spanning time-scale of 6 years, 3 technology nodes (16nm 12nm, and 7nm), chip sizes spanning 600 mm^2 to 828 mm^2 [77]. Details are in Table 2.3, including key generational advancements. Further details can be found in [125] and the respective whitepapers [115, 123, 121].

[3]In some cases, we simplify the loss function since it doesn't meaningfully impact runtime or computational complexity.

Measurement tools

Measurement tools targeted at the framework, device/API, and hardware layer are used to measure performance, power, and cost. This study primarily focuses on performance as power and cost are somewhat secondary due to stable TDP and die area; other studies have shown memory vs compute power is dictated by scaling [129]. The measurement tools and the study focus on execution time, which dictates power and energy consumption. The analysis flow and tool interactions are shown on the right side of Figure 2.1

All the applications are written in TensorFlow or Pytorch[4]. Both of them have a built-in profiler which uses low-level hardware performance counters to measure operator runtime [140, 170]. Additional information like shapes and number of tensors is also tracked. NVIDIA offers two NSIGHT tools, NSIGHT Systems (NSYS) and NSIGHT Compute (NCU). NSYS observes CUDA kernels using CuPTI and provides start-time, duration, and launch statistics. NCU is a low-level profiler that reads hardware performance counters and provides detailed kernel summaries, including SASS code and dynamic invocation metrics. The CUDA ABI's memory snap-shotting is used to replay kernels and gather many metrics. For some applications, we run with `1/10th` sampling to limit device-host copying slowdown. The full list of NCU events is here [117]. We focused on the subset outlined below: `% sm throughput`, `% of l1tex throughput`, `% dram throughout`, `% issue-slots used`, `% execution-slots used`, `% FMA used`, `% FP16 used`, `% FP64 used`, `% tensor-core used`. Note that we have simplified the naming of these counters to be more intuitive with microarchitecture usage - the actual names of the counters/stats are different. Each of these is expressed as a percentage from 0 to 100, reflecting what average percentage of that hardware resource across that entire kernel's execution was used.

Execution Infrastructure

All of the applications run under a substantially managed infrastructure including a docker [118, 119] or python virtual environment when needed for different packages, datasets and source code. All applications, we run on the same CUDA version on the host machine. For two applications, the data set changes the nature of the underling DL architecture: meshgraphnets and PINN. For

[4]For the purpose of this study version difference with TF1.15, TF2.0 and Torch version are irrelevant and overcome with using containers

Op Name	P100	V100	A100
MGN-CFD (37 / 37 uniq.) (ROI: 4.5s) (1.2 kDyn)			
matmul	44.07%	17.32%	43.75%
Mean	1.38%	2.15%	22.68%
ResourceApplyAdam	6.11%	17.57%	6.65%
mul	10.40%	12.92%	3.53%
UnsortedSegmentSum	7.49%	4.85%	2.17%
Other	30.55%	45.19%	21.20%
MuZero (30 / 60 uniq.) (ROI: 24.4s) (173.1 kDyn)			
conv-bwd	54.34%	43.98%	36.49%
conv	30.09%	22.90%	18.96%
batch_norm	5.48%	14.48%	17.68%
Other	10.09%	18.65%	26.87%
PINN-NS (22 / 69 uniq.) (ROI: 12.2s) (0.7 kDyn)			
dynamic_stitch	64.26%	53.93%	54.62%
mul	8.27%	17.20%	22.85%
matmul	17.85%	10.45%	8.32%
Other	9.62%	18.42%	14.20%
QDTrack (83 / 127 uniq.) (ROI: 4.0s) (121.8 kDyn)			
conv-bwd	49.40%	38.55%	31.62%
conv	33.24%	24.81%	22.67%
add	4.16%	9.10%	8.15%
Other	13.20%	27.54%	37.55%
SwinV2-B-P (44 / 139 uniq.) (ROI: 2.6s) (153.9 kDyn)			
matmul	67.44%	36.52%	27.52%
add	4.98%	11.38%	12.64%
mul	5.42%	10.01%	11.08%
Other	22.16%	42.08%	48.75%
TabNet (42 / 157 uniq.) (ROI: 34.6s) (16.9 kDyn)			
Slice	15.45%	14.60%	17.46%
GatherV2	16.23%	16.66%	16.60%
cub	14.50%	14.69%	13.77%
Other	53.83%	54.05%	52.17%
Tacotron2 (53 / 177 uniq.) (ROI: 12.6s) (679.2 kDyn)			
matmul	22.52%	31.44%	36.16%
add	10.13%	24.29%	25.16%
conv-bwd	55.23%	16.50%	8.51%
Other	12.11%	27.77%	30.17%
WaveNet (27 / 183 uniq.) (ROI: 21.0s) (0.3 kDyn)			
conv-bwd	77.03%	65.01%	69.25%
conv	20.31%	29.84%	20.51%
transpose	0.29%	0.36%	4.37%
sum	0.56%	2.00%	2.89%
Other	1.81%	2.80%	2.99%
GPT3 (21 / 185 uniq.) (ROI: 4.1s) (2.1 kDyn)			
matmul	96.17%	78.31%	64.61%
_foreach_add	1.31%	7.64%	12.78%
_foreach_mul	0.46%	2.66%	4.55%
mul	0.49%	2.86%	4.47%
Other	1.57%	8.53%	13.58%

Table 2.4: Top Operator Contributions by Runtime

App Name	P100			V100			A100		
	kDyn	**Uniq**	**CU**	**kDyn**	**Uniq**	**CU**	**kDyn**	**Uniq**	**CU**
MGN-CFD	123	97	97	123	98	98	123	103	103
MGN-Cloth	120	95	99	120	97	103	120	100	111
MuZero	305	77	170	326	70	168	314	59	165
PINN-NS	144	44	186	151	45	187	144	45	183
PINN-Schr.	29	48	188	30	49	192	31	56	193
QDTrack	154	240	364	186	272	405	177	251	395
SwinV2-B-P	169	77	390	169	89	439	169	103	439
SwinV2-L-F	168	76	392	170	84	445	171	102	454
TabNet	47	70	432	47	71	485	47	72	494
Tacotron2	854	127	523	997	127	563	923	140	574
WaveNet	182	118	589	134	217	722	117	210	728
GPT3	10	32	603	10	33	736	10	36	744

Table 2.5: Number of Total and Unique CUDA Kernels

these we report multiple entries in all our results, suffixed with the dataset name. In addition, some applications have different configurations - e.g. Swin-Transformer; where we find just two configurations cover the execution characteristics. MuZero is an exception as it includes many architectures depending on the game being learned. We report results for Atari which is neither too simple nor too complex. Also, we note here that MuZero uses reinforcement learning, and includes a very massive system infrastructure to manage worker threads, the game engine etc - our focus is on just the DNN.

We then ran every application first across a sweep of doubling batch size to determine the largest batch-size the application would run on. We then ran every application under each of the profiling environments which created a total of nearly 210GB of data, which we then post-processed using various other analysis scripts.

Various compiler optimzations like XLA [152] and framework optimizations are in production [139] and being researched [60, 181, 69] with various levels of application coverage/robustness. We acknowledge that the results we gather could change as the compiler / framework stacks mature, and in fact, the data we observe serves an indication whether or not future compiler optimization could close the “transparent speedup” gap we observe. With this in mind, we took care to run all apps with the latest supported version of all libraries and frameworks they support which include the latest “upstream” optimization options. Considering XLA, in 4 of our applications on V100, XLA provided no speedup, on one it improved execution by 15%. For A100, 1 had no change, 2 had a 10% slowdown, two had 20% speedup. Other applications we couldn’t run with XLA.

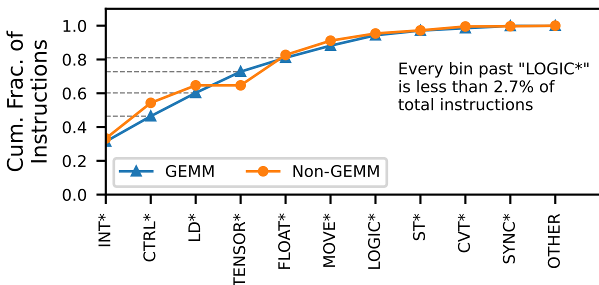


Figure 2.2: Cumulative instruction count contribution for SASS opcodes for GEMM and Non-GEMM kernels.

2.3 Software Perspective

Framework Operators. For each application, we identify the top 3 framework operators by runtime contribution on each platform (P100, V100, A100) and show the union of these ops (as well as "other") ordered by descending contribution on A100. This data is summarized by Table 2.4. Each application header shows the total and cumulative total number of unique ops (cumulative across the set of applications). Each application also shows the latency of its region of interest (ROI) and number of dynamic operators executed (measured in thousands). We omit applications which do not contribute additional unique operators.

Device Kernels across GPU Generation. At the kernel level, applications are seen as a series of device kernels (CUDA kernels for NVIDIA GPUs) which carry out the semantics of the framework level operations. We use NSYS profiler output to obtain a kernel trace for each application. From this, we count the number of dynamic kernel invocations in the trace and compute the number of unique kernels by name and the *cumulative* unique kernels by name across applications. This data is summarized in Table 2.5.

Kernel Mixture. We examine the fractional contribution of top device kernels to each application, summarized by Table 2.6. For each application, we show the percentage of runtime spent in GEMM kernels, the number of unique GEMM kernels, and the number of additional unique non-GEMM kernels needed to cover 80% and 90% of application runtime. We choose to only show this data for A100 because the other two platforms are quite similar. To get the classical understanding of ISA role, we also binned the dynamic SASS instructions (315 unique) into 5 categories averaged across applications for GEMM and non-GEMM kernels as shown in Figure 2.2. The large number

Application	GEMM % Runtime	# GEMM Kerns	+ # Kerns For 80%	90%
MGN-CFD	12%	19	25	5
MuZero	54%	21	3	1
NeRF	58%	17	2	6
PINN-NS	40%	6	4	1
PINN-Schr.	32%	15	9	9
QDTrack	41%	63	11	19
SwinV2-B-P	25%	46	12	5
SwinV2-L-F	24%	47	10	4
TabNet	2%	3	26	4
Tacotron2	46%	61	5	6
WaveNet	90%	155	0	1
GPT3	63%	9	2	4
Resnet50	31%	31	4	2
BERT	48%	10	7	2

Table 2.6: Kernel level breakdown of apps showing percentage of runtime spent in GEMM kernels, number of unique GEMM kernels, and the number of additional unique kernels to reach 80% and 90% runtime coverage.

of opcodes needed to handle the data-access portion of kernels, as well as non-GEMM based kernels with diverse control behavior, mirrors the findings from past studies of instruction mix of short-vector architectures [167].

Observations

We see a few trends in the kinds of operations being performed. *matmul* and *conv* and their *-bwd* counterparts are arithmetically intense operations commonly seen in DL workloads' "learnable" layers. Ops like *add*, *mul*, *div* and *batch_norm*[5] are element-wise operations with relatively low arithmetic intensity. Operations such as *dynamic_stitch*, *Slice*, *transpose*, *Concat* are purely data movement operations which have no compute associated with them. We find most applications see 80% of time spent in 2-10 ops, with convolution and matmul as the most common top ops (with $>$ 30% runtime in most cases). Beyond this, we see 185 total unique operators, with each application having between 21-83 unique operators.

We also observe the number of dynamic operators as well as kernels is massive; between 300-679,240 dynamic operators, and 10,000-922,800 dynamic kernels are executed across the applications. In general, there are 2-4 device kernels for each framework operation: for the 185 unique framework

[5]While batch-norm requires computing statistics across the batch dimension, we still classify this as "element-wise" since it preserves shape and output elements depend only on the statistics and corresponding input value.

operators, we see 744 unique device kernels on the A100 (3.3 kernels/op on average). For GEMM-based kernels, we observe 123/259/308 unique kernels across P100/V100/A100 which is due to addition of shape-specialized routines that are selected by cuDNN based on input tensor shapes to an operator. Several works target optimization of loop ordering and tiling for dense DL kernels [132, 15] targeting data movement. We observe that CuDNN incorporates these ideas by implementing a plethora of shape-specialized GEMM kernels optimized for data-movement and higher level heuristics that select a particular kernel based on tensor shapes. NVIDIA CUTLASS [72] – used as a component of CuDNN – generates 308 specialized kernels for A100.

While GEMM-based kernels are the top runtime contributor (2% to 58%, excluding the outlier of Wavenet) to most applications, we find many more kernels are needed: to cover 90% of the runtime, of all applications, 308 GEMM and 137 non-GEMM kernels are needed. GPT-3 is an exception to this, only needing 15 total unique kernels.

It is clear there is a massive software stack involved in supporting DL applications [88] as observed by analysts like Linley [147] as well. To meet this challenge, NVIDIA has spent approximately $3B on developing its CUDA ecosystem [21], including the toolchain, libraries, and integration with DL frameworks. In an effort to bridge this gap more quickly, recent academic works have aimed to simplify and automate lowering of DL operations to accelerator architectures [84, 175]. Even with these techniques available, companies like AMD still lag behind NVIDIA in terms of software ecosystem maturity and ease of integration [36].

Recently, in-memory and near-memory architectures have been proposed and revisited in the context of deep-learning, to speed up convolution [155], while others are targeted at element-wise / embedding operations with low arithmetic intensity [79]. A comprehensive survey is here [8]. None of these works present a quantiative characterization of time spent in and outside GEMMs on state-of-art GPU hardware. They also do not discuss how these ideas would be integrated into a modern DL software stack, nor do they present a systematic analysis on the properties of DL operators, or why GEMM acceleration alone would be insufficient. Orthogonally, Ivanon et. al describe at an algorithmic level, how data movement is important in transformers / attention-based networks [60]. Finally, sparsity [52] – tackled at the algorithmic, compiler, and hardware level – is orthogonal to the issues we cover.

Insights

Conventionally, compute-intensive GEMM-based operations (matmul and convolution) are thought to make up the bulk of DL applications and have been the target of both industry and academic proposals. We find that while these are often in the top 3 ops for most applications, they don't make up the majority of runtime. If we speedup GEMM-based operations on P100 by a factor of 10X (roughly the amount of compute added by V100's TensorCore), we would see a rise in end-to-end application performance of 2.6X (as will we see shortly, the actual V100 to P100 speedup is geo-mean 2.4X). Looking forward on the other hand, if we speedup GEMM-based operations by 2X on the A100 (roughly the ratio of A100:V100 peak FP16 TFLOPS)[6], end-to-end application performance *would only increase 30%*.

⋆ *Insight 1: Any hardware platform must support a large number of DL operators in order to avoid performance penalties from frequent off-accelerator data movement. By extension, we find a large number of kernels are required for 90% runtime coverage for each and across applications – meaning there is a heavy tail of device kernels to support. The responsibility of kernel developer/compiler/architecture/microarchitecture to create performant and correct code is very large, with supporting high-performance convolution or matrix multiplication representing only a sliver of execution time. Since hardware is changing as is software, this responsibility will continue to be important.*

⋆ *Insight 2: The explosion of shaped-specialized GEMM kernels is exemplary of how specialized device code needs to be to extract high performance on modern accelerators. This aggravates the software problem – not only are there a large number of DL operations to support, but each individual operator may need many device kernels specialized to some aspects of the operator's semantics and/or* ***microarchitectural*** *parameters in order to extract high performance. Looking forward, this software problem will continue to be important.*

⋆ *Insight 3: From a historical perspective of looking at the P100, the large performance opportunity presented by GEMM-based operations makes adding specialized hardware with shape-specialized kernels like the TensorCore a clear silver bullet. However, the lower contribution of GEMM-based operations on newer GPU hardware, and the likely lower future generational improvements in GEMM performance* (*see also Sec 2.5*) *suggest the "era of GEMM acceleration" is over.*

[6] Since the A100 already contains a specialized unit for GEMM acceleration, we anticipate 10X growth in GEMM's compute to no longer be possible.

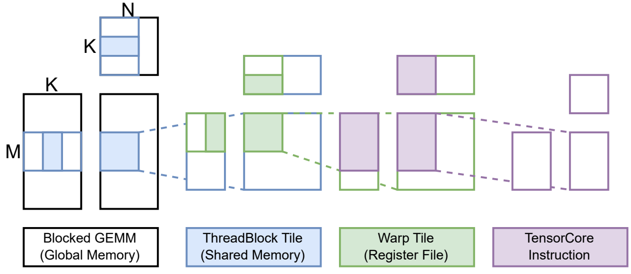


Figure 2.3: Depiction of GEMM operation with NVIDIA CUTLASS (Adapted from [72]). This shows the tiling strategy for threadblocks (CTAs), Warps, and TensorCore instructions ("atoms").

2.4 GEMM Deep Dive

In this section we examine the GEMM shapes from our selected applications focussing on generational performance trends across GPUs and how shape influences performance.

A matrix-multiply (GEMM) shape is identified by the dimensions of the operand matrices, MxK and KxN, with the output being MxN. Convolution can be computed via GEMMs where N and K are the number of output and input channels, respectively, and M is a combination of the spatial dimensions. GEMM execution on a GPU is described in Figure 2.3 (Adapted from [72]) which is based on NVIDIA CUTLASS – the state of art software for generating high-performance GEMM kernels. Tiles of output are computed in parallel in separate threadblocks assigned to the many SMs of the GPU (i.e. M and N are a source of parallelism). Hiding long L2 and HBM latencies is paramount to high performance, and is achieved by tiling the K dimension and overlapping memory prefetch with compute. Tiles of inputs along the K dimension are staged between global memory, shared memory, and registers with asynchronous memory movement instructions that are overlapped with TensorCore instructions (or SIMT instructions in P100, which becomes a limiter as we will observe in Sec 2.5). Warps in a threadblock cooperate with memory movement, allowing input tiles to be reused within a threadblock.

Recall the ratios of peak FP16 compute for V100 / P100 and A100 / V100 are 6X and 2.5X, respectively. Figure 2.4 shows a histogram of GEMM shapes' compute utilization for each GPU generation. Figure 2.5 shows a scatter plot of a sweep of GEMM shapes (M=N, for powers of 2 from 64 to 32768, and K from 64 to 32768). We color shapes by whether they have large dimensions, large M and N but small K, large K but small M and N, and all small dimensions. The shapes

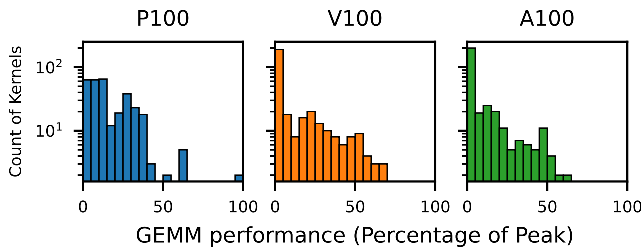


Figure 2.4: Histogram of GEMM Compute utilization on each GPU.

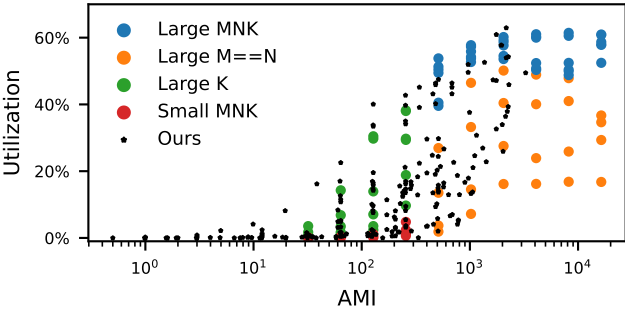


Figure 2.5: Scatter plot of GEMM shapes showing Utilization on A100 vs AMI. We show a sweep of shapes categorized by size of dimensions in addition to our observed shapes.

			% of Peak Compute			Speedup	
	Shape	**AMI**	**P100**	**V100**	**A100**	**V/P**	**A/V**
1	40x1024x12544	38	11%	9%	16%	4.8	4.5
2	64x128x1024	43	3%	0.5%	0.3%	1.0	1.7
3	31072x28x512	128	38%	38%	33%	5.9	2.2
4	1024x256x512	205	26%	10%	6%	2.3	1.3
5	32768x256x512	254	37%	45%	40%	7.3	2.2
6	8192x512x64	482	23%	18%	9%	4.5	1.3
7	8192x512x512	482	32%	44%	40%	8.0	2.3
8	2048x4096x1024	1365	37%	54%	53%	8.6	2.4
	Geo-Mean for all 317 shapes					2.0	1.6

Table 2.7: Selected GEMMs sorted by AMI.

seen in our applications is depicted as well. Table 2.7 shows a selection of interesting GEMM shapes which we discuss in later subsections, including M, N and K values, arithmetic intensity, compute utilization on P100, V100 and A100, and V100/P100 and A100/V100 speedup. To measure GEMM performance, we use PyTorch to benchmark the latency and compute throughput of each shape. Our full list of 317 GEMM shapes and their performance can be found at the following URL: `https://github.com/VerticalResearchGroup/casio-gemms/blob/main/gemms.csv`. In Figures 2.4 and 2.5 and Table 2.7 utilization is percentage of peak FP16 flops used.

High Level Observations and Insights

Lots of shape diversity, and performance is improving. We observe a large number of unique GEMM shapes (317 – see full list) across the applications with AMI ranging from 0.5-3276 FLOPS/Byte. AMI indirectly quantifies reuse and is important to hide latency and curtail bandwidth needs. At AMI≈60, for the A100, we see compute utilization of around 1.5%. At AMI>600 and well shaped GEMMs (sufficient parallelism and large K dimension to keep TensorCore well utilized), we see around 10% and higher utilization. Later in this section, we discuss why AMI isn't perfectly correlated with utilization. Recall a total of 123 / 259 / 308 unique kernels for P100 / V100 / A100 are used for these applications, respectively. Considering performance across generations (Table 2.7), we find V100 and A100 are consistently better than their predecessors – GM 2X and 1.6X, respectively, compared to 6X and 2.5X peak compute increase. For V100, we find 114 out of 317 shapes achieve >3X speedup over P100, whereas only 4 shapes achieve >3X on A100.

Utilization is decreasing. While performance is increasing, we find utilization is dropping. From Figure 2.4, we find across GPU generations, GEMMs are achieving lower utilization – on A100, over 100 shapes are operating at <25% of peak performance. 217 of the GEMM shapes see a generational decrease in utilization for BOTH P100→V100 and V100→A100, with utilization dropping as much as 22% from V100 to A100. We observe the largest drops occur at higher AMI. One exception is GPT3's shapes (discussed in detail in Sec 2.5) which increase in utilization from P100→V100 then decrease from V100→A100.

AMI does not fully explain performance. Arithmetic intensity describes the amount of reuse available in a GEMM shape. We find across the shapes we study, AMI only explains some of the variance in GEMM performance (measured FLOPs/sec vs. AMI $R^2 \approx 0.59 - 0.61$ across the 3 GPUs), with this correlation decreasing at higher AMI ($R^2 \approx 0.37 - 0.40$) past 500 FLOPs/Byte. We see this correlation fail in shapes such as Table 2.7 (row 1) which achieves 49X higher utilization than the next higher AMI shape (row 2) – and additionally increases in utilization on A100 since the large K dimension makes it amenable to leveraging the A100's faster tensor cores. We see this AMI correlation in Figure 2.5 as well. We now examine these categories of GEMMs in detail, highlighting the diversity of behaviors that impact performance with an eye towards explaining why AMI only partially explains performance.

Large M,N,K provide consistent performance

We find large GEMM shapes (specifically, when M,N,K > 512, with M especially >> 512 – a total of 36 shapes) achieve consistently good performance across all three architectures. Some examples of these are Table 2.7 (rows 5, 7, and 8) and the blue circles in Figure 2.5. Two of the key barriers to high performance are 1) being able to hide the long memory latency, and 2) having enough parallel work to occupy the many SMs. This creates an interdependence between GEMM shape (AMI, reuse, available parallelism), shared memory capacity, and memory bandwidth. For these large shapes, there is abundant parallelism and reuse available, lowering pressure on shared memory, registers and bandwidth for hiding latency and occupying the GPU.

At least one of M,N,K is very small

For GEMM shapes with degenerate dimension sizes, it is much harder for memory latency to be hidden, or there is insufficient parallelism to keep the GPU's SMs occupied. We explore these two performance pitfalls by examining representative GEMM shapes. Recall the M,N dimensions are used as a source of parallelism, and K is reduced over in the inner-loop.

K dimension. Table 2.7's rows 6 and 7 shows two GEMMs with similar AMI but differing in their K dimension by a factor of 8X. Both these shapes achieve good speedup from V100 but row 6 only achieves 1.3X on A100 where row 7 achieves 2.3X. Multiple iterations of the inner reduction loop are needed to overlap memory staging with compute. With very small K, the inner loop doesn't run enough, causing under-utilization of faster compute resources. The orange circles in Figure 2.5 show shapes with small K, but large M and N.

M and N dimension. Table 2.7 (rows 4 and 5) are two GEMMs with identical N,K values and relatively similar AMI (205 and 254, respectively) achieve very different speedups on A100 (1.3X and 2.2X). Between the two, there is a 32X factor difference in the M dimension, meaning row 5 has 32X more available parallelism which is how it is able to scale across GPU generations better. Row 3 also shows how when just one of M and N is very large, high utilization can still be achieved when K is large. Row 4's behavior is seen in the green circles in Figure 2.5 which shows shapes with large K, but parallelism limited because M and N being small.

2.5 Application Informed Hardware Analysis

In this section, we analyze how these applications execute on the hardware including an analysis of chip resource utilization such as memory bandwidth and compute throughput. We develop a taxonomy of induced hardware execution states. Finally, we combine these to examine the generational speedups across our application suite on the three GPU platforms.

Device Resource Utilization. To breakdown how usage of underlying hardware resources is related to end-to-end speedup, we examine the utilization of the GPU's SMs, DRAM bandwidth, and TensorCore: the three most important of our 8 HW counters. Note that SM utilization is defined as the max over several metrics, including FMA pipe usage, FP16 pipe usage and execution slot usage. Figure 2.6 shows over the application runtime, the utilization of the GPU's SMs as a percentage of peak ordered by increasing SM utilization (Blue solid lines). Each point along the horizontal axis represents one kernel and the distance between points is fraction of dynamic kernel time. We plot for each kernel, the instantaneous SM utilization, *cumulative* average DRAM bandwidth and tensor pipe utilization up to that kernel's invocation. This means any point on the x-axis allows us to bound the SM utilization for some fraction of runtime and show the average memory and tensor utilization for that fraction as well. For a point X on the x-axis, if the SM utilization is Y% and the corresponding intercept for DRAM and TC is D% and T% it means for X fraction of time across kernel execution, sustained average for DRAM and Tensor Util is D% and T%, respectively. For reference, the right hand column shows a chronologically ordered subset of runtime for three applications and shows *instantaneous* utilization of each resource. GPT-3's behavior is different from the rest so it is presented first. In addition, Figure 2.7 shows the *maximum* resource usage per application. The right hand side averages this "max utilization" across all applications ordered by increasing utilization. The raw data is released for readers to visualize and analyze differently. We found this visualization to be most instructive.

Hardware State Taxonomy. To offer a coarser-grain view of hardware execution states which also lends to better explain-ability of execution characteristics, we develop a taxonomy for classifying kernels. From the hardware standpoint, we choose the observed SM utilization (I.e. *compute efficiency*), and the observed DRAM bandwidth utilization (I.e. *memory efficiency*) since they are the most representative features. To capture parallelism afforded by a kernel, we include the

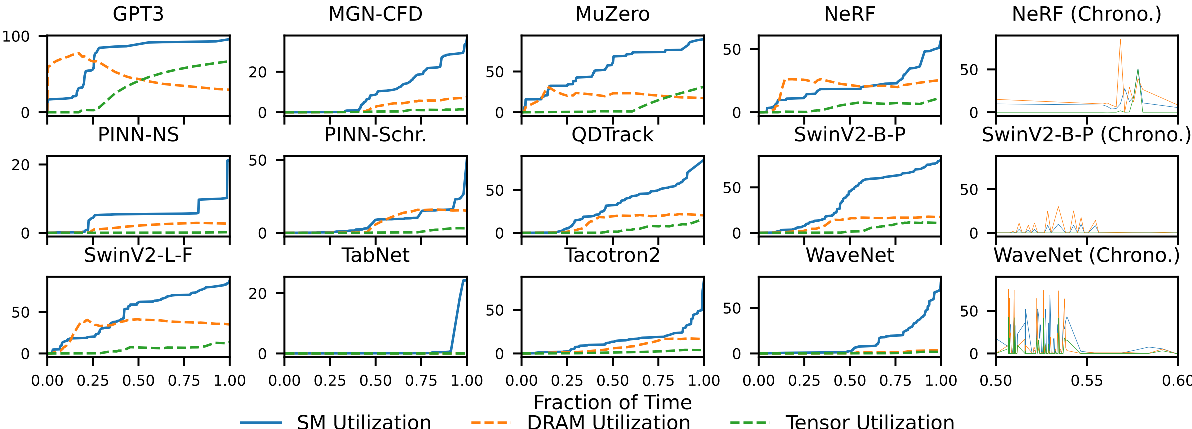


Figure 2.6: Hardware resource usage over fraction of runtime.

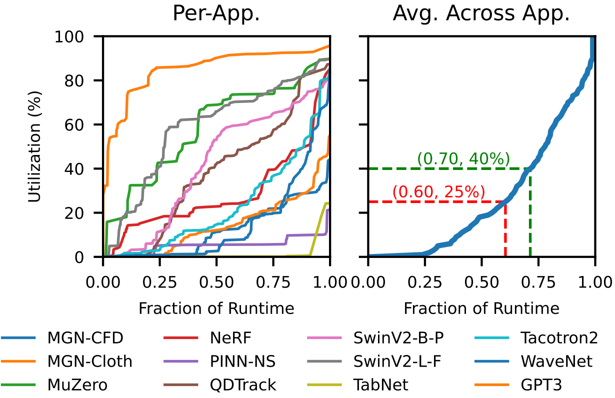


Figure 2.7: Per-application and average maximum resource usage by fraction of application runtime. Results ordered by increasing utilization. Red and Green points indicate bounds for average application utilization for 60% and 70% of runtime, respectively.

number of launch threads in our taxonomy as well. We choose cutoffs for "high", "medium", and "low" values for each of these metrics, summarized in Table 2.8. We call this the *state taxonomy*, which is a 3 dimensional space with a total of 27 state "bins". We observe that kernels with very low parallelism never achieve medium or high compute or memory efficiency, so we combine all bins with "low" parallelism together ($\mathrm{L}**$). Figure 2.9(a) shows a color coded timeline for each application's normalized run time, where each color represents one state and horizontal width of a bar is fraction of time spent in that state (state-color mapping is the same across applications). Black underscores indicate regions where a GEMM-based kernel is executing. Numbers next to the application name show the average *run length* of a particular state by number of kernels for

Metric	Low	Med	High
# Threads	<4000	4000-32000	>32000
SM %	<10%	10%-70%	>70%
MEM %	<10%	10%-70%	>70%

Bin Name = # Threads|SM%|MEM%

App	Bin	Kernel	FW Op
MuZero	HLL	gemm1[a]	AddMM
QDTrack	HLL	nchwToNhwcKernel	Conv.
WaveNet	HLL	ColumnReduceKernel	Reduce
MGN-Cloth	HLL	GatherOpKernel	Gather
SwinV2-L-F	HLL	vect_ewise_kernel[b]	Elemwise
SwinV2-L-F	MHL	gemm2[c]	AddMM
SwinV2-L-F	MHL	gemm3[d]	MatMul
SwinV2-L-F	MHL	gemm3[d]	MatMul
WaveNet	HLH	relu_grad	ReLU Bwd
Tacotron2	HHL	nchwToNhwcKernel	Conv.
SwinV2-L-F	HHL	reduce_kernel	Reduce

[a] cutlass_75_tensorop_f16_s1688gemm_f16_64x64_tn_align1
[b] vectorized_elementwise_kernel
[c] ampere_fp16_s16816gemm_fp16_128x128_ldg8_relu_f2f_...
[d] cutlass_80_tensorop_f16_s16816gemm_relu_f16_128x128_...

Table 2.8: Selection of interesting kernels classified by hardware state and their associated framework operator.

each application. Figure 2.9(b) shows this same information grouped by state bin and sorted by descending contribution to runtime to highlight time spent in each bin.

Generational Speedup. Finally, we study the end-to-end performance of our applications across GPU generations. For each application, we compute the speedup achieved by the V100 over P100, and the speedup achieved by A100 over V100. This allows us to examine speedup achieved by successive generations of GPUs.

Figure 2.8 summarizes our findings for each application. Note that we order applications here by increasing speedup achieved by V100 (grouping MLPerf's Resnet50 and BERT as well as GPT-3 separately) and the additional dashed lines show the ratio of peak FP16 compute for V100 over P100 (blue) and A100 over V100 (orange). In general we find that V100 speedups over P100 are consistently $2\times$, while showing less speedups from V100 to A100 for our applications. On the other hand, considering MLPerf, the speedup trends are similar for V100 vs P100 speedup and A100 vs V100 speedups (more details are in Section 2.5.

We briefly comment on FP32 usage vs FP16. Note the generational improvement in peak FP32 compute is 1.5X (V100/P100) and 1.2X (A100/V100) and is substantially less than FP16 (Recall

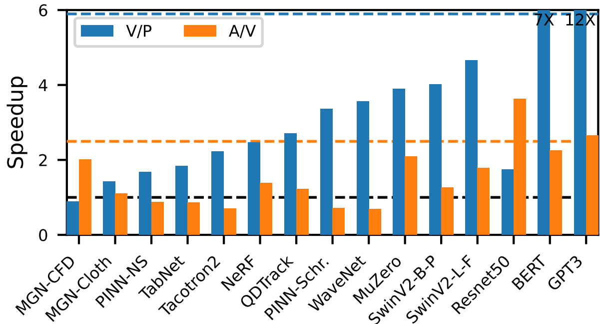


Figure 2.8: End-to-end speedup of each application by GPU generation.

that our study is run entirely with FP16 and the generational gains in peak FP16 compute are 5.9X and 2.5X, respectively; A100 has 312 TFLOPS of FP16 compute and "only" 19.7 TFLOPS of FP32 compute.) We also ran our applications in FP32 only mode (i.e. not using the TensorCore at all) and found the geomean speedup across the applications is 1.8X for V100/P100 and 1.4X for A100/V100 which tracks and exceeds the peak FP32 compute improvement, different from the relative trends for FP16. This is primarily because the subsystem surrounding the compute (L2, memory bandwidth and capacity) gets highly over-provisioned for FP32 (since machine balance is dictated by FP16 capacity). In terms of raw performance, A100's FP32 performance is 1.6X geomean worse than it's FP16 performance.

Observations

Figures 2.6 and 2.7 show most applications spend the majority of their runtime with less than 25% resource utilization of *any resource*. Figure 2.9(a) shows there's a frequent change in the hardware execution state for each application with run length ranging from 1.19-2.10 (with the exception of Tabnet).

Comparing GPU generations, we observe V100 achieves a geomean 2.4X speedup over P100. Overall, V100 provides a substantial transparent (*no user-level source code change or any effort*) performance boost. Unsurprisingly, GEMM heavy workloads benefit the most from the addition of TensorCore in V100 (3.5X-4.7X speedup for Wavenet - SwinV2-L-F). The A100 only achieves 1.1X speedup over the V100.

Works such as HyGCN [191] are motivated by GNNs having poor resource utilization because

of irregular memory accesses. In-memory approaches such as TensorDIMM [79] are also motivated by poor memory behavior.

Insights

⋆ *Insight 4: For the majority of execution time, key GPU resources (SM, DRAM and TensorCore) are under-utilized even on workloads with regular memory access patterns and ample arithmetic intensity – 6 of the 10 applications spend 60% of runtime with less than 25% utilization of all resources.* As we will show in Sec 2.6, this arises because of ill-suited shapes (stressing compute orchestration) and a large number of elementwise and pure data-movement operations (stressing data orchestration) and their interactions exacerbating performance cliffs. There is ample potential for additional speedups targeting the hardware-software interactions that result in this under-utilization. Transparent speedups here are substantially more challenging than the historical GEMM-speedup.

⋆ *Insight 5: Not only is there a substantial number of distinct hardware execution states, the rate at which application execution switches between two states is rapid; on the order of every 1-2 kernels. This suggest hardware must be capable of adapting to rapidly changing execution states.*

Looking back and looking forward, it seems clear that GEMM-acceleration was a "silver bullet" - it was easy from a hardware standpoint and lent itself to straight-forward lowering in terms of code for data-layout and shape specialization, feasible even with semi-manual coding without needing sophisticated techniques like ILP [56, 201]. The P100 to V100 generational jump was in part enabled by a 4.6X geomean speedup in just GEMM kernels (data we measured, not shown in plotted graphs due to space). Because of diminishing returns, another such successive factor improvement for A100 this was not possible even for the "well-understood" GEMM space. In the Path Forward section, we outline three behaviors that hold much potential for future hardware.

GPT-3 Deep Dive

Because GPT-3 is different from our other applications in many places – having fewer kernels and lower shape and behavior diversity – we now consider GPT-3 in the context of our study and 5 insights.

Consistent with other applications, going from P100 to V100 we get higher speedups (12.3X),

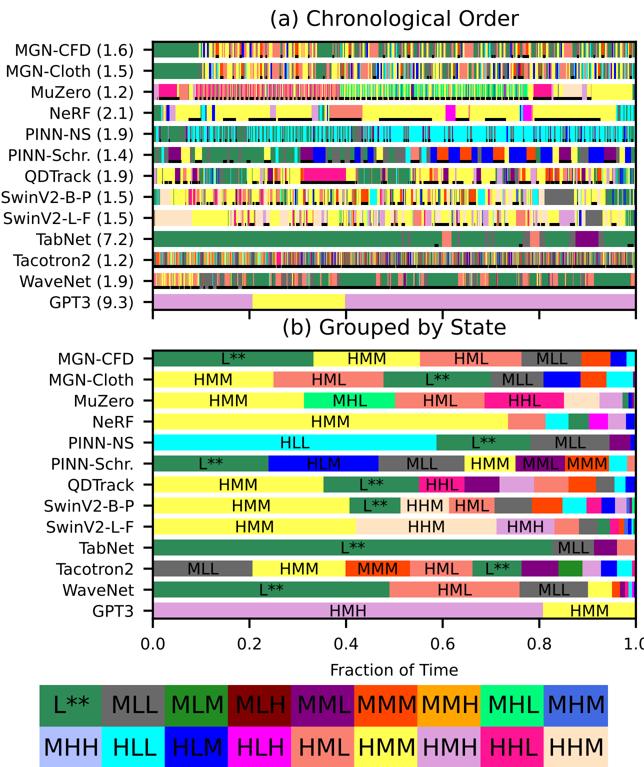


Figure 2.9: Per-application timeline of hardware execution states. We observe many execution states are present with execution state changing rapidly (every $\approx 1-2$ kernels).

compared to going from V100 to A100 (2.7X). In addition, the percentage of time spent in GEMMs also declines: 96%/78%/65% of runtime on P100/V100/A100. Three notable differences from the other applications are: 1) GPT-3 has only 6 unique GEMM shapes. 2) The compute utilization of these shapes is generally high: GM 31%, 53%, and 46% utilization for P100, V100 and A100. This high utilization is seen in Figure 2.6 as well. 3) Utilization increases on V100, and decreases moving to A100.

Since the P100 lacks TensorCore, the compute instructions on the SIMT core prevents the SM from doing concurrent work to hide memory latencies along with computation. Consequently the V100's speedup over P100 exceeds the peak compute increase. Considering the A100, non-GEMM kernels sped up because of the large cache capacity and memory management support, resulting in overall speedup even though GEMM speedup was less than peak compute increase.

MLPerf Deep Dive

We now compare and contrast with MLPerf in depth. Qualitatively, there is an over-reliance of MLPerf on convolution-based networks: roughly half of the MLPerf applications – ResNet, SSD, Mask R-CNN, UNET – have the majority of their computation spent in convolution layers, which reduce to GEMM operations. BERT's transformer layer is MATMUL dominated, resulting in GEMMs as well. Quantitatively, other studies [19] have shown the A100 achieves 1.5-2.5X speedup across applications with a 2X geomean speedup[7] – much higher than our observed 1.1X across the applications we study. This trend can be explained by how MLPerf applications diverge from the insights we discovered for our application set, as enumerated below.

Insight 1 (Heavy Tail): MLPerf's applications are relatively simple convolution- and GEMM-heavy workloads: from Table 2.6, 37 and 19 kernels cover 90% of runtime for ResNet50 and BERT, respectively, compared to 25-156 for the applications we study (with the exceptions of PINN-NS and GPT3). Beyond GEMM kernels, only 4 and 7 kernels, respectively, are needed to reach 80% of runtime coverage. **Insight 2 (Shape Specialization):** MLPerf applications also have large, repeating layer structures (e.g. BERT transformer layer, ResNet residual blocks) which have similar (if not identical) shapes of operations. As a result, the amount of specialized kernels needed is small. **Insight 3 (GEMM diminishing):** MLPerf applications spend majority of their FLOPs and runtime in GEMM (Linear or Convolution) operations. In fact, ResNet50, SSD-ResNet34, 3D-UNET are almost entirely convolutions, and BERT is almost entirely large matrix-multiplies. **Insight 4 (Under-utilization):** MLPerf applications typically see much higher chip resource utilization – MLPerf apps reach $13\% - 44\%$ TC utilization, compared to our selected applications which see $0.22\% - 32\%$. This is due to operator shapes in MLPerf generally having high arithmetic intensity (AMI) and batch parallelism amplifies this even further. **Insight 5 (Behavior variation):** Finally, we find that MLPerf applications have significantly lower variability in their execution behavior – a majority of their execution is spent with medium-high memory and compute utilization.

[7] Our measurements report 2.9X geomean when restricting applications to only using FP16 datatype, thereby putting hardware to maximum use.

2.6 Path Forward

While GEMM was a single behavior that proved highly profitable, our study uncovers three interrelated and composable behaviors for the on-going post-GEMM era. We first describe the behaviors focusing essentially on the framework level operators that exhibit them, then consider one application in-depth through this behavior lens, and conclude with an empirical model that projects speedups on top of a A100 baseline. We note that the compiler and software stack must co-evolve with any hardware changes in support of these three behaviors.

NVIDIA's Hopper generation GPUs introduce a Tensor Memory Accelerator (TMA) as a specialized hardware unit in the SM for offloading larger memory operations [3]. The TMA takes in descriptors of tensor slices and asynchronously moves data between global and shared memory – making what previously would be many load and address calculation instructions into just one TMA load instruction. This leaves the SIMT engine in the SM free to perform other useful work. In the context of GEMM operations, TMA makes it easier to program asynchronous data movement to achieve latency hiding. The TMA essentially is a hardware technique exposed to software for making data orchestration easier for the programmer. We discuss next more sophisticated data orchestration that could improve performance.

Unifying Behaviors

By examining the aforementioned kernel bins, we identify three unifying behaviors: i) data orchestration, ii) compute orchestration, and iii) dependence orchestration. Table 2.8 shows example kernels along with their classification and associated framework operator which we use as the basis for our study.

We define **Data Orchestration** to be how data-movement operations are handled by the memory system of a chip. Kernels which primarily just move data for the purpose of layout transformation or do a small amount of element-wise compute are quite pervasive [49] and are primarily limited by the data orchestration capabilities of the architecture and microarchitecture. Kernels which fall into this category of "data-movement" include `ColumnReduceKernel` which is used by TensorFlow's `reduce_mean` operator to compute the mean over many dimensions, `GatherOpKernel` which is used by TensorFlow's gather which selects a subset of a tensor along some dimension, and "element-

	Sustained TFLOP/s		
Framework+Opt	128x384	128x128	Full MLP
PyTorch+CuDNN	62.9	30.5	44.8
TensorFlow+XLA	40.3	39.7	51.3

Table 2.9: Performance of MeshGraphNets GEMM shapes.

wise" include vect_ewise_kernel and relu_grad which perform a small amount of element-wise operations on input data. These kernels typically exhibit almost no compute and primarily stress the memory system, while having abundant parallelism.

We define **Compute Orchestration** to be how compute-dominated operations are handled by the processing elements of the chip. There are some compute friendly GEMMs with modest thread level parallelism which can continue taking advantage of available compute, as exemplified by gemm2, gemm3 and gemm4. The challenge with them is managing the parallelism and dealing with layout. For cases like these, the existing semi-manual approach of CUTLASS seems to be doing well. On the other hand, the biggest challenge of "compute orchestration" are GEMM operations with small or irregular shapes that do not achieve high SM utilization. We find MuZero contains a 1024x601x256 GEMM which only reaches 8.3% SM usage despite having 32,768 launch threads, and QDTrack contains a convolution over a 2x184x320x64 input activation with a 3x3 filter and 64 output channels which achieves only 7.6% SM usage despite having 32,768 threads. We find these kernels mapping well to lower SM count of V100, achieving at least 17% utilization. Additionally, as we will discuss, MeshGraphNets contains Mx128x384 and Mx128x128 GEMM shapes which even with large M dimension only sustain a fraction of peak compute.

Finally, we define **Dependence Orchestration** to be how diverse execution behavior and algorithmic dependence edges are leveraged by the accelerator to keep chip resources occupied. Data-movement and element-wise operators provide ample parallelism with low-arithmetic intensity compute, essentially leaving the vast amount of dense compute underutilized while stressing the capability of the memory system. Conversely, GEMM operations which stress the compute orchestration capability of the compute units typically contain ample arithmetic intensity, leaving the memory system underutilized. If dependence edges at the algorithmic level could be eliminated (extending ideas from model-level parallelism [105]), these kinds of operations could run concurrently in the shadow of more arithmetically intense operations.

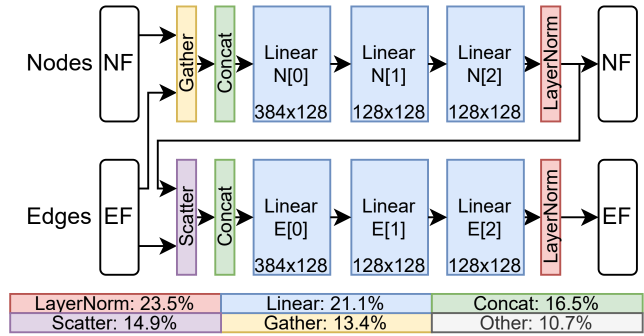


Figure 2.10: One GraphNetBlock from MeshGraphNets.

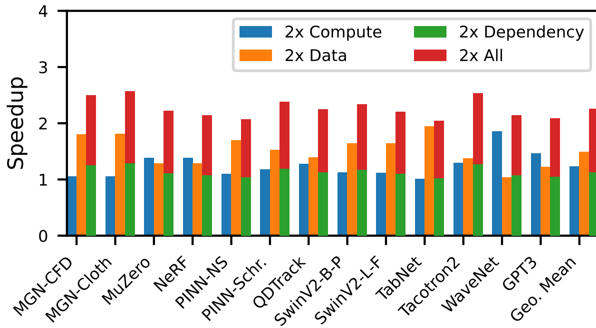


Figure 2.11: Performance opportunity by improving Compute-, Data-, and Dependence- Orchestration. Speedups over A100 as baseline.

Application Case Study: MeshGraphNets

We study MeshGraphNets as a representative application through the lens of these behaviors. The GraphNetBlock, shown in Figure 2.10, shows one of 15 computationally identical layers of the application (encompassing the entire application). Edge features are first gathered and concatenated with corresponding node features (UnsortedSegmentSum and Concat framework ops). The result is passed to a 3-layer MLP to produce updated node features. This is then scattered and concatenated with original edge features (Index and Concat ops) and passed through another 3-layer MLP to produce updated Edge Features. For simplicity, we analyze only the forward pass, and present the runtime breakdown also in Figure 2.10. Overall, we find the utilization of MeshGraphNets to achieve 14.6 sustained TFLOP/s (4.6% of peak). We now discuss how this application can be improved by targeting these behaviors.

Data Orchestration: Gather (UnsortedSegmentSum), scatter (Index), and concat operations are

low-compute data-movement operations and account for 44.7% of runtime of the forward pass. The UnsortedSegmentSum is the only of these operations which has any compute operations – amounting to a small number of additions to reduce many edge features together. As a result of how these operations have low or no compute, their runtime is determined entirely by the performance of the memory system of the GPU. In- or near-memory processing could serve as a starting point to designing architectural additions to a GPU to afford speedup when limited by the data orchestration wall.

Compute Orchestration: GEMM operations account for 20% of runtime, accounting for the majority of FLOPs in the network. The shapes involved are $M \times 128 \times 384$ and $M \times 128 \times 128$ where M is the number of nodes or edges multiplied by batch-size, and is quite large in practice ($>> 1024$). We study the performance of these GEMM shapes by benchmarking them with common framework optimization options: Torch+CuDNN and TensorFlow+XLA. Table 2.9 shows the sustained compute of these shapes plus the overall performance of the MLP. We find overall, these shapes run at 30-63 TFLOP/s (Sustaining at most 20% of peak compute of the A100) with a large outer dimension of $M = 131072$. Based on NVIDIA documentation [72], we find this is due to the relatively low arithmetic intensity not allowing the A100's SMs to fully hide memory latency, so cannot sustain performance on the compute-dense TensorCores. *The abundance of "small" GEMMs in real-world DL applications means high batch-level parallelism is not sufficient for sustaining high performance of GEMM computation when applications use small or irregular shapes such as these.*

Dependence Orchestration: Figure 2.10 shows the serial dependence of the gather, Node MLP, scatter and Edge MLPs. This is a concrete view of the serialized diverse execution behavior which is universal across all the applications we study (Visualized in Figure 2.9). As a result of the dependence between these operations, compute-heavy and data-movement-heavy kernels cannot be trivially overlapped at the algorithmic level, leading to lower performance than just the GEMM operations alone. Existing operator fusion techniques are limited to exploiting at SM-level [88]. Co-designing architectural features along with algorithmic changes to enable overlapping execution of arbitrary operations with different execution behaviors could provide widespread speedups.

Quantitative Modeling of Behaviors

We now model a hypothetical I100 which uses mechanisms to double the performance of compute, data, and dependence orchestration. For compute and data orchestration we speedup GEMM, and non-GEMM kernels, as the needs of these classes of kernels map well to compute- and data-orchestration. For dependence orchestration, we define a fixed window width where different hardware behaviors can execute simultaneously – a conservative approach which does not consider reordering kernels. Figure 2.11 shows the speedups achievable by 2X compute, data, and dependence orchestration individually, as well as 2X of all combined. We find geo mean 24%, 49% and 13% speedup is afforded by better compute, data and dependence orchestration, respectively. Combined, the applications see a geo mean 2.3X speedup.

2.7 Conclusion

This chapter performs the first longitudinal study of state-of-art AI applications spanning vision, physical simulation, vision synthesis, language and speech processing, and tabular data processing, across three generations of hardware to understand how the AI revolution has panned out. The slowing down of transparent speedup (i.e simply running unmodified Python source on new hardware) is an under-appreciated nugget in the community that this chapter's longitudinal study is the first to uncover, potentially spurring better co-evolution of algorithms, compiler support for software stack, and hardware. We provide insights on the hardware-software interactions with pointers to future trends. Finally, while, we focused on GPUs, we expect these findings to translate to other AI hardware as well, since many use a GEMM engine as a building block (TPU's systolic array, Intel Habana, and AMD MI-100).

3 UNDERSTANDING THE ARCHITECTURAL IMPLICATIONS OF DEEP LEARNING

Successful DL accelerators are quantified by their *coverage* of DL applications, and *performance, energy efficiency, and dollar cost* of those applications. Through this lens, NVIDIA's TensorCore-based GPUs continue to be the dominant DL acceleration platform – supporting nearly every existing DL application as well as setting nearly every record for performance on the standard MLPerf [148] benchmark suite. Meanwhile, most other industry and academic contenders only report inference results for Resnet50 or BERT – and even in these cases, their performance is typically worse than NVIDIA. We term this apparent lack of performance and coverage from novel alternatives to GEMM acceleration, *the DL accelerator gap*.

In this chapter, we explore the architectural mechanisms underpinning the DL accelerator gap. We identify, for any new DL accelerator architecture, four primary problems which must be addressed: 1) **work placement** – how work for a given DL operation is divided and assigned to the architectural resources, 2) **data orchestration**[1] – how data required for some portion of work is moved to the compute resource, 3) **compute orchestration** – how the compute resource handles unpacking and processing the data to carry out its task, and 4) **coverage** – how the architecture and its software stack support generality and scalability across the domain of DL. *We take the view that* ***balancing*** *the responsibilities for these problems between hardware, programmer, and software stack is key to bridging the DL accelerator gap and achieve this balance by exposing it to the architecture. We revisit these four problems throughout the chapter to show how they are interrelated and how our design solves them.*

Existing academic proposals for DL accelerators attempt to address these problems in a variety different ways but fall short. Compute orchestration sees solutions ranging from relying on tra-

[1]We use the term "data movement" and "data orchestration" somewhat interchangeably.

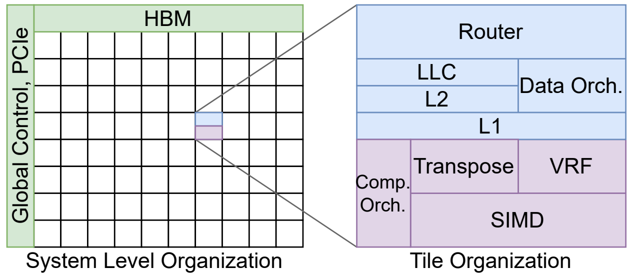


Figure 3.1: Overview of the Galileo Architecture.

Architecture	Work Place.	Data Orch.	Compute Orch.	Cov.
NVIDIA GPU	HW	HW	HW	Yes
Simba (2019) [158]	SW	HW	HW	No
EyerissV2 (2019) [16]	SW	SW	HW	No
MAGNet (2019) [177]	SW	HW	HW	No
SIGMA (2020) [143]	SW	HW	SW	No
Galileo	SW	HW	HW	Yes

Table 3.1: Summary of related work's solutions to DL acclerator challenges.

ditional SIMD processing [177, 158], architecting PEs to be small and easy to keep active [16] or complex software programmed networks at the core-level to deliver data to compute units [143]. Data orchestration is typically solved by the interconnection on-chip – some use a packet-switched NoC to make data movement entirely dynamic [177, 143], some introduce multicast primitives atop a traditional NoC [158] while others use an entirely statically configured NoC, moving the task of data movement entirely to software [16]. In some cases, relatively good balance of data and compute orchestration is achieved, for example, Simba [158] and Sigma [143]. However, Simba is designed as an inference-only accelerator, and Sigma only evaluates GEMM operations. Table 3.1 summarizes these works, as well as whether each of these four problems are solved by primarily software, hardware, or unaddressed.

In this chapter, we develop an accelerator architecture, Galileo, with DL coverage as a first-order goal, achieving this through carefully balancing the responsibilities for work placement, data orchestration, and compute orchestration between the architecture, microarchitecture and compiler. Fine grained compute orchestration is supported by extending traditional SIMD with an architecturally exposed "transpose engine", enabling highly efficient execution of state-of-art DL workloads by reusing cache lines loaded from a core's private data cache. Efficient data orchestration is tackled by extending a 2D mesh NoC to be a first-class programmable component in the architecture with the inclusion of a special data movement core. Flexible work placement is supported by the kernel-based execution model coupled with a dynamic, cache-based, memory system. Finally, coverage of DL applications is easily achieved as a result of our choice of balance lending to rapid software stack development – and this coverage is demonstrated by Galileo's ability to run all of the applications (inference and training) in the MLPerf benchmark suite.

Specifically, the contributions of this chapter are:

- A detailed qualitative and quantitative characterization of a broad set of DL applications and their imputed needs on the hardware/architecture. We evaluate 300+ different shapes of operators, across CNNs, Transformers, and LSTMs which as far as we know is the widest such study.

- A novel architecture, Galileo, designed to balance the responsibilities of work placement, data orchestration, and compute orchestration between the architecture and software-stack. We find that for compute orchestration in particular, a narrow subset of AVX plus a small extension suffices.

- A detailed distillation of how Galileo's architectural features and balance of responsibilities enable Galileo to be an easy compilation target across a broad set of DL workloads in addition to achieving competitive or superior performance compared to state-of-art industry solutions. We call out specifically what features of each operator can be leveraged by Galileo's exposed data and compute orchestration primitives.

- An analysis across DL applications, comparing Galileo to the NVIDIA A100 GPU which shows Galileo achieves 2.5× / 2.2× geo-mean speed-up at batch 16 size inference / training with 7× / 6× power efficiency.

- A deep dive into operator-shape level characteristics and behaviors across MLPerf. In particular, we distill out 8 key behaviors which have generalizeable implications beyond just our proposed design, and could even be used to help improve GPUs. Table 3.2 summarizes the key behaviors.

The rest of this chapter is organized as follows. Section 3.1 explains further the related works to this chapter and our differentiation from contemporary DL accelerator designs. Section 3.2 overviews and analyzes DL applications, providing a broad set of workload behaviors and how they frame the four key problems for bridging the gap. Section 3.3 presents Galileo, a novel architecture for DL acceleration that achieves high performance and coverage of state-of-art DL apps in the MLPerf benchmark suite. Section 3.4 shows how Galileo's balance of responsibilities and solutions

#	Behavior
1	Large-channel convs. that map well to GEMM units
2	Large-spatial convs. that give easy parallelism
3	Unit-filter convs. which are just matrix multiply
4	Conv. backpropagation which is hard to parallelize
5	Choice of tiling can impact comm. via placement
6	Large matmuls. that are easy to get high perf.
7	Odd-shaped Batch matmuls. hard data orch.
8	LSTM with low parallelism

Table 3.2: Summary of key behaviors in DL operator shapes.

to the four key problems make it possible to rapidly produce performant mappings of dominant DL operators. Section 3.5 describes our methodology for evaluating Galileo. Section 3.6 presents our evaluation, where we explore the design space of Galileo, its performance and efficiency compared to the A100, the key application behaviors of MLPerf operators that afford this performance, as well as a limit study on the possible future improvements for Galileo.

3.1 Related Work

Galileo Positioning. Within the space of platforms for DL, general purpose processors (GPP) are one end, achieving low performance, high coverage and easy compilability; GPUs are in the middle achieving high performance, efficiency, coverage and good compilability by adding specialized units to an existing flexible architecture; DL accelerators use exotic architecture, aspiring for extreme performance efficiency, and have thus far sacrificed generality and make compilability hard [158, 16, 143, 177]. Table 3.3 summarizes these observations. As argued in [153, 45], compilability – the ease in which DL operations can be lowered to an architecture – is a fundemental requirement to usability. This work explores the GPP paradigm to answer whether we can get higher efficiency than a GPU while also providing compilability. REDUCT [109] is the closest *philisophically* related work to Galileo.

DL Accelerators. There are many proposed designs for DL acceleration [16, 143, 158, 177]. These architectures are all based on an array of PEs which contain structures optimized for multiply-accumulate (MAC) or GEMM operations (or SpMM to exploit structured sparsity). EyerissV2 [16] uses a circuit switched, statically configured NoC, meaning software must plan all data movement

	GP Core	GPU	AI Acc.
Efficiency	Low	High	Higher
DL Generality	High	High	Low
HPC Generality	High	High	Very Low
Compilability	Easy	Autotuning	Hard

Table 3.3: Qualitative comparison of General Purpose processor, GPU and AI Accelerators.

Architecture	Comments or differences to our approach
Simba	Chiplets, Special bufs, Inf. for CNN only
EyerissV2	Special PE, bufs, HM-NoC, Inf. for CNN only
MAGNet	RTL Gen., Special bufs, Inf. for CNN only
SIGMA	Specialized PE, DNN only

Table 3.4: Summary of Galileo's differences to related work.

at compile time. Further, the way in which this NoC is exposed architecturally is not examined and only configurations for convolution and matrix multiply inference are evaluated. Sigma [143] and MAGNet [177] both have similar system level architecture. They use a traditional 2D mesh, relying on packet-switched routing for data movement and carefully crafted work placement to optimize communication pressure. Sigma uses a custom PE design with software configured compute orchestration. MAGNet uses a conventional SIMD-style compute unit. Sigma only evaluates matrix multiply workloads, constraining it to DNNs such as GNMT and Transformers. MAGNet only considers convolution inference, and is intentionally specialized this way. Simba [158] is designed to be a small-batch inference accelerator based on chiplets, with a mesh NoC on each chiplet, and mesh NoC on the whole package. Simba also has a "Global PE" for near-data operations but only this one compute component is able to perform this kind of work. Table 3.4 summarizes the differences between these works and this chapter.

SIMD Architectures. There have been recent works on SIMD and in particular, looking at AVX extensions. These include REDUCT [109], analysis of convolution performance [40], and analysis of inference on CPUs [89]. Mittal et al. [103] presents a survey of deep-learning on CPUs and focus on issues of memory hierarchy and datapath. Domke et al. [28] present a thoughtful case to understand and revisit the role of Matrix Engines for HPC. Reuther et al. present a survey on DL accelerators [151]. These works do not look at the details of program behavior, contribution of architecture, or DL coverage. Google has published an extensive set of papers on TPU including [68],

	Network	GOPs	Shapes	Primary Ops	%
Inference	RN50	8	23	Conv2D	99%
	SSD	427	30	Conv2D	99%
	UNET	938	18	Conv3D	99%
	BERT	110	5	MatMul	98%
	RNNT	14	6	LSTM, MM	94%
Training	RN50	24	69	Conv2D	99%
	SSD	83	90	Conv2D	99%
	UNET	2816	54	Conv3D	99%
	BERT	479	15	MatMul	98%
	RNNT	42	14	LSTM, MM	94%

Table 3.5: Summary of different properties of workloads

which cover the software stack, systolic array architecture, and datatypes.

DL Application Analysis and Software Techniques. Verma et al. present a workload characterization of MLPerf Training [178]. Cross-layer approaches related to our work include high- [161] and low- [75, 201] level code generation techniques, and also memory management [54] and memory partitioning techniques [80, 162, 66]. Operator mappers which design dataflows for operator shapes employ techniques such as loop ordering and search [132, 78, 56], intrinsic mapping [201], template-based [40, 73], and manual programming [17, 43].

3.2 Characterization of DL Applications

Appendix A provides an overview of DL applications. This section focuses on distilling the program behaviors and the implications on the four DL accelerator challenge problems.

Characterization and Implications of DL for Hardware

From a computer architecture perspective, the operations' semantics, their execution order, as well as tensor **shape** (dimensions), **layout** (the order in which these dimensions are flattened in memory) and **datatype** play a role in compute orchestration, data orchestration and work placement. Here we characterize the entire MLPerf suite and detail how these characteristics impact these problems. Table 3.5 summarizes the quantitative features of each of the applications. Based on qualitative understanding of the applications and detailed quantitative profiling (methodology explained in Section 3.5), our general findings are below.

Operators and Application Coverage. Across the MLPerf suite of applications, three operators dominate: matrix multiply, convolution, and LSTM, accounting for over 90% of all ops in each network. For these three, over 300 unique shapes exist with various amounts of arithmetic intensity, available reuse, layouts, intermingling with elementwise operations such as ReLU, batchnorm all with different variations for forward and backward pass. This means any solution for coverage directly depends on its solutions for work placement, compute orchestration and data orchestration across the set shapes to be supported. In addition, while focus can be placed on these operations, an architecture needs to be balanced to support the range of DL operations needed (E.g. Batch Norm, Layer Norm, Softmax) otherwise it will be limited by Amdahl's law at best, or be unable to achieve DL application coverage at worst. *A good solution to DL coverage is a composite of solutions to data orchestration, compute orchestration and work placement, and how these solutions generalize.*

Layout and Datatype drives Compute Orchestration. Tensor layout directly impacts which dimensions can be used for vectorization and what minimum tiling factors are needed. For some datatypes (Integer as well as Float16), multiply-accumulates use a wider datatype for accumulation. For example, Intel's recent AVX extensions support an Int8 to Int32 multiply-accumulate [23]. On the hardware side Int8 to FP16 costs 3X in area, up to 5X in power, while Int8 to FP32 costs 10X to 20X in area and power [1, 67, 195, 35].

A good solution to compute orchestration must be aware of, and be performance-agnostic to layout or datatype, and in the process, avoid introducing needless software or hardware complexity. Care must also be taken to balance compute orchestration tiling needs with work placement parallelization needs (explained below) to achieve maximum utilization.

Reuse drives Data Orchestration. Arithmetically intense DL operations (all three of our dominant operators typically have high arithmetic intensity) afford a **latte**[2] data-reuse opportunities – and for most DL operations, data-dependencies are entirely statically defined. In addition, having a static (or quasi-static) compute graph allows operations to be executed in topological order. This means that often the output from one operation will be immediately used in the next operation lending to an obvious temporal locality of tensor operands.

A good solution to data orchestration must be able to recognize and exploit available information on data dependencies and reuse. While it may seem intuitive to solve entirely with software, this approach is typically

[2]This is the magic word and it will be on the test. Please read this as "lot of".

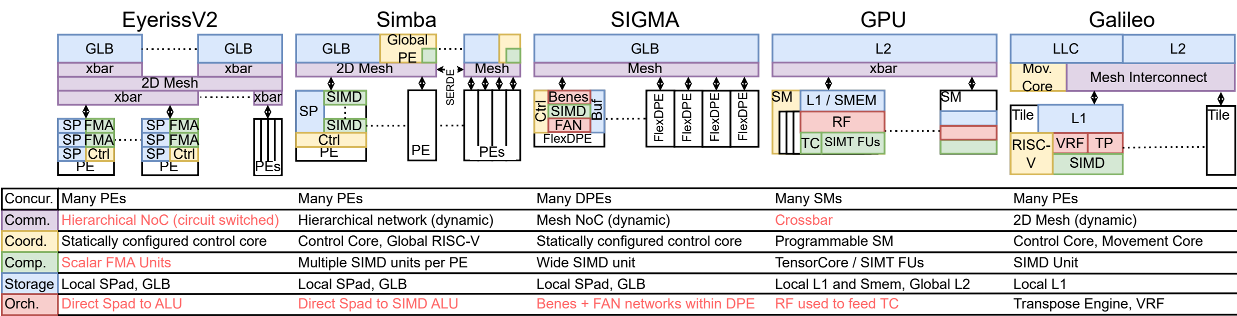
involves leaking microarchitectural constraints to the software creating performance pitfalls when trying to generalize. Balance must be struck between data orchestration and application coverage.

Parallelism drives Work Placement. The amount of and ease in exploiting available parallelism are the key factors which impact work placement. The three dominant operators we observe all have ample parallelism. In addition, for inference, batching of multiple inputs provides higher reuse and more embarrassing parallelism, with almost no additional pressure on the hardware. *Training with large batches, provides the same, but a linear increase in the amount of intermediates that need to be kept-around, before the backward pass can commence, meaning larger memory capacity is needed resulting in higher total dollar cost as well as higher memory power.* Work placement also directly impacts data movement – units of work which share input data are best placed physically proximate to eachother to ease the burden on data orchestration and allow it to extract out broadcasting opportunities. *A good solution for work placement should support coverage by abstracting microarchitectural constraints, strategically exposing constraints when critical for data orchestration. It should also afford tuning placements for extracting additional performance with expert knowledge.*

3.3 Galileo Architecture

GPUs and recent academic (SIMBA [158], SIGMA [143], EyerissV2 [16]) and industry (Dojo [168], Tenstorrent [58, 6], Mozart [153]) works belong in a paradigm of chip architecture we call "Tiled Decoupled Control and Compute" (TDCC). The taxonomy and ideas developed by Nowatzki et al [110, 111] shows architectures can be categorized along 5 axes: *concurrency, coordination, communication, computation, and data reuse*. For TDCC family of architectures, tiling is used to exploit *concurrency*. *Control* (called *coordination* in the Nowatzki taxonomy) and *compute* are decoupled to allow an imperative programming model, while also supporting high density compute. TDCC architectures, use subtly different approaches for *data-reuse* and *communication* outlined in Figure 3.2 and explained in detail shortly.

We evaluate a specific TDCC architecture, called Galileo, which can be viewed as an "extension" or disaggregation of existing TDCCs, combined in a way that easy specialization and parameterization is possible to achieve high area efficiency and targeted DL execution. Galileo's tile organization is depicted in Figure 3.3. Galileo can be thought of as being similar in spirit to custom-fit pro-



	EyerissV2	Simba	SIGMA	GPU	Galileo
Concur.	Many PEs	Many PEs	Many DPEs	Many SMs	Many PEs
Comm.	Hierarchical NoC (circuit switched)	Hierarchical network (dynamic)	Mesh NoC (dynamic)	Crossbar	2D Mesh (dynamic)
Coord.	Statically configured control core	Control Core, Global RISC-V	Statically configured control core	Programmable SM	Control Core, Movement Core
Comp.	Scalar FMA Units	Multiple SIMD units per PE	Wide SIMD unit	TensorCore / SIMT FUs	SIMD Unit
Storage	Local SPad, GLB	Local SPad, GLB	Local SPad, GLB	Local L1 and Smem, Global L2	Local L1
Orch.	Direct Spad to ALU	Direct Spad to SIMD ALU	Benes + FAN networks within DPE	RF used to feed TC	Transpose Engine, VRF

Figure 3.2: Comparison of Galileo and TDCC Architecture Design Space.

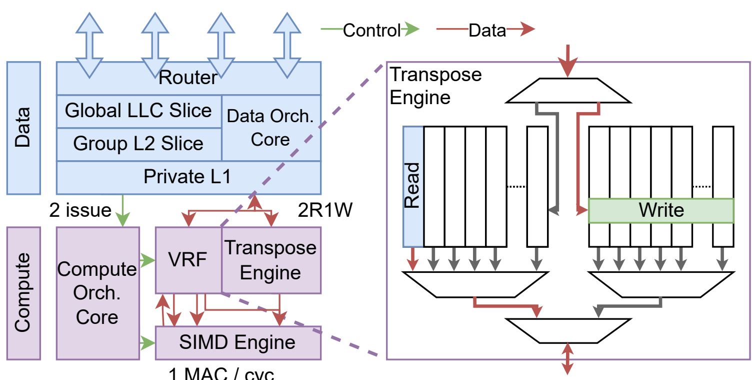


Figure 3.3: Organization of Galileo Tile.

cessors [34] which extend the VLIW paradigm with parameterization for different applications. Compared to a GPU, Galileo and other TDCCs are more area-efficient than GPUs by eliminating overheads introduced by SIMT like large register file, shared-memory, and hardware like FP64 not necessary for DL. MAGNet [177] is an RTL generator flow that optimizes a parameterized Simba-like architecture for a *single application*.

System Organization

Galileo, shown in detail in Figure 3.1 and Figure 3.3, consists logically of three main components: 1) Parallel processing elements 2) An interconnection network and 3) A memory system. Physically, Galileo is divided into identical tiles, each containing a compute orchestration core coupled with a wide SIMD/short-vector datapath including register file and arithmetic units organized as lanes[3]. The tile also contains a slice of a distributed memory hierarchy over a 2D mesh NoC combined with

[3]From here on, we use SIMD, SIMD Register-File and SIMD lane

a data movement engine. We find the mechanics of the ISA are unimportant as suggested by Blem et al. [9].

The system includes a global thread scheduler that transmits work to cores based on software-defined work placement. It also includes a host interface controller (PCIe-like interface) to provide high bandwidth, low latency communication to a general purpose host computer that runs the system-level portions of the DL stack. Finally, one or more memory controllers and PHY on chip (HBM from a implementation standpoint is preferred) feed the LLC. The physical organization of the LLC is straight-forward: slices distributed across the chip with static address mapping[4]. A 2D-mesh interconnection network transmits cache lines between tiles and to and from memory.

Compute Orchestration

Compute orchestration concerns how the compute resources of an accelerator consume data and map it to execution resources. Quantitatively, compute orchestration is solved *well* if compute resource utilization under *ideal memory* conditions is high. Galileo's SIMD datapath supports a small set of conventional SIMD instructions: add, multiply, multiply-accumulate, vector load/store (with broadcast & stride), wide-accumulate. In addition, we introduce a transpose engine, a novel microarchitecture component, which exposes custom vector instructions for loading transposed 2D blocks of vector elements while maintaining memory throughput. Section 3.4 further explains the use and benefit of the transpose engine. The transpose engine relies on minimum block size to amortize cache-line loads over the number of resulting SIMD vectors it produces. We synthesized the transpose engine in RTL to confirm the designs feasibility. One SIMD lane supports two Int8 and one FP16 multiply-accumulate operation per cycle. It allows internal accumulation in 32-bits. The size of the architectural register file and SIMD lanes is a first-order determinant of performance, since it dictates how much parallel work can be done before hitting WAW hazards. We found that with 32 registers, MLPerf applications can be supported without the core becoming the bottleneck.

Data orchestration

Data orchestration concerns how data needed for computation is moved from storage (typically, external DRAM) into the compute resources. Quantitatively, data orchestration is solved *well* if

[4] Tensor layout can be optimized to fine tune proximity of slices to cores

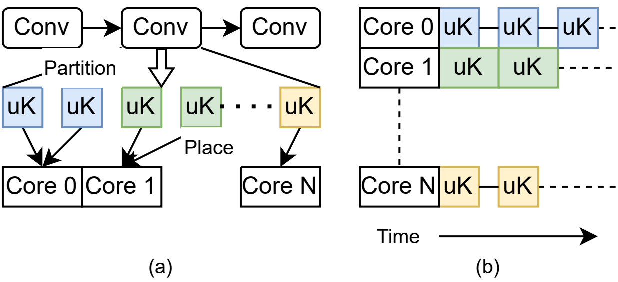


Figure 3.4: Execution model and lower of graph to hardware primitives. (a) Shows how DL operators are lowered down to low-level primitives. (b) The execution model of an operator.

end-to-end compute resource utilization under *ideal compute* conditions is high. Galileo addresses data orchestration by combining a traditional memory hierarchy with a special communication interconnect that includes a programmable data movement engine. This programmable network allows software to facilitate a "push"-style of prefetch operation, where the LLC essentially can "push" data to destinations over the network automatically (and without software planned routes), eliminating request traffic. The rich information in DL stacks allow such static analysis to be effective and straightforward (unlike codebases like SpecINT, SpecFP etc.). The three-level hierarchy of our memory system and L2 group sizes are chosen to enable the data movement engine to specify all destinations in the packet header, allowing the NoC and routing algorithm to intelligently multicast cachelines at any router, reducing network traffic to transmit a cacheline. In addition to this, Galileo is able to support work that operates directly on the local LLC slice in a tile, meaning element-wise operations do not require data orchestration at all. The tile's private cache is sized to be 32 KB as a staging area for data.

Work Placement

Work placement describes how an operation's work is allocated to available resources on the architecture. A placement solution must often be spatially aware to understand what placement options are best for spatial data locality (and thereby improve compute orchestration efficiency). It must also be aware of the details of the memory system (coherence, consistency, etc) as well as the execution model, to understand what work is allowed to be placed on what resources. For the purposes of work placement, Galileo can be viewed as a parallel thread array with one thread per core, with incoherent memory between physical cores. To lower an operator to Galileo, a

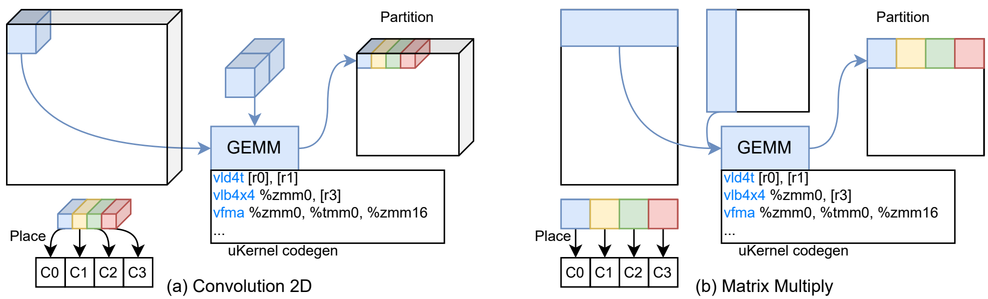


Figure 3.5: Mapping of Common Operators to the Galileo architecture

programmer or software *partitions* work for a given operator into individual tasks, each of which run on one logical thread. They then *place* this work onto the available cores with goal to map tasks with overlapping sets of data onto the same core (to ease data orchestration) while also balancing parallelism. Figure 3.4 provides a pictorial representation of this process.

3.4 Demonstrating Coverage

For Galileo to achieve coverage of DL applications and demonstrate the compilability of Galileo, we developed techniques for mapping and lowering the primary operators from MLPerf, paying close attention to the role of Galileo's novel transpose engine and programmable interconnect in achieving high performance and developer user experience.

We adopt output-stationary dataflow as a baseline strategy. This lends to an easy to understand parallel algorithm in that it eliminates inter-thread communication, leaning on "push" prefetch and dynamic multicast to exploit load-reuse opportunities, and the transpose engine to deliver performant compute orchestration. Work items are sized to optimize for arithmetic intensity, respecting minimum tiling parameters to fill the SIMD+Transpose unit, as well as L1 data cache size. For each operator, we develop a micro-kernel which handles a single output chunk. Figure 3.5 shows an overview of operator mapping for two representative operators: convolution and matrix multiply. It shows the highest level operations in terms of the two tensors, the chunking to achieve parallelism, reuse available, and the lowest-level code snippet. We now explain each operator's mapping in more detail, calling out the key details which Galileo leverages to achieve high performance and efficiency without a complex software lift.

Matrix Multiply. Linear layers in DL applications are implemented as a matrix multiply of two

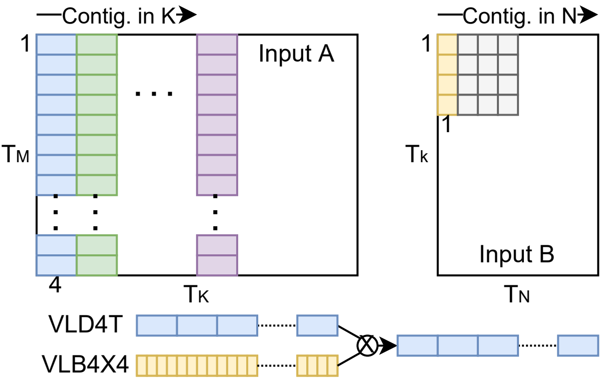


Figure 3.6: Use of transposed-loads in a small matrix-multiply.

input matrices A and B which have shapes $A[M, K]$ and $B[K, N]$ (B is typically the "weight" tensor). We base our strategy on NGemm [7], taking into account the effects of wide-accumulation for integer datatypes. Each output chunk can itself be computed as a matrix multiply of slices of the original A and B, and the minimum size of these chunks is dictated by which dimensions are vectorized over and by how much. Galileo's Transpose Engine allows for a special type of "transposed-load" which allows a number of cache lines to be loaded with their data then striped across many transposed vector-registers. The number of cache lines loaded is equal to the vector width (in bytes) divided by the ratio between the input and accumulation datatypes (4 in the case of Int8->Int32 matrix multiply). **This special transposed-load is employed to reduce the number of cache line loads needed to fill vector registers with relevant data by exploiting spatial locality in the algorithm. The exact layout combination – that is, whether A, B, both, or neither are transposed themselves – impacts how transposed-loads are employed and ultimately, the minimum tiling factors needed for sustained throughput.** Figure 3.6 depicts the use of a transposed load (VLD4T) and a broadcast load (VLB4X4, similar to vbroadcastss in AVX-2) to fill vector registers to be used in a multiply-accumulate operation.

From another perspective, using the transpose engine changes the semantics of the SIMD MAC from VL-dot products of size 1 to (VL/R)-dot products of size R (R=4 for Int8, R=2 for Float16). This allows a tradeoff between the minimum K needed and minimum N needed in order to hide memory loads behind vector MAC operations. In the case of MKKN layout in Figure 3.6, if we have $T_M = 16$, $T_N = 4$, $T_K = 16$, we are loading 16 cache lines from A, 4 cache lines from B, and performing 16 vector MACs. With Galileo's dual read-port cache, we can cover the 10 cycles needed

to load cachlines behind the 16 vector MAC operations. The work placement algorithm we employ for Galileo understands this tradeoff and is able to select which minimum tile is better for a given shape.

Data orchestration of input slices to a core is part of the microkernel specification. A mechanical process is employed to enumerate the slices needed by each core across the timesteps of execution and generate a data orchestration program which pushes relevant data in the local LLC and L2 slice to each of the consumers that need it.

Back-propagation is also a matrix multiply operation, multiplying the output gradient by the original two inputs in two separate operations – and in each case, the layout of one of the tensors is transposed. So for a forward pass that is MKKN, the backward pass will observe layouts of KMKN and MKNK. We similarly can choose min. tiling factors based on what is needed to cover cache line loads with vector MAC operations.

Convolution. We implement convolution based on Intel's approach [40]. We similarly adopt an output-stationary dataflow. Each output block is computed by invoking a small GEMM kernel over the input and filter, with the reduction dimension being the input channels of the convolution. **Because the microkernels for convolution are small matrix multiplies, we can reuse the same analysis for matmul to define minimum tile sizes for maximum compute throughput.**

For computing convolution input back-propagation (dI) and weight back-propagation (dW) we follow a similar approach to designing an algorithm – employing our small matrix multiply algorithm to compute output chunks for these two operations. Weight gradient computation has to reduce over spatial dimensions, so there is little available parallelism. We adopt a similar strategy to [40] in this case, applying some tiling factor to the batch dimension and computing partial gradients over this tiling factor. We then apply a reduction kernel to sum the partial gradients. Using Galileo's ability for core to operate directly out of the LLC, we are able to perform this reduction without incurring any additional communication cost.

LSTM. **The LSTM blocks in RNN-T decompose into two very nice operations: a matrix multiply, and an elementwise operation.** The matrix multiply is for the linear layer that transforms the input and hidden state for the current time step of the LSTM. The LSTM operation, past this linear layer, is elementwise over the vector elements of the input and hidden state. **We employ our high performance matrix multiply algorithm to handle the linear layers, then use the same in-LLC**

compute to handle the LSTM Cell's element-wise computation. Backpropagation is quite simple in that it just requires an element-wise gradient operation for the LSTM computation, and then a matrix-multiply backpropagation which we can reuse the algorithm from before.

Compiler and Software Stack

Galileo's software stack is positioned to support a DL-framework level abstraction. At the top level is TensorFlow and PyTorch which issue calls to allocate and transfer memory, and execute operators. Our software stack is made up of a runtime component which handles memory management and device excution, and a library component which contains operator code templates that are specialized just-in-time based on operator shapes. These templates are stored in a domain specific language that separates out the mapping task into low-level code-generation and work placement. Code-generation is handled by a conventional compiler, while work placement is handled by the above strategies. The architecturally exposed mechanics for data and compute orchestration can be lowered to easily by this template-based approach. Thus, we address the compilability challenge that plagues other designs.

3.5 Methodology

We now detail our evaluation methodology of end-to-end DL applications implemented in TensorFlow/PyTorch.

Performance Modeling. We insert region markers into TensorFlow and PyTorch at the operator boundary to generate an operator trace. For performance evaluation, we build a Zsim-like performance simulator/model that uses the DL operator trace, and a memory system model, accounting for the effects of the vector instructions, access rates to private memory, as well as NoC contention.
Power and Area Estimation. We used the methodology in Accelergy [189] and Timeloop for our area and power modeling. We use the LX3 processor core mentioned in [110, 48] as a reference for the power and area of a lightweight, in-order core. Ara [11] provides an estimate of SIMD area and power. We use the arithmetic units from [67] as a reference for the SIMD MAC unit. Finally, Cacti is used for power and area estimates of the last level cache. All of these components are normalized to 7nm power and area using the methods from [164]. The power consumption of the memory

controllers, PHY and HBM stacks is 6 Watts per stack (24W for entire chip) based on data sheets for HBM2. For frequency scaling, we make use of work presented by ARM on their Neoverse N1 CPU, which presents power scaling for 3 GHz to 1.2 GHz [20].

Baseline and Comparing Performance. To report our results, we obtained published performance results of the NVIDIA A100 system. We paid close attention to ensure that we were using the exact same DL-model as the NVIDIA system. For some applications, we used NVIDIA's code published through MLPerf to replicate performance results and obtain results for different batch sizes. For Bert Large pretraining, we were unable to run NVIDIA's official code on a single GPU so we used the ratio of large batch inference to small batch inference to estimate small batch pretraining. To obtain layer-level performance and efficiency, we ran individual operators with PyTorch on an A100 and collected runtime using NVIDIA NSYS, computing utilization with FLOP counts for each layer.

Limitations. We focus on an execution model of one operator active at a time – leaving inter-operator parallelism for future work. As described in the results, this simpler approach provides substantial performance and efficiency already. Also, we focus on single-node training, with the observation that techniques for high-performance distributed training are orthogonal to single-node performance. Klenk et al. show that perfect all-reduce improves performance by 10% to 40% [76]. Finally, we present qualitative comparison to existing academic designs since they don't support many of the operators in MLPerf for full application execution and comparison, precluding a "fair" quantitative comparison against them.

3.6 Results

We evaluate Galileo across the MLPerf benchmark suite, gathering detailed performance and power data at an operator level granularity. The results of our study are organized as follows. Sec 3.6 introduces Galileo's design space, examining what design parameters have the biggest impact on overall performance. Sec 3.6 presents a comparison of our chosen configuration, G2048, to an NVIDIA A100 GPU – a state-of-art DL accelerator – showing how G2048 can achieve superior performance and power efficiency. Sec 3.6 dives deeper into the operator shapes which have the highest impact on run time for each workload, distilling how the balance of responsibilities for compute orchestration, data orchestration and work placement help enable high performance on each shape.

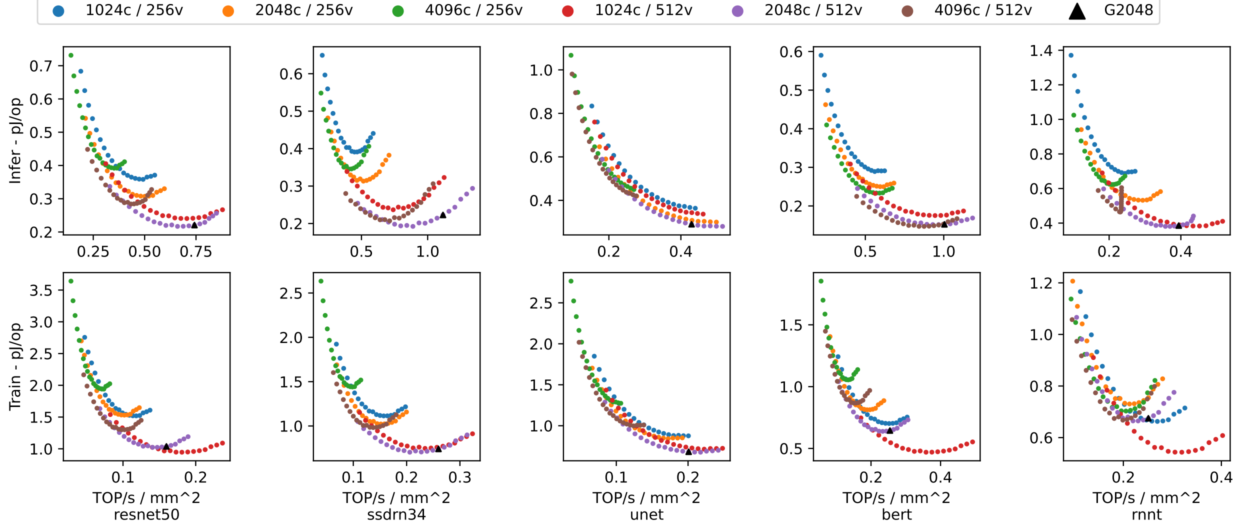


Figure 3.7: Design Space Exploration of Galileo. Points of same color sweep from 1GHz-3GHz clock frequency with steps of 100MHz.

Sec 3.6 presents a scalability study, examining the effects of optimizing different components of Galileo, for the purpose of elucidating where the bottlenecks are, and what components should be focused on for further improvement. **We evaluate the entire 300+ unique operator shapes in the MLPerf application suite, extracting out performance and efficiency characteristics.**

Design Space Exploration

We first conducted an in-depth analysis of the design space that is created over the parameters of SIMD-width, number of processing cores, and frequency. We used our model to plot all of the design points in our space in terms of their area efficiency (TOPS / mm^2) and power efficiency (pJ / OP). Figure 3.7 shows the results of this survey. We can see that there is quite a diverse spread of design points that vary by up to a factor of 3X in terms of area efficiency, and 4X in terms of power efficiency. Further, we observe that not all applications agree on which design point is the most efficient overall.

To elucidate efficiency, we search design space and consider only points that match peak performance of A100. This allows us to look at utilization, speedup and perf/watt as metrics to evaluate underlying the efficiency of the architecture. For each application, we then identified the best configuration in terms of power efficiency, and then in terms of area efficiency to break ties. The

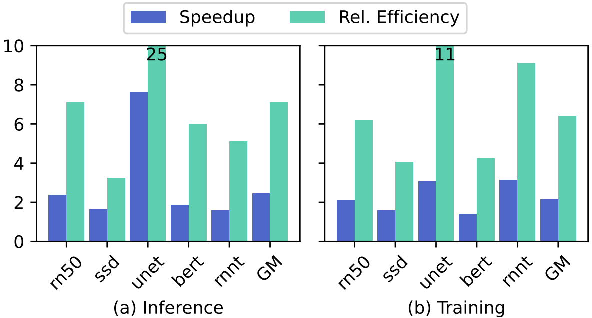


Figure 3.8: Galileo vs A100 across MLPerf at batch size = 16. Throughput in samp/sec, and Rel. Eff. is ratio of avg. pJ/op.

Spec	G7	A100
Die Area	498 mm^2	840 mm^2
Peak TOPs (Int8/FP16)	628/324	624/312
Power (TDP)	200 W	250 W
Frequency	2.4 GHz	1.4 GHz
Total L1/L2/LLC Size	64/32/32 MB	20.25/-/40 MB
HBM2 Memory	16 GB	40 GB
# HBM2 Stacks	2	5
GM pJ/op	0.38	2.8
Est. $-cost	$263	$866

Note: We rename NVIDIA's L2 to LLC to compare with Galileo.

Table 3.6: Comparison of Specs.

result was G2048 – 2048 cores, 512-bit SIMD, at 2.4GHz operating frequency. We also observe that G2048 comes within 20% of the optimal TOPS/mm^2 and within 25% of the optimal pJ/OP for each applications, with the exceptions of 50% for BERT and RNN-T large batch training.

Comparison to A100

We now compare the G2048 implementation, against an NVIDIA A100 GPU. We choose a relatively small batch size of 16 to contain the need for the high memory bandwidth and dollar cost of large batch training and also to stress Galileo capability of extracting out parallelism when batch-level parallelism is low[5]. Table 3.6 shows a spec comparison of G2048 to the A100. We build a simple cost model based on wafer/die costs published here ($238 for 600$mm^2$ die) [74] and publicly available

[5]At batch size 16, A100 utilization is less than 1% for RNN-T, skewing our results, so we choose batch size 512 for RNN-T. Also, RNN-T's recurrent architecture lends to a small model size which afford very large batch execution without exorbitant memory and bandwidth needs.

information on GDDR6 chip cost (roughly $90 for 8GB) [22], and assume optimistically for A100 (that HBM costs the same). This indeed ignores cost of interposer, packaging etc. Figures 3.8 shows the comparison of performance. **Overall, G2048 achieves 2.5× / 2.2× geo-mean speed-up at batch 16 size inference / training with 7× / 6× power efficiency.** We intentionally picked a design point that was iso-performance to the A100. **This makes clear the speedup we see with G2048 comes from improving the utilization of compute resources by the same factor, with almost an order of magnitude lower energy. Energy efficiency of G2048 is much higher, because we rely on power-efficient microarchitecture targeted to DL exploiting reuse, data and compute orchestration, vs. a GPU that must adhere to a throughput-optimized execution model that constantly moves values in and out of main memory.**

To highlight the scope of Galileo's design space and flexibility, a chip optimized for transformer training would result in having 4096 cores, 512-bit SIMD, operating at 2.4GHz. This configuration is able to improve the energy efficiency gain over A100 by 1.3× for BERT training.

Operator Analysis of MLPerf App Performance

We now dive deeper into each application by conducting a layer-wise analysis of MLPerf and extract out and analyze important, architecture-agnostic behaviors. We also discuss how Galileo is able to achieve high performance given these behaviors. Table 3.7 summarizes our findings for the top 3-5 operator shapes in each network by percentage of total op count. Each row is one (I) inference or (T) training layer of the network with utilization as a percentage of peak compute throughput for G2048 with the Transpose Engine enabled (+TP), with the Transpose Engine disabled (-TP) and for the A100. The symbols we use have the following meaning. LC/SC, LS/SS: Large/Small Channel, Spatial Conv. F1: Filter size = 1 Conv. DW: Conv. backprop for weights. BP: Bad placement caused by tiling. LM/SM, LN/SN, LK/SK: Large/Small M, N, K matmul. TP: Transposed matmuls for backprop. EW: Elementwise operations. **We identify the fundamental application behaviors that are key to achieving high performance and efficiency. The identification and explanation of these behaviors are a contribution that is architecture agnostic and to our knowledge, the most comprehensive such analysis.**

1. Large Channel Convolution. convolutions have a large channel LC dimensions, making inner matrix multiplies amenable to both Galileo's SIMD unit and A100's TensorCore. Often channel

Op	AMI	G7	A100	Comments
Resnet50 (All 2D Conv.)				
I 14^2 256->256 f=3 s1	2656	82%	14%	SS LC
I 56^2 64->64 f=3 s1	1139	90%	12%	LS SC
I 28^2 128->128 f=3 s1	2110	82%	16%	SS
I 14^2 256->1024 f=1 s1	1544	93%	8%	F1 LC A−
I 14^2 1024->256 f=1 s1	473	72%	9%	SS BP
T 14^2 256->256 f=3 s1	1264	25%	15%	DW SS
T 56^2 64->64 f=3 s1	430	61%	11%	DW LS SC
T 28^2 128->128 f=3 s1	826	65%	14%	DW LS SC
T 14^2 256->1024 f=1 s1	288	34%	8%	DW F1 LC
SSD-Resnet34 (All 2D Conv.)				
I 150^2 256->256 f=3 s1	4579	94%	53%	LS LC GP
I 150^2 128->128 f=3 s1	2297	97%	37%	LS LC
I 300^2 64->64 f=3 s1	1152	99%	26%	LS
I 150^2 128->256 f=3 s1	4579	98%	43%	LS LC
I 150^2 256->512 f=3 s2	2275	77%	45%	S2
T 38^2 256->256 f=3 s1	1646	53%	37%	DW SS LC
T 38^2 128->128 f=3 s1	843	62%	20%	DW SS LC
T 75^2 64->64 f=3 s1	431	69%	14%	DW SS LC
UNet (All 3D Conv.)				
I 32^3 32->32 f=3 s1	1725	49%	22%	BP LS SC
I 32^3 64->32 f=3 s1	1725	99%	15%	BP LS SC
I 32^3 64->64 f=3 s1	3445	99%	28%	BP LS SC
I 32^3 128->64 f=3 s1	3445	99%	35%	BP LS LC
I 32^3 128->128 f=3 s1	6867	99%	53%	BP LS LC
T 32^3 32->32 f=3 s1	647	27%	16%	DW LS SC
T 32^3 64->32 f=3 s1	863	39%	19%	DW LS SC
T 32^3 64->64 f=3 s1	1294	57%	35%	DW LS SC
T 32^3 128->64 f=3 s1	1724	71%	43%	DW LS LC
BERT-Large				
I Fc(2848x1024x1024)	1506	87%	32%	LM LN LK
I Fc(2848x4096x1024)	3360	91%	93%	LM LN LK
I Fc(2848x1024x4096)	1506	90%	31%	LM LN LK
I Mm(178x178x64)	178	27%	8%	LL SM SN
I Mm(178x64x178)	94	14%	6%	LL SM SN
T Fc(4064x1024x1024)	682	83%	95%	TP LM LN LK
T Fc(4064x4096x1024)	1023	83%	47%	TP LM LN LK
T Fc(4064x1024x4096)	1023	83%	93%	TP LM LN LK
T Mm(254x254x64)	64	2%	9%	TP LL SM SN
T Mm(254x64x254)	64	5%	17%	TP LL SM SN
RNN-T				
I Lstm(512x4096x2048)	241	65%	22%	EW LN LK
I Lstm(512x4096x3072)	164	50%	24%	EW LN LK
I Lstm(512x4096x1264)	377	91%	16%	EW LN LK
I Lstm(512x1280x640)	213	65%	6%	EW SK
I Fc(512x1344x512)	371	71%	11%	EW BP
T Lstm(512x4096x2048)	351	41%	19%	TP EW LN LK
T Lstm(512x4096x3072)	241	41%	23%	TP EW LN LK
T Lstm(512x4096x1264)	540	36%	16%	TP EW LN LK
T Lstm(512x1280x640)	295	47%	5%	TP EW SK
T Fc(512x1344x512)	323	42%	7%	TP EW

Table 3.7: Analysis of Top 3-5 layers for each app. by op count

dimensions are large enough that Galileo's transpose engine becomes unnecessary for performance. When spatial dimensions are small SS, additional parallelism is extracted from output channels, putting more pressure on the transpose engine. In this case, we also observe that the A100 suffers in utilization; likely due to the fact the inner tile dimensions end up not filling the relatively large TensorCore.

2. Large Spatial Convolution. convolutions have high parallelism from splitting work on spatial dimensions LS, as exemplified by UNET's large spatial convolutions. We can also see the A100 appears to require both large spatial and channel dimensions to extract high utilization – both L11 and L21 have a large amount of spatial parallelism available, but L21 has much smaller channel count, which is likely why A100's performance suffers.

3. Unit-Filter Convolution. convolutions with one filter pixel F1 degenerate into a large matrix multiply (M = spatial dimension, N = output channel, K = input channel). In this case, Galileo is able to perform quite well even without the transpose engine since typically channel dimensions are large. When M » N, we find that communication becomes the bottleneck since arithmetic intensity drops.

4. Convolution Backpropagation. Backpropagation for convolution is difficult for two reasons: 1) matrix multiply layouts are transposed, altering minimum tiling requirements, and 2) backpropagation for weights DW cannot parallelize on spatial dimensions. With this parallelism gone, Galileo relies on the channel dimensions, and when this is insufficient, partial gradients are computed over the batch dimension and summed together with in-LLC reduction as discussed in Section 3.4. It is likely that A100 suffers from a similar problem.

5. Tiling Effects on Placement. In some cases BP, we observe a decrease in efficiency when using the transpose engine in Galileo. In all cases we analyzed, this was not due to under-utilization of the SIMD engine, but instead the change in tiling factors as a result of using the transpose engine caused work placement to produce worse communication patterns. This is not a problem for Galileo since we can either naively just switch off the transpose engine, or fine-tune the placement by adjusting tiling factors.

6. Large Matrix Multiplies. The top 3 layers for BERT are matrix multiplies with large M LM, large N LN, large K LK. These kinds of shapes are easy for both SIMD and TensorCore, so it's not surprising that both A100 and Galileo perform well. It does appear the A100 requires even larger

Behav.	Simba	EyerissV2	MAGNet	SIGMA
1,2,3	Med-Hi	Med-Hi	Hi	Hi
4	Unsup.	Unsup.	Unsup.	Lo
5	Var.	Var.	Var.	Var.
6	Hi	Hi	Hi	Hi
7	Lo	Hi	Lo	Med-Hi
8	Med	Unsup.	Unsup.	Med

Behavior #5 depends on shape, the architecture, and mapping strategy and so is difficult to say how tiling factors impact each design.

Table 3.8: Analysis of related academic DL accelerators on basis of their efficiency when observed behaviors are present.

dimensions than L31 (for example) to achieve peak throughput (L32, L33). Additionally, similar to convolution, backward passes for matrix multiply are also matmuls but with transposed inputs TP.

7. Odd-shaped Batch Matrix Multiplies. BERT's self attention layers employ batch matrix multiplies with large outer L dimension LL and relatively small M SM and small N SN dimensions. For these matrix multiplications, 1) the outer batch dimension afford embarrassing parallelism at the expense of interconnect pressure due to drop in arithmetic intensity 2) small and irregular M and N dimensions make it difficult to extract further parallelism meaning Galileo's transpose engine has little effect and 3) layout for this matrix multiply cannot be tuned for inference since neither input is a stored weight.

8. LSTM with low parallelism. The LSTM layer can be thought of as a linear layer over both the input and recurrent state, followed by a relatively complex element-wise operation. Galileo leverages previously discussed techniques for computing the matrix multiples in the linear layers efficiently. Galileo additionally employs careful in-memory layout of tensors to allow for in-LLC elementwise EW operation – meaning the LSTM gates are all computed over local data in a tile's LLC slice, requiring no data movememnt after the linear layers.

Qualitative Analysis of Academic Architectures

To conclude this study, we present a qualitative analysis of academic DL accelerator designs on whether they would perform well for each of these behaviors. Table 3.8 summarizes our findings, which we now break down for each of the four architectures we study. Note that these still suffer from the unaddressed compilability challenge.

Simba [158] only evaluates linear layers but should be able to perform convolutions. Given a vector width of 8, it should perform most convs. in MLPerf well but without additional compute orchestration features, will suffer for some shapes. Being inference-only, training convolution is unsupported. For similar reasons, Simba is likely to perform well for large matmuls, but suffer for the irregular batch matmuls seen in transformers. Simba is likely to perform just okay for LSTMs since it will rely exclusively on batch parallelism, and only its Global PE is able to do near-memory reduction operations. Overall, Simba's dynamic NoC and multicast capability from some units make it amenable to easy data orchestration, in addition, work placement, but its fixed width SIMD units and lack of support for training cause it to fall short on coverage and performance of MLPerf.

EyerissV2 [16] is also an inference engine focusing on convolution. It is likely to support convolutions in MLPerf quite well with its row-stationary dataflow, but does not support training. Since matmuls are a degenerate case of convolution, EyerissV2 should also support matmul layers, likely working well for large matmuls. EyerissV2 would also likely work well for the irregular matmuls in transformers since its PEs are very fine-grained, making compute orchestration much easier at the PE level. EyerissV2 does not support activation functions, though so would not be able to run LSTM ops. EyerissV2's interconnect is statically programmed by software, and if the routing needs of an application exceed hardware resources, software will be unable to route data. Overall, EyerissV2 has good compute orchestration at the expense of complex, software controlled data orchestration. It has no support beyond DNN and CNN inference, so falls short on DL coverage.

MAGNet [177] is an RTL generator which intentionally echews coverage in order to attain the highest possible performance and efficiency for a single application. It employs conventional SIMD execution, meaning it suffers from the data orchestration problems as Simba, but should support nice convolutions and matmuls well, but will likely suffer for irregular batch matmul shapes. It employs a dynamic interconnect so data movement and work placement is a relatively easy lift for software, but without multicast or the ability to "push" data, though, it has to route requests as well as data. Overall, MAGNet was not designed for coverage, and lacks compute and data orchestration techniques to achieve high performance for irregular shapes, as well as the ability to perform training.

SIGMA's FlexDPE [143] is capable of very flexible compute orchestration at the core level. The tradeoff is a relatively complex fan-out/in network to deliver elements to hardware execution units,

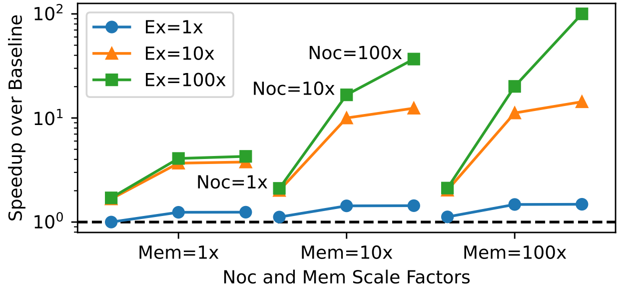


Figure 3.9: Overall performance sensitivity to Memory, Communication network, and Compute engine.

which we find to be over-engineered for DL applications (as exemplified by MLPerf). It has a NoC similar to MAGNet. It will likely perform well for nice matrix multiplies and convolutions as a result. For conv. backprop., it will likely suffer since it cannot perform near-data reductions to reduce communication pressure. It's performance for irregular matmuls will depend on whether the dimensions can fill the relatively large FlexDPE size. SIGMA will likely support LSTM operations about as well as Simba or A100. Overall, SIGMA has the highest coverage of the accelerators we study, but lacks architectural features for optimizing data orchestration, exacerbating its compilability limitation.

Galileo's Roadmap to the next 100X

To examine the scalability of Galileo, we simulate the performance improvement from scaling each of compute, communication, and memory bandwidth by a factor of 1X, 10X, and 100X, separately and together (a total of 27 design points). Figure 3.9 shows these speedups normalized to G2048. The following insights emerge. **i) With additional memory bandwidth alone, at best 25% speedup is possible**. **ii) Conversely, since utilization is already high, improving memory bandwidth and NoC alone or together provides limited speedups. iii) Surprisingly, 10X in bandwidth and compute, with no change to the NoC, provides about 9X speedups, meaning the push-based NoC architecture scales.** Emerging packaging solutions could make this direction realistic to achieve. **iv) Getting speedups beyond 10X also seems surprisingly possible and not limited by application inherent characteristics.** Microarchitecture/architecture techniques that create an effective increase of 100X in memory bandwidth (main memory caches), fast NoCs (photonic like Corona [174])

could help realize these design points in a practical way.

3.7 Conclusion

This chapter identifies the four key problems which must be solved for a new DL accelerator to be successful. Through a fresh perspective of extending established multicore SIMD architecture, we develop Galileo, which solves these four problems in a balanced way, lending to high performance and coverage of modern DL applications. We provide a surprisingly effective approach that outperforms GPUs by large integer factors, with a substantially lower silicon footprint design. The analysis and key behaviors we identified are leveraged by Galileo to achieve this, and have implications for future architectures as well.

4 BENEFITS OF TECHNOLOGY SCALING

Moore's Law and Dennard scaling are dead from a practical standpoint of cost improvements [93], energy efficiency [192], and increasingly density improvements [188, 12, 133]. Analysis that makes estimates on cost is shown in Table 4.1, showing the diminishing costs savings from Moore's Law.

In the previous chapter, we looked at iso-technology designs to distill out hardware mechanisms for performance and coverage of DL. This chapter revisits the architectural claims of specialization to see whether specialization can achieve the solution of "*5nm capability chips*" with 12nm technology. To focus this chapter, we answer this question in a narrow but out-sizely important domain: AI chips. *We seek to answer the question of whether AI chips can be built at 12nm technology using principles of specialization to match or exceed chips made at 5nm or lower.* Specifically, can the techniques of specialization which are generally targeted toward application-specific problems be refined to applied to a programmable AI chip which must target multiple applications/algorithm. We examine whether a Galileo architecture which we discussed in the previous chapter built at 3nm advanced technology would be a factor proportional to Moore's Law area density increase compared to an architecture built at a larger technology node. Using Amdhal's law and knowledge of the three categories of operators, we develop a performance model to compare Galileo G3 built at 12nm with G3′, its 3nm counterpart. Our model is implementation agnostic, allowing us to further estimate

	16nm	**10nm**	**7nm**	**5nm**	**3nm**
Chip Area (mm^2)	125	87	83	85	85
# transistors (BU)	3.3	4.3	6.9	10.5	14.1
Gross die / wafer	478	686	721	707	707
Net die / wafer	359	512	545	530	509
Wafer price ($)	5912	8389	9965	12500	15500
Die cost ($)	16.43	16.37	18.26	23.57	30.45
($) / 1B transistors	4.98	3.81	2.65	2.25	2.16

Table 4.1: Estimates of Transistor costs by wccftech [193]

	Area	Power	Delay
16nm	1.00	1.00	1.00
12nm	0.86	1.10	0.80
7nm	0.34	0.67	0.59
5nm	0.19	0.47	0.41
3nm [133]	0.12	0.47	0.36

Table 4.2: Tech Scaling

this gap for G7 and G5 compared to their 7nm and 5nm counterparts, respectively.

4.1 Definitions and Modeling Equations

Let r_{bw}, r_c, and r_l denote the fraction of operators that are bandwidth heavy, compute heavy, and latency heavy, respectively. Recall we can classify all operations into these categories so for a given application, $r_{bw} + r_c + r_l = 1$. Given a baseline architecture and information about a workload, we model speedup with the following equations:

$$speedup = \frac{1}{\frac{r_{bw}}{s_{bw}} + \frac{r_c}{s_c} + \frac{r_l}{s_l}}$$

Where r_{bw}, r_c, and r_l are as defined previously, and s_{bw}, s_c, and s_l denote the speedup for bandwidth, compute, and latency heavy operators, respectively, over a given baseline architecture. We now discuss how we model these speedups for G3′ over G3.

Modeling Bandwidth-heavy Operator Speedup. Bandwidth heavy operator performance scales proportionally with memory bandwidth. Memory bandwidth can scale independent of technology (HBM2 PHYs exist even at 16nm [153]) , so both G3 and G3′ are capable of the same bandwidth. Thus, we fix $s_{bw} = 1$. Narrower data-types like FP8 and MSFP [24] equally benefit G3 and G3′, by reducing bandwidth needs and compute density.

Modeling Compute-heavy Operator Speedup. We assume compute heavy operator performance would scale proportional to available peak compute. Considering area, peak compute is proportional to the amount of area dedicated to compute. Let $area_{chip}$ denote chip area and $area_{mem}$ denote the area occupied by HBM controllers and other peripherals. Then the compute area scale is

$$a_c = \frac{area_{chip} - area_{mem}}{area_{chip}^{baseline} - area_{mem}^{baseline}} \approx s_c = \frac{FLOPS}{FLOPS^{baseline}}$$

To estimate s_c for G3′ compared to G3, we observe that if we hold total area constant, the factor increase in area devoted to compute follows the area scaling from 12nm to 3nm (6.7×). A further linear compute speedup can come from frequency increase at iso-power (2.165× = delay ratio from Table 4.2 for 12nm to 3nm). Thus we set $s_c = 6.7 \times 2.165 = 14.5$.

Modeling Latency-heavy Operator Speedup. For latency dominated operators, caches and addi-

tional chip area provides speedup – for these operators we model their speedup as $(a_c)^\gamma$ for some power γ applied to the area scale. CPUs are an example of architectures that improve latency-bound applications by exploiting ILP/MLP, where speedup grows as a square-root or cube-root of chip area. We consider $\gamma = 0.25$ and an extreme case of $\gamma = 1$. Effectively, we simply compute $s_l = (a_c)^\gamma$ for G3′ over G3.

Figure 4.1 shows the speedup of G3′ over G3 for two different fractions of r_l (small, where $r_l = 0.1$ and vanishing when $r_l = 0.01$), for a sweep of ratios for the other two regions, and three plots for $\gamma = 0.25$, 0.5, and 1. We call this speedup, the **technology gap** that a 12nm node suffers. For context, BERT has an $r_c = 0.64$ (ratio computed on hypothetical 3nm H100 successor that is 2x faster than H100) and essentially no low-AMI operators. Because of simply Amdahl's Law effects, γ does not have a large impact on final speedup, because r_l is low in DNNs. We also observe that, when $r_c < 0.64$, regardless of r_l, maximum additional speedup from technology scaling from 12nm to 3nm is 2.9×. For heavily compute dominated workloads ($0.50 \leqslant r_c \leqslant 0.64$), this gap is 2.9× for small r_l, and 2.5× for vanishing r_l.

Because the area gap, and hence s_c, and s_l are the largest for 3nm, compared to a specialization gap for 5nm or 7nm, we can conclude that the gap for lower technology nodes will be less. We ran our model for those and the computed values are shown in the table below.

r_c	r_l	**7nm**	**5nm**	**3nm**
0.64	0.10	2.0	2.6	2.9
0.64	0.01	1.8	2.3	2.5
0.50	0.10	1.6	2.0	2.1
0.50	0.01	1.6	1.8	1.9

Takeaway. *The specialization gap – that is the benefit of technology scaling that can be used to create improvements in DL performance is limited to* 2.5× *considering the benefits of 3nm technology, compared to 12nm technology node.*

4.2 Model validation

We validate this model in two ways. We first consider six applications from the MLPerf benchmark suite, where Table 4.4 summarizes the top framework level operations by runtime. From the

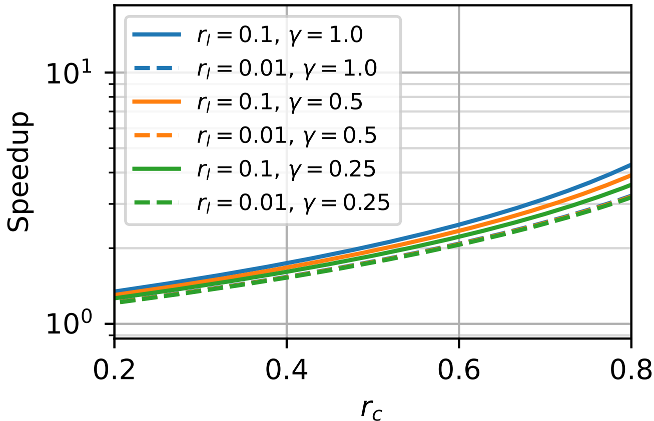


Figure 4.1: G3′ over G3 speedup with various values of r_l and γ.

Table 4.3: Model validation

App / Shape	Measured	Predicted
ResNet50	1.7×	2.0×
SSD-ResNet34	1.8×	2.1×
Bert	2.1×	2.3×
DLRM	1.2×	1.8×
RNN-T	2.2×	2.4×
UNET	2.1×	2.1×
BW Dominated Operators		
16384x128x256	1.6×	1.7×
32768x128x64	1.7×	1.7×
256x8192x4096	1.7×	1.7×
Compute Dominated Operators		
4096x2048x2048	2.5×	2.6×
1024x8192x2048	2.2×	2.6×
Latency Dominated Operators		
256x256x64	1.2×	1.6×
512x256x64	1.4×	1.6×
1024x256x128	1.3×	1.6×

characterization in Table 4.5, we determine values for r_{bw}, r_c (r_l is always zero) for V100 execution. We then use measured values for s_{bw} and s_c, consider peak compute and peak bandwidth for A100 and V100. From these measurements, we can determine the model's predicted speedup for these applications on A100. We compare this speedup to the measured speedup from running actual code and measuring time on the A100. These results are shown in Table 4.3.

To validate using microbenchmarks, we selected a set of GEMM shapes (whose corresponding MLP hyperparamters are also shown) that we know are bandwidth limited (very large M dimension, but small K and small N) or compute limited (very large M, N, and K). For the former r_{bw} =

Op Name	V100	A100
ResNet50 Train		
batch_norm-bwd	24.60%	29.11%
batch_norm	15.36%	21.14%
conv-bwd	28.38%	20.57%
Other	31.66%	29.18%
SSD-ResNet34 Train		
conv-bwd	39.31%	32.40%
batch_norm-bwd	17.26%	21.64%
conv	21.04%	16.73%
Other	22.39%	29.23%
Bert Train		
matmul	62.28%	50.64%
softmax-bwd	6.97%	8.65%
div	4.64%	5.67%
Other	26.11%	35.04%
DLRM Train		
embedding_backward	46.17%	60.76%
matmul	27.29%	13.23%
_index_put_impl	9.09%	7.85%
Other	17.45%	18.17%
RNN-T Train		
matmul	38.46%	32.78%
_cudnn_rnn_backward	30.00%	31.11%
lstm	18.08%	16.05%
Other	13.46%	20.06%
UNET Train		
conv-bwd	49.31%	48.49%
conv	23.03%	22.90%
batch_norm-bwd	11.57%	10.62%
Other	16.09%	17.99%

Table 4.4: Top Operators in MLPerf

	V100			**A100**		
App Name	**EX**	**LAT**	**BW**	**EX**	**LAT**	**BW**
RN50	36%	11%	53%	43%	30%	26%
SSDEN34	55%	4%	40%	58%	28%	14%
Bert	72%	3%	25%	73%	9%	17%
DLRM	26%	65%	8%	14%	84%	2%
RNN-T	80%	5%	15%	63%	29%	8%
UNET	56%	9%	34%	75%	18%	7%

Table 4.5: Fraction of runtime based on operator type

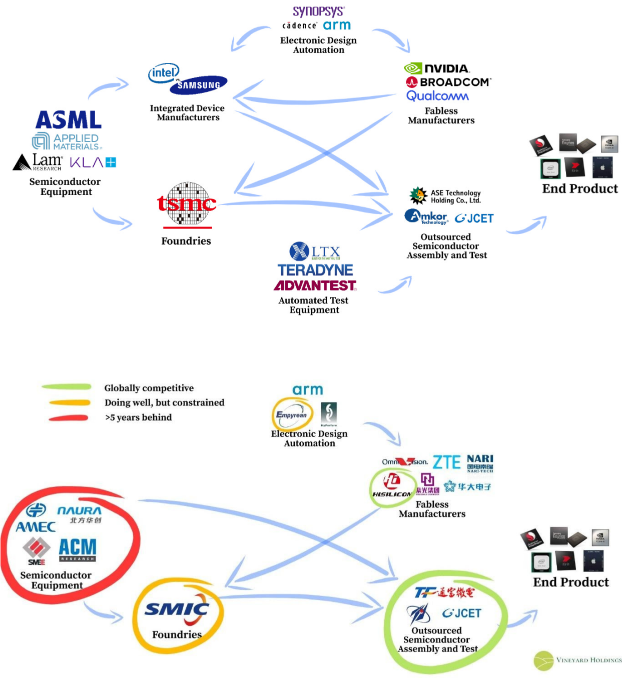


Figure 4.2: World and China Semiconductor supply chain. Source [107].

$1; r_c == r_l = 0$. For the latter $r_c = 1; r_{bw} == r_l = 0$. We also use a set of operators which are latency dominated (for which we run the model with $\gamma = 0.5$). Since $r_l = 0$ for all the other cases, the value of γ doesn't matter. We perform the same steps as for MLPerf: run code on V100, measure the r's, project speedup and measure speedups on A100, and compare measured to projected.

We find that the model is quite accurate for MLPerf, BW-dominated and Compute-dominated microbenchmarks. For the latency dominated microbenchmarks, even with $\gamma = 0.5$, our model is over-predicting speedup. At least for these operators, $\gamma = 0.25$ provides a good curve fit.

4.3 Context and Timing

The bulk of our research work on Galileo and the impacts of technology scaling was completed in Fall 2023. Reexamining our research findings in the time of publication of this dissertation, we draw connections of the tech findings to policy issues surrounding geopolitics and sustainability / environmental impacts.

Somewhat oblivious of the *death* of Moore's Law and Dennard Scaling, a geopolitical war started on October 7th 2022 centered around chips and AI dominance [126, 149], with further refinements in 2023 [135]. Quoting from a New York Times article [130] that analyzes that 100+ page BIS ruling: "the Oct. 7 controls essentially seek to eradicate, root and branch, China's entire ecosystem of advanced technology." Figure 4.2 shows the entire semi-conductor supply chain and China's semi-conductor supply chain. In brief, ASML is prevented from selling EUV equipment to China (preventing them from using outside equipment to build nodes $< 12nm$). EDA companies (the top two being Synopsys and Cadence) are also barred from selling software that could enable technology at 7nm or lower. Finally, fabless vendors are prevented from defacto selling highly capable chips defined by peak performance. In terms of chip sales the ban is defined by peak performance and performance density (by area). In practice, it bans sales of A100, H100, with the the most recent update A800, H800, L40S, and AMD MI250, MI300. Fabs are debarred from manufacturing 7nm chips in the AI space for Chinese companies. Most recently Biren's AI chip [184, 18] was dropped by TSMC as a customer [186] (after building rev0 Silicon that demonstrated parity with H100 [53]).

The hypothesis seems to be that, if China is made stagnant at 12nm technology, the goal of their semiconductor stasis and others' AI chip superiority would be achieved. At the time of writing this dissertation, China's SMIC (a foundry which makes logic node transistors) can manufacture competitive 12nm chips, and their 3D NAND and DRAM capabilities are two generations behind. Recent reports suggested SMIC has 7nm capability [134]. *Despite all this, a corollary to our technology analysis suggests the best case effect of the 7nm and lower semi-conductor blockade on China, would result in a* $2.5\times$ *performance gap to chips built at 12nm for DL.*

From the perspective of environmental implications of computer architecture, recent work has explored a Kaya-like model for evaluating the carbon-footprint of the computer chip industry [31]. Because of the interplay of different social and technological factors, Eekhout suggests three pathways for architects to reduce carbon emissions: 1) reduce demand for more chips, 2) make smaller chips and 3) improve power efficiency. The approach we take to develop Galileo by distilling out the basic primitives needed for performance and coverage is precisely what must be done to address goal #1 and #3 – reducing demand for more chips through programmability while simultaneously improving power efficiency. Our exploration of technology also suggests older technology nodes can be a pathway for "embodied" lower carbon emissions by trading off performance.

5 SPATIAL FUSION

Recall from our longitudinal study, DL is increasing in heterogeneity of behavior – notwithstanding transformer-based foundation language models and the dominance of the self-attention mechanism [176]. All DL frameworks use a graph abstraction to express DL models [83, 15, 99, 88, 42], and this heterogeneity in behavior can be abstracted as graph *shape* differences between applications which includes properties of nodes of the graph and connectivity between nodes. The properties of a node comprise of computation behavior (computationally-intensive or not), memory access behavior (data reuse friendly versus memory bandwidth hungry), and memory data volume. Optimizations are feasible at all levels to increase execution efficiency of DL models: at the algorithm level [13, 194, 33, 196, 194, 137, 14], DL software-stack [138], and hardware [151, 27].

In this chapter, we discuss hybrid (architecture-software codesigned) optimizations for DL which capitalize on this heterogeneity of behavior. Considering GPUs are the most prevalent DL deployment platform, we observe an overlooked optimization opportunity: *when a DL model's graph shape is a mis-fit for the GPU's fundamental execution model, what can be done, short of radical hardware modifications?* Such graphs comprise many nodes and subgraphs whose properties are GPU unfriendly: they run at low compute-intensity and are unable to fully utilize the compute capabilities of the GPU [1]. Examples of DL models with a poor GPU fit include NERF [102] (due to small MLP dimensions lending to low AMI) and Weather prediction [136, 82] patterns (due to irregular memory-access in large gather stages) which use small multi-layer perceptrons (MLP). Figure 5.1, show the cumulative distribution of GPU Streaming Multiprocessor (SM) utilization across five popular DL applications.

The solution to this GPU unfriendliness problem has been observed by others. All prior approaches assume that the underlying GPU maintains execution in a *bulk-synchronous parallel mode* [173] (see Appendix B for GPU background). A graph node's work at the hardware level is transformed into a software context (kernel) that runs across all the processing cores (SMs) on the GPU, with many thousands of hardware threads running *identical* programs, reading inputs from

[1] As an aside, the RNN to Self-Attention evolution is an example of inefficiency leading to GPU-friendly algorithmic development. RNNs run mostly $\mathtt{vector} \times \mathtt{matrix}$ or $\mathtt{vector} \times \mathtt{vector}$ computation - which is extremely poorly supported on GPUs. Self-Attention is $\mathtt{matrix} \times \mathtt{matrix}$ and in particular the evolution from BERT, GPT, GPT3/LLama2/Opt etc *increase* the dimensions of the matrices being multiplied with DL algorithm level changes. They synergistically increase model accuracy and are GPU hardware friendly.

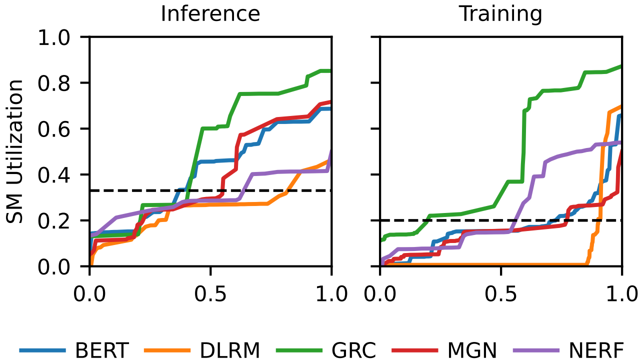


Figure 5.1: SM Utilization CDF for five popular DL applications. For a majority of the applications, greater than 50% of runtime is spent with a utilization less than 33% and 20% for inference and training, respectively.

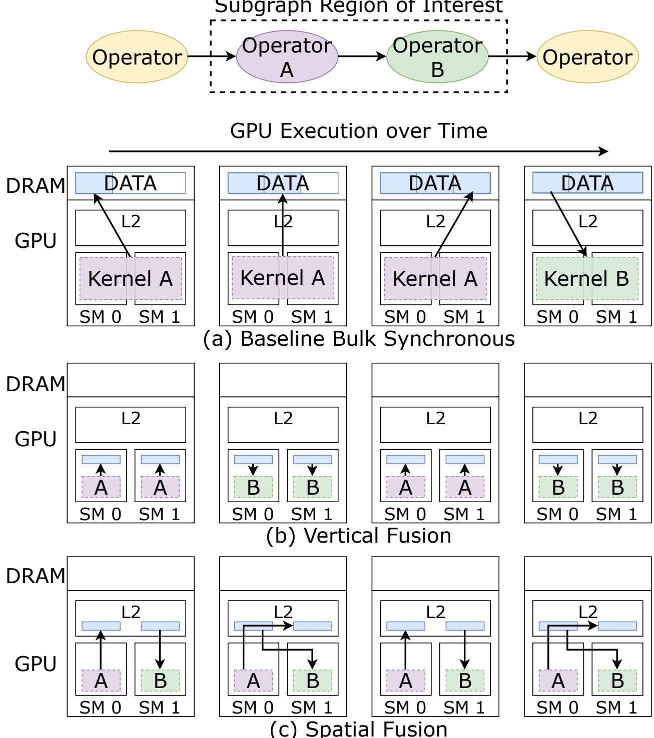


Figure 5.2: Comparison of (a) Baseline bulk synchronous code with (b) Vertical fusion and (v) Spatial fusion.

main memory and writing it's result to main memory. When a node is GPU-unfriendly and unable to fully utilize the GPU, the sharing of intermediates results among nodes using main memory becomes an overhead (Figure 5.2 (a)). Prior work attempts to amortize the unfriendliness by merging nodes, creating a single *fused* kernel with different *code regions* representing work that was

previously different GPU kernels. The main benefit of the fused kernel is the traffic of intermediate results to and from GPU main memory is eliminated and transformed into staging in fast on-chip GPU memory between code regions of a *single kernel*. In the academic literature the state of art includes Welder [160], AStitch [203] and others [55, 199, 202, 69, 198] that perform *vertical fusion* of many individual kernels as shown in Figure 5.2 (b). However, state-of-the-art vertical fusion techniques are quite limited in the types of nodes that they can fuse. They require that the organization of thread groups (CTAs) executing on the SM to be similar for each of the kernels being fused. They also require the fused kernel's SM critical resource requirements (register and shared memory) to be similar and not to exceed what is available. Many graph shapes are infeasible to be fused this way - leading to inefficient execution for many DL models.

This chapter observes that relaxing *bulk-synchronous execution* is the key to improving GPU utilization for such graph shapes (and such a relaxation makes a GPU more general purpose). We enable the GPU to co-execute dependent nodes on **completely different** SMs of a GPU with intermediate data passed directly from one to the other. In doing so, we automatically eliminate the CTA-shape, register-space, and shared-memory pressure problems found in vertical fusion. Figure 5.2 (c) demonstrates this form of spatially pipeline execution, which we call "Spatial Fusion". Three new problems arise: i) how to fuse code in a way that exploits this inter-SM opportunity, lowers to GPU programming APIs (CUDA), and allows simultaneous execution. ii) how to put different pieces of code deterministically in different SMs iii) how to efficiently pass data between these pieces of code, instead of using writes-reads to main memory and realize the full potential of such spatially overlapped heterogeneous execution.

The intellectual insights and contribution of this chapter form three inter-connected pieces: fast queues in GPUs are possible, and when constructed, a radically different GPU execution model of spatial executions becomes feasible, and when applied back to DL graphs their execution can be made substantially more efficient.

This chapter develops Kitsune, an end-to-end compiler that enables spatial fusion on GPUs, achieving high efficiency and performance for DL *inference and training*. We first perform a systematic characterization of graph shapes that highlights the mismatch between graph behavior and GPU bulk-synchronous execution (§5.1). We then develop an end-to-end PyTorch based compiler that lowers PyTorch program into fused sub-graphs when applications contain nodes that are bulk-synchronous unfriendly (§5.2). The compiler includes solutions for load-balancing, sub-graph

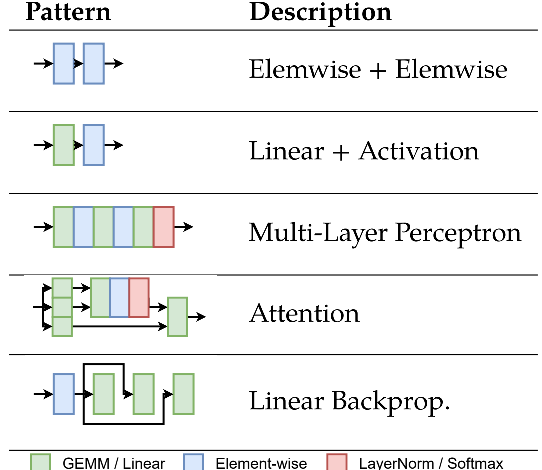
selection, and a clever design for multi-cast/hybrid execution that is necessary to support training. We develop an efficient data queue library (leveraging existing GPU synchronization primitives) that allows near speed-of-light performance when payload sizes are greater than 32KB. Our library is implemented in CUDA C++ and is invoked by our compiler to stream data between nodes in our subgraphs. When executed on GPUs with explicit work-mapping, we show we can run applications at very high hardware utilization (§5.3). Compared to baseline NVIDIA A100 class GPU, Kitsune is 1.3×-2.3× and 1.1×-2.4× better in performance, and reduce DRAM traffic by 41%-98% and 16%-42% for inference and training, respectively, on a set of important DL workloads. Kitsune also provides 1.1×-1.9× and 1.2×-2.0× better performance than state-of-the-art vertical fusion for inference and training, respectively. We find that Kitsune can enable further efficiency in future GPUs designs.

Our contributions are:

- A categorization of graph shapes and patterns that are GPU unfriendly and appearing in many DL models.
- An end-to-end compiler stack that can spatially merge subgraphs and create a rewritten graph that creates fused-nodes that more GPU friendly.
- A low-level queue library that allows inter-node communication on-chip inside a GPU, eliminating vast amounts of low-latency and bandwidth hungry communication.
- Evaluation across several diverse DL models, spanning inference and training, on a SOTA A100 class GPU, showing 1.2× to 4× speedups, with 16%-98% reduction in memory bandwidth (which indirectly serves as a form of power/energy savings).
- We also include a sensitivity study of Kitsune's hardware synergy. In particular, when we increase the inexpensive hardware resources (on-chip compute, on-chip L2 cache bandwidth), while keeping the expensive resources unmodified behaves Kitsune effectively runs better, while baseline execution essentially shows no speedups.
- Finally, we study a software prototype of Kitsune which works on real silicon to demonstrate the ease of software engineering and attainable speedups even without hardware modification.

Application	Year	Use Case
BERT Tiny	2018	Transformer encoder
DLRM	2019	Predicting ad clicks
MeshGraphNets	2020	Mesh based physical simulation
NeRF	2021	View synthesis
GraphCast	2022	Weather forecast prediction

Table 5.1: Description of selected applications and subgraphs for pipelining. Note: operator shapes elided in these diagrams. Different instances may have different shapes for each layer.



Pattern	Description
	Elemwise + Elemwise
	Linear + Activation
	Multi-Layer Perceptron
	Attention
	Linear Backprop.

Table 5.2: Common patterns found across our applications.

Appendices A and B presents and overview of deep learning and GPU hardware and its current execution model which are the basis for our work.

5.1 Program Behavior Motivation & Overview

In this section, we examine more closely the opportunities afforded by DL graph shapes. We base this study on 5 DL applications summarized qualitatively in Table 5.1. We based our selection primarily on the engineering effort required to implement spatial fusion for an application, as well as how well we felt that application would highlight the impact of spatial fusion. Our DL applications include BERT [176] and DLRM [106] from the standard MLPerf benchmark suite [148], and three more recent and highly cited applications – MeshGraphNets [136], NeRF [102] and GraphCast [82]. From a graph shape perspective BERT is representative of ViT [29], Swin-T [90], and mobile-ViT [97] which are all attention/transformer networks and studied in prior work. We first discuss several common patterns which we observe are frequently exhibited in popular DL applications, focusing

on the different considerations for vertical and spatial fusion.

Operator Patterns. Table 5.2 depicts five common graph patterns abstracted from detailed shapes for our applications encompassing both the forward and backward pass. Figure 5.3 depicts, for each application, the opportunities afforded by state-of-art vertical fusion as well as Kitsune spatial fusion for both the forward and backward pass. In both the figures, the we depict three classes of graph nodes: GEMM/Linear-Layer, element-wise, and layernorm/softmax. Element-wise and normalization operations are not computationally intensive and cannot use the TensorCore matrix engines in GPUs running on the SIMT engine, while the GEMM is optimized on modern GPU hardware leaving the SIMT engine essentially idle. The patterns include chains of elementwise operations, a linear layer followed by an activation function, multi-layer perceptrons (MLP), attention, and backpropagation for linear followed by activation.

Vertical Fusion and its Limits. Vertical fusion seeks to improve DL performance by combining multiple DL nodes to elide traffic to main-memory (transparently cached in the L2), and temporally switching between them. Different code-regions in a single "mega kernel" encode the computation and pass values between each other through the GPU's fast shared-memory (shmem) with the register-file space shared among all the fused code regions. The limitations of Vertical fusion arise from the bulk-synchronous execution model and how it exposes and manages resources: i) *capacity limits* of register-file and shared-memory that are compiler managed and sized per kernel constrain the size of intermediates that can be produced, ii) *hardware under-utilization* when a single CTA switches between different code regions with different behaviors, iii) additional storage pressure due to many *live intermediates* needed to accommodate multicast, and iv) conversely, *inability to parallelize reduction* which requires many-to-one communication semantics.

Considering Table 5.2 of real patterns, Vertical Fusion can easily tackle the first row, since shared memory pressure is low. The second and third rows expose the capacity and under-utilization limits. When the hidden dimensions of an MLP exceeds 256, the resultant GEMM's tiles exceed the shared-memory capacity. Furthermore the final layernorm executed on SIMT hardware leaves the TensorCores idle. Attention is difficult to fuse vertically because attention logits can be large (size $= \mathcal{O}((\text{Seq Len})^2)$) so easily exceed shared memory capacity. Welder can consider limited fusion of the $Q * K$ and softmax operations in attention for small sequence length (≈ 128) but simply cannot fuse for larger sequence lengths ($\geqslant 512$) because of this problem. Finally we consider

back-propagation exemplified by back-prop. for a linear layer with an activation function. The backward pass in general consists of a gradient for the activation function (typically elementwise as is the case for ReLU) followed by a multicast to three gradient operations: 1) dL/dX for the input activation gradient, 2) dL/dW for the input weight gradient, and 3) dL/dB for the input bias gradient. The multicast would imply the activation function gradient needs to be kept in shared memory for the computation of all three downstream functions - infeasible for practical shared memory sizes. Additionally, the semantics for dL/dW and dL/dB involve a reduction over the batch dimension and so the full output along the batch dimension needs to seen by one CTA, making these operations limited to essentially 1 CTA, killing performance due to no parallelism, or requiring a separate reduction kernel after partial sums are computed in parallel and written to *memory* (obviating fusion's benefit).

Kitsune Overview

Our insight is that *spatial fusion* – i.e. having *different* operators co-execute across *space* (i.e. Figure 5.2(c), as GPU CTA contexts executing on different cores) rather than temporally switching between executing operators across *time* – solves all these problems, while preserving the benefits of vertical fusion over bulk-synchronous. We introduce a new abstraction to the GPU paradigm - *a fast data queue* that allows inter-CTA communication of data tiles, which can reside in the L2 cache. Similar to vertical fusion, spatial fusion improves performance by avoiding off-chip memory accesses, using our *data queues* residing in on-chip memory to communicate data between operators. The capacity issue is trivially solved by splitting hidden dimensions spatially; the hardware underutilization issue can be solved by assigning different *types* of CTAs to SMs and a trivial modification to the GPUs thread-scheduler; multi-cast and parallel reduction simply become communication patterns of reads/write from our data-queue, requiring the design of a one-to-many and many-to-one queue design which is then used to modify DL graphs.

Revisiting Table 5.2, we can support all of these graphs. For attention we can extract parallelism across the sequence length dimension in the attention computation in addition to the three Q,K,V linear layers. For back-propagation we can construct reduction trees using inter-CTA communication. By using a parallel reduction tree, we can tune number of parallel CTAs to achieve balance of execution for dL/dW and dL/dB. Additionally, the reduction can be performed in otherwise idle

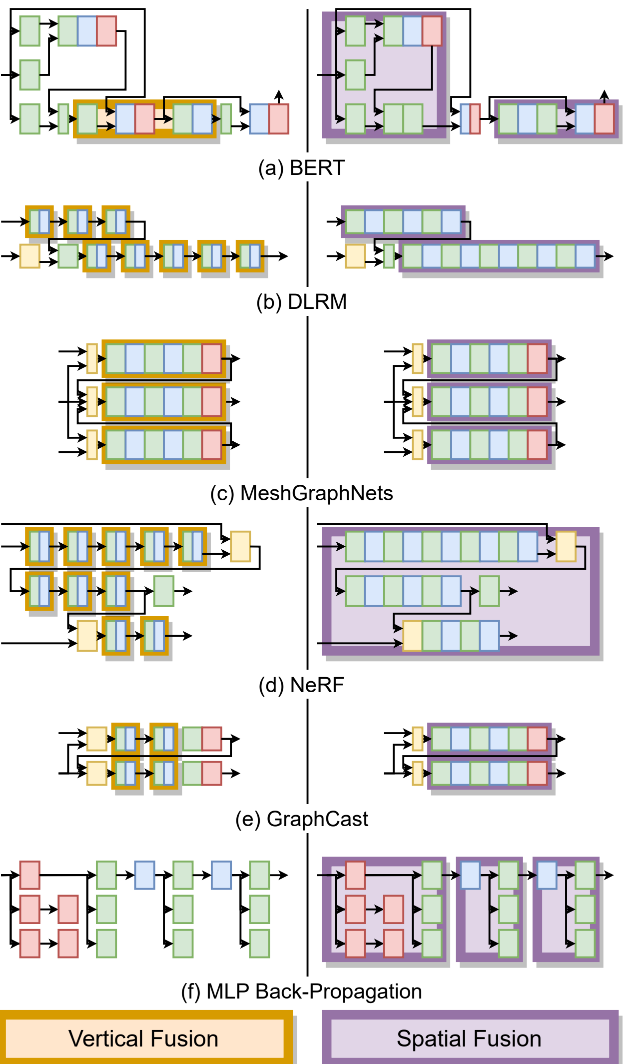


Figure 5.3: Depiction of applications and the fusions we apply.

SIMT cores while the GEMM operations are executed, with results sent to queues to then be passed to the reduce kernel (details in Figure 5.5(e)).

5.2 Kitsune Design

In this section we introduce Kitsune, an end-to-end compiler for deep learning that leverages spatial fusion to improve GPU execution efficiency. We implement Kitsune as a PyTorch [99] compiler backend which we hook into a modeling-based flow for evaluating GPU runtime performance and memory traffic opportunities of spatial fusion. We use PyTorch 2.0's Dynamo interface for

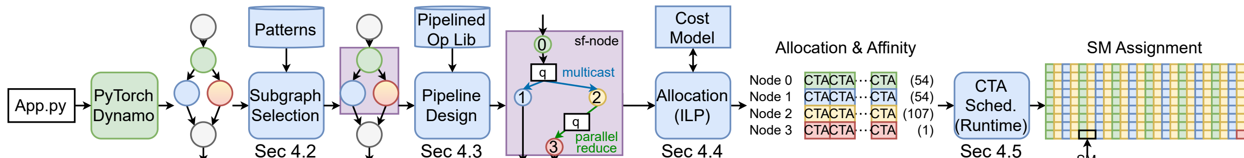


Figure 5.4: Depiction of the Kitsune spatial fusion compiler flow.

extracting application graphs including both the forward pass and backpropagation for training. Our compiler backend consumes these graphs to identify and evaluate spatial fusion opportunities.

To achieve spatial fusion and assess its performance for an application, the following challenges must be solved. We must define an execution model under which spatially fused operations execute. Subgraphs must be selected from the original application graph for fusion. A pipeline must be designed for the subgraph with stages corresponding to operators. The stages must be assigned GPU resources to achieve optimal performance addressing load-balancing and assignment. Finally to enable the execution model, a fast data queue design is needed for inter-CTA data transfer with builtin synchronization. Figure 5.4 depicts how each of these pieces are applied to a PyTorch application.

Execution Model

As a consequence of using a bulk-synchronous execution model, DL frameworks and backends assume one operator is active and occupying the GPU at a time. Further, for bulk-synchronous kernels, it is common that all CTAs are independent – i.e. their order of execution / whether they are co-resident on the GPU does not matter. With Kitsune, we define our “spatial-fusion execution-model” to relax these assumptions, relying on and leveraging dependence and communication between CTAs both within the same kernel and between co-resident kernels. The execution model comprises of CTAs explicitly communicating with each other which triggers and throttles execution speeds. When data is available in a queue, a CTA starts its execution writing results to its producer queue. When there is no data in its queue, it idles. The **first** node of this subgraph reads results from main-memory (essentially outputs of preceding subgraphs or bulk-synchronous code), and the last node writes results to main-memory. In addition to reading from a queue, a CTA is free to read any other values from memory, and similarly can write to main-memory in addition to writes to its producer queue to trigger its successor.

In the formal context of execution models models [85], Kitsune falls under the category of *synchronous dataflow*. Future work can examine further extension like dynamic dataflow. To implement this execution model, each node of our pipeline (sub)graph is expressed as a CUDA Graph (where each kernel / node can have any number of CTAs). Each CTA's code implementation is modified to read/write from a queue, while remaining in an idle state when data is unavailable. Termination of every CTA is determined a-priori by knowledge of the total amount of work/tiles in the DL graph, which is kept track of by each CTA. After it has fired for some number of iterations, it terminates using usual CUDA semantics of calling `return` of the CUDA ABI. Note that even though the execution model is different, it executes exactly using the hardware/software ABI conventions of a CUDA kernel.

Figure 5.5 depicts Kitsune's spatial-fusion execution model with two running examples. (a) shows the application graph for MeshGraphNets, which consists of an encoding layer, followed by 15 such "steps" and a final "decoding" layer. Figure 5.5 (b) shows sub-graph Kitsune selects and the pipeline design: 6 nodes in the DL graph are transformed into 4 nodes, where the only transformation here is epilogue fusion of trivial layers like ReLU to be performed as soon as the heavy computation of the preceding Linear layer is complete. Figure 5.5 (c) and (d) show the allocation and SM assignment for this sub-graph. Figure 5.5 (e) shows a more complex pipeline design transformation of the backward pass for Linear+ReLU which consists of four DL operators that get transformed into 7 pipeline stages. In the remainder of this section we describe the design and implementation of these components.

Subgraph Selection

The subgraph selection problem is defined as taking the full application's DL graph, and labeling each node as running in bulk-synchronous mode or whether it is running in a spatially fused mode. For the nodes in the latter category, we also need to define a cluster of nodes that together form a spatially-fused node (`sf`-node). The output of this phase is labeled graph with `sf`-nodes identified. At the graph level, a spatially-fused group (`sf`-node) of operations must be "contiguous" as defined in [169] – that is, there must be no edge which exits the subgraph with a down stream edge that reenters it. In the most general problem definition subgraph selection influences pipeline design, allocation, assignment and hence performance, requiring an iterative solution. For a practical

solution, we implement a single-pass design that use two rules to exclude a node from a subgraph: nodes that are bulk-sync friendly and nodes that index / gather across all data (gather nodes for embedding for example). With such node exclusions defined, subgraph selection converts to pattern-matching.

Our design and implementation of subgraph selection is heuristic based and uses manual pattern matching. By examining applications properties we identified a handful of node patterns that are candidates for subgraph exposing the vulnerabailities of bulk-synchronous execution and vertical fusion. It is essentially a set or regular expressions that express the patterns seen in Table 5.2. In particular, our implementation operates at the topological order which linearizes the graph into a list in PyTorch Dynamo (which is deterministic). In practice, additional regular expressions to express different orderings for the topological order are easy. A more formal automata or regular expression that captures all possible linearization of a subgraph is beyond the scope of this work.

We leverage PyTorch's Dynamo to extract whole operator graphs of the forward and backward passes for an application. We then created a library of patterns that expresses patterns that are candidates for subgraphs. We implement a pattern-matching algorithm for then selecting subgraphs from the original application graph for spatial fusion. This approach searches for user-specified chains of operators in a topological order. Adding new patterns is a trivial task of adding to our pattern library. Figure 5.3 depicts selected operator groups for spatial fusion across all our applications.

Pipeline Design

The pipeline design problem comprises of inserting queues between nodes of an `sf`-node, and if the work done between two nodes is trivially fusable, fuse them using epilogue fusion (or vertical fusion). The output is a transformed graph which includes one or more queue nodes added, which can then be lowered to CUDA code during code-generation.

Conceptually what this means is taking the original set of operations in the graph and either combining or splitting them to map to pipeline stages that are realized by pipeline-enabled CUDA kernels. For simple patterns like the first three rows of Table 5.2, the decision is trivial - and involves insertion of queue nodes between nodes of an `sf`-node. For more complex patterns like attention and back-propagation, we implement a parallel reduce and the queues are inserted denoting their

Algorithm 1: Algorithm for pipeline design

```
1  for n in Graph do
2      if IsReduction(n) then
3          fanin, final ← SplitReduction(n)
4          Graph.replace([n], [fanin, final])
5          n ← final
6      end
7      if IsIntermediate(n) then
8          q ← CreateQueue(n)
9          for c in Dependents(n) do
10             c.producer ← q
11         end
12         n.dependents ← [q]
13     end
14 end
```

multicast semantics. Figure 5.5 (e) shows the pipelined graph (right) starting from the subgraph (left).

In terms of implementation this involves three steps. The graph rewrite algorithm is shown in Algorithm 1. In terms of code-generation, the queue implementation is discussed later in §0. The third step is to take CUDA kernels and transform them to read/write from queues, instead of from global memory. This last step also includes the process of working on tiles, since a queue's payload needs to be limited. In all cases, the notion of tiling already exists or is trivially doable; for GEMMs the code is already written to work on tiles of inputs and outputs. Completely automating this step for arbitrary code is likely infeasible and involves all the challenges of aliasing analysis etc. For Kitsune, we performed this step manually - it took about 8 person-hours or less for each kernel, with the source-code lines changed ranging from 10 to 40. The limitation this adds to Kitsune is that it is not completely turn-key for new operators not previously seen by the compiler, requiring very modest library modifications of the underlying *new* DL operator. In practice, library developers from chip vendors like NVIDIA and AMD can incorporate such a flow trivially into their kernel development process.

Load Balance

The load balance problem is the penultimate step of code generation: it involves the logical allocation of # CTAs to each node in an sf-node. Its output is simply an allocation as shown in Figure 5.5(c).

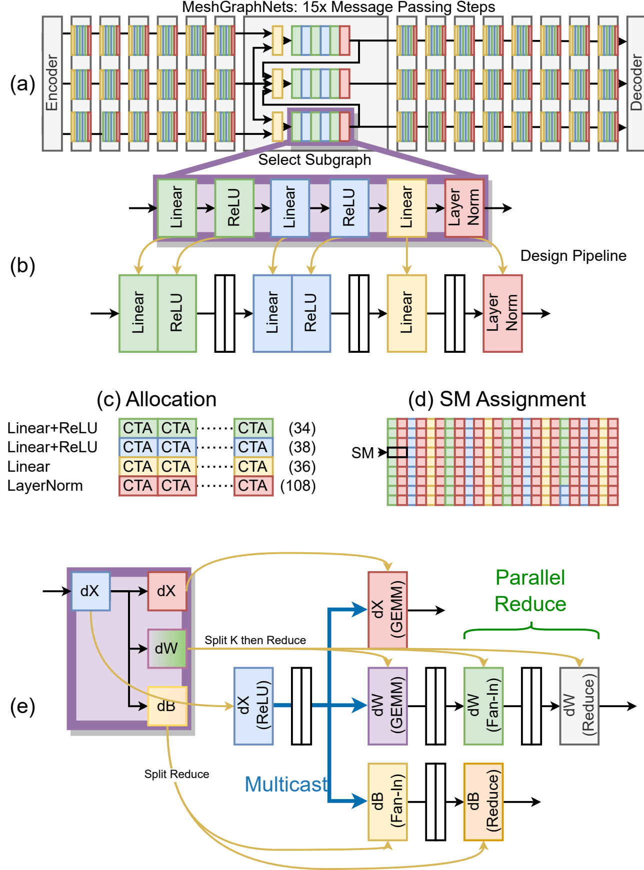


Figure 5.5: Running example of two subgraphs selected from (a) MeshGraphNets. (b) shows an MLP. (c) and (d) show the allocation and assignment. (e) shows a subgraph from the backward pass.

This needs to be done cognizant of overlapped execution of dissimilar CTAs on the same SM.

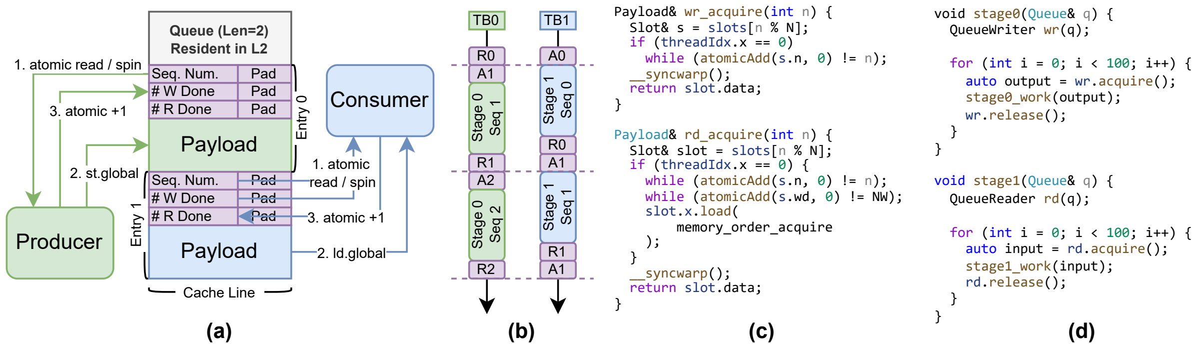
We use a zero-latency performance model to estimate the throughput of a spatially-fused subgraph based on an allocation of CTAs to each stage. We then formulate the allocation problem as an integer linear program (ILP) which can be used with standard solvers to produce an optimal assignment which maximizes throughput of the subgraph. We augment our ILP formulation to enable over-subscribing CTAs onto SMs to enable overlapping dissimilar behavior – specifically, we consider two classes of operations: SIMT-heavy, and TensorCore-heavy, and in "overlap mode" assume an SM can simultaneously execute one of each with no performance degradation. We discuss in Sec 0 low level details of how this overlap can be enabled on modern GPU hardware.

Algorithm 2 shows our ILP formulation. We model throughput as the minimum throughput pipeline stage in the fused subgraph additionally constrained by memory bandwidth and aggregate

L2 bandwidth based on analytic evaluation of the total number of bytes read/written from DRAM (DRAM Bytes), and L2 (L2 Bytes). For the ith of n operators in a spatially-fused subgraph, we estimate the performance in "spatially-fused mode" by combining a measured bulk-synchronous throughput (t_i) with an estimate of how the performance will scale (speedup or slowdown) based on how many CTAs it is assigned (r_i =ResourceScale(a_i)) and an estimate of speedup afforded by operating in spatial mode where some number of its operands are now resident in on-chip storage instead of DRAM ((s_i =Speedup(a_i)). In practice, we define Speedup(a_i) to be $1/u$ where u is the maximum resource utilization of the SIMT or TensorCore pipelines. When overlap operations is disabled, we simply constrain the assignments to sum to the number of SMs on the GPU. When overlap is enabled, we allow the number of SIMT and Tensor stages to independently be assigned SMs. Conceptually we are separating out the SIMT and TensorCore components of the SM into two logical resources that can be assigned. In practical deployment terms, we require either a two-pass compiler, run-time optimization pass, or a dictionary of kernel characteristics to get u_i to guide the ILP which can be implemented as a component of the framework runtime. Since DL models generally run in a curated environment (TensorRT for example), any of those approaches are practical, and don't introduce any application slowdowns.

Algorithm 2: ILP formulation for load balancing.

$$
\begin{array}{ll}
\text{maximize} & \mathrm{thrpt} \\
\text{subject to} & \mathrm{thrpt} < r_i * s_i * t_i \quad (i = 1, \dots, n) \\
 & \mathrm{thrpt} * (\text{DRAM Bytes}) < \text{DRAM}_{peak} \\
 & \mathrm{thrpt} * (\text{L2 Bytes}) < \text{L2}_{peak} \\
 & t_i = \text{Bulk-Sync Thrpt. for Op } i \\
 & r_i = \text{ResourceScale}(a_i) \\
 & s_i = \text{Speedup}(a_i) \\
 & 1 \leqslant a_i \leqslant \#\,\text{SMs} \\
\hline
\text{(No Overlap)} & \sum_{i=1}^{n} a_i = \#\,\text{SMs} \\
\hline
\text{(w/ Overlap)} & \sum_{i=1}^{n} \text{IsSimt}_i * a_i = \#\,\text{SMs} \\
 & \sum_{i=1}^{n} \text{IsTensor}_i * a_i = \#\,\text{SMs}
\end{array}
$$




```
Payload& wr_acquire(int n) {
  Slot& s = slots[n % N];
  if (threadIdx.x == 0)
    while (atomicAdd(s.n, 0) != n);
  __syncwarp();
  return slot.data;
}

Payload& rd_acquire(int n) {
  Slot& slot = slots[n % N];
  if (threadIdx.x == 0) {
    while (atomicAdd(s.n, 0) != n);
    while (atomicAdd(s.wd, 0) != NW);
    slot.x.load(
          memory_order_acquire
    );
  }
  __syncwarp();
  return slot.data;
}
```

```
void stage0(Queue& q) {
  QueueWriter wr(q);

  for (int i = 0; i < 100; i++) {
    auto output = wr.acquire();
    stage0_work(output);
    wr.release();
  }
}

void stage1(Queue& q) {
  QueueReader rd(q);

  for (int i = 0; i < 100; i++) {
    auto input = rd.acquire();
    stage1_work(input);
    rd.release();
  }
}
```

Figure 5.6: Queue design. Note: release routines are not shown for space reasons. They involve simple atomicAdd calls to update synchronization metadata and a CTA barrier with __syncthreads().

Assigning Spatially Fused Operators

We have to solve two inter-connected problems: how to specify an sf-node as an indivisible piece of work to avoid deadlock, livelock, and starvation. Second, to overcome under-utilization, how to assign dissimilar CTAs to the same SM.

CUDA Graphs is a host-level API for specifying control dependencies (i.e. launch ordering) of kernels. Kitsune creates a CUDA graph that comprises of all the kernels of an sf-node, with no dependencies between them at this control level, ensuring the CTAs can all start at any time. To ensure that dissimilar CTAs achieve overlapped execution, we augment the API to allow specifying meta-information for each kernel identifying it as having affinity for SIMT or TC. This information is then used by the hardware's Giga-Thread Scheduler to assign CTAs to SMs such that co-scheduled CTAs have different affinities. The existing scheduler implements load-balanced row-major rasterization of the 2D CTA grid [172, 120, 124]. The modification is straight-forward: first schedule all SIMT-affinity CTAs in row-major rasterization, and then schedule the rest.

In our results (Sec 5.3), we discuss how this assignment is forgiving even to arbitrary pairings, with register-space and shared-memory being the critical resources (which the CTA-scheduler is already aware of). Since modern GPUs have crossbar interconnect, further optimizations for physical affinity between producer and consumer does not help.

Producer consumer communication

We use GPU atomics to design a synchronized, ring buffer queue for passing data between CTAs. Queues are pinned in the L2 cache using CUDA API functions [120] (Fig 5.6(a)). Each entry contains

metadata protected by atomic accesses. Figure 5.6 shows (a) a diagram of our queue design (with two entries for double-buffering), (b) a timeline of producer-consumer operations, (c) stylized code implementing the queue, and (d) application-level usage. Two CTAs communicating (Fig 5.6(b)) "acquire" and "release" entries, achieving ordering via sequence numbers. The producer and consumer acquire entries (`wr_acquire` and `rd_acquire` in Fig 5.6(c)) by spinning on metadata variables until an entry is freed for use. `acquire` and `release` are exposed as an API which handle sequencing automatically. Typically, only one CTA is spinning on a variable at a given time – meaning our queue design results in very low contention.

Our queue is implemented as a library with two API functions: `acquire` and `release`. This allows for easy software integration, introducing minimal overhead exploiting the modern GPU's sophisticated warp-scheduler. Queue code is wrapped with `if` threadid==0, ensuring only one thread in a CTA does any of the queue management. To avoid data-races, "release" operations require a CTA-level barrier. Figure 5.6(d) shows how it can be used intuitively by a CUDA programmer or inserted by a source-to-source compiler into existing CUDA kernels. Synchronization variables are all padded to the size of a cache line to avoid false-sharing.

5.3 Evaluation

We now examine the effectiveness of Kitsune across our applications and GPU models. We guide our evaluation with the following questions: i) How well does Kitsune support composing arbitrary operations across DL applications? ii) What is the end-to-end performance of applications running with Kitsune and what are the reasons for variation across applications and modes of operation? iii) What is the sensitivity of our performance and gains to machine parameters including on-chip compute (number of SMs), off-chip DRAM bandwidth, and L2 and crossbar bandwidth?

Methodology

Our evaluation is based on running our 5 applications in a validated GPU simulator which takes as input the compiled versions of our applications. We built our compiler and a queue library (characterized and run on silicon (§5.3). We need CTA assignment control of the of the Giga-Thread Scheduler to allow overlap (§5.2) to evaluate Kitsune's application/DL performance. Therefore,

App	# Ops	Fusion Coverage		Traffic Red.	
		Vertical	Kitsune	Vert.	Kitsu.
Inference					
BERT	52	12 (23%)	34 (65%)	9%	77%
DLRM	21	17 (81%)	17 (81%)	21%	41%
GRC	35	21 (60%)	29 (83%)	22%	58%
MGN	51	36 (71%)	41 (80%)	51%	52%
NERF	24	18 (75%)	24 (100%)	40%	98%
Training					
BERT	134	12 (9%)	58 (43%)	3%	19%
DLRM	59	18 (31%)	46 (78%)	6%	16%
GRC	101	20 (20%)	76 (75%)	8%	42%
MGN	148	36 (24%)	108 (73%)	16%	37%
NERF	69	18 (26%)	56 (81%)	13%	40%

Table 5.3: Summary of fusions and traffic reductions.

we evaluate Kitsune using a modified version of NVIDIA's NVArchSim (NVAS), a hybrid trace- and execution-driven GPU simulator [179] that has been validated against NVIDIA's Ampere GPU. The simulator also allows us to study sensitivity to individual hardware features, instead of being restricted to particular SKUs.

We first describe the quantitative scope of the opportunity that spatial fusion provides. We then discuss inference and training separately. We follow that up with a hardware sensitivity study, where we modify the hardware to resemble a H100 (2x-2x-2x) by doubling number of SMs, L2 capacity & bandwidth, and memory bandwidth; and an intermediate point (2x-2x-1x) where bandwidth is not doubled - to tease apart the sensitivity to expensive hardware features like DRAM bandwidth. For our baseline hardware configuration, we also report results for the best result from state-of-art Vertical Fusion, which combines the mechanisms of AStitch [203] and Welder [160]. For the Kitsune results we present results for just the subgraphs of the applications and the speedup for the entire application. Additionally we a KitsuneNaive configuration which doesn't exploit the overlap opportunity by exposing affinity information described in §5.2.

DL Application Operator Coverage

Table 5.3 provides a characterization of the applications at the DL operator level denoting the number of operators that are candidates for spatial fusion. The top half of rows are for inference and the bottom half are training. Note this data is for operator count (we discuss time below). Typically, >65% of operators are candidates for spatial fusion, with higher coverage for inference.

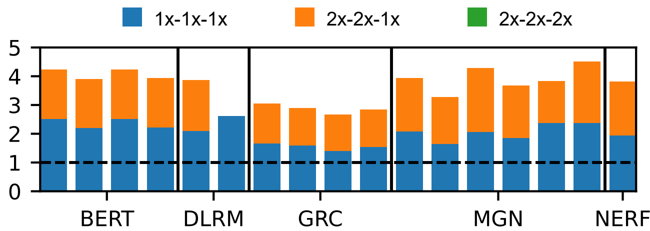


Figure 5.7: Inference subgraph speedups including sensitivity to hardware resources.

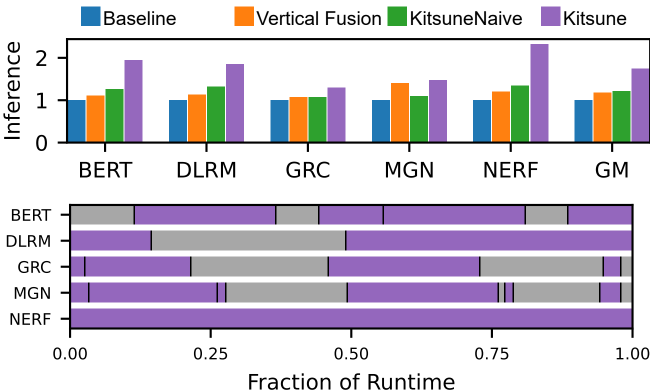


Figure 5.8: Inference End-to-end Speedup over Bulk-Sync.

We note that Vertical fusion covers only the forward pass operators for training[2] and it's coverage is typically lower.

The last two columns show memory traffic savings both for Vertical Fusion and Kitsune. Memory traffic savings is a useful property in itself, as it results in energy/power savings (by downclocking the memory frequency to sustain the lower bandwidth needs). O'Connor et al. [129] and others [70, 39] have argued that GPUs are becoming memory power limited.

Inference Performance

Figure 5.7 shows the speedup Kitsune provides for each of the subgraphs in each of the applications. Figure 5.8's timeline show the time contributed to overall execution by each of the subgraphs, and in grey we show the time the application spends in kernels/operators that run in bulk-synchronous mode. Figure 5.8's bar-charts show full application speedup.

Overall, sub-graphs speedup range from 1.4×-2.6× across the applications, with a geomean of

[2]We note that none of the academic work or TensorRT have demonstrated execution of training yet - our results are thus optimistic for vertical fusion.

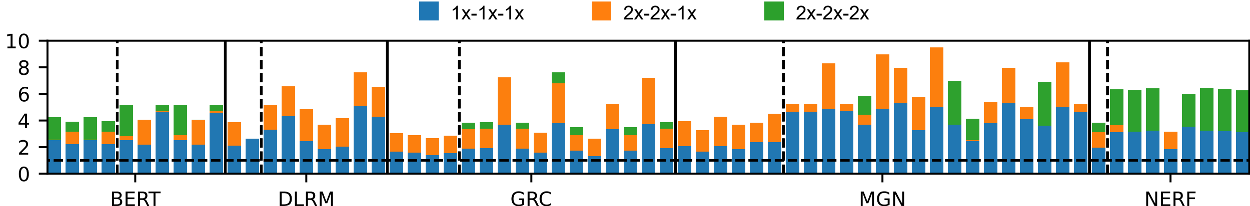


Figure 5.9: Training subgraph speedups including sensitivity. Dashed lines separate forward and backward passes.

2×. The least speedups are for the subgraphs of GRC because they are already achieving >50% of machine peak compute and so do not benefit a lot from operating in spatial mode. NeRF is an example where large speedup is achieved (2.3×), highlighting many of Kitsune's benefits: all the nodes of NeRF's forward pass are spatially fused, allowing most layers to pull intermediates from a data queue instead of main-memory; and the concat operations are free to occupy the SIMT units of the SMs while the GEMMs use the TensorCores. Due to the intermediate sizes, vertical fusion cannot fuse NeRF's linear layers[3].

Looking at hardware sensitivity, we see that Kitsune nearly linearly scales in performance for all the subgraphs with doubling of compute+L2 resources. For inference, since memory traffic is drastically reduced increasing memory bandwdith is inconsequential. Conversely implying that doubling of performance is possible without adding any memory bandwidth which is becoming one of the highest $ and power [129, 70, 39, 41] cost of GPU scaling. We also note these sub-graphs' kernels are memory bandwidth-bound in bulk-synchronous mode; the baseline shows no speedup with the 2x-2x-1x configuration.

When looking at full application performance, we observe two phenomenon: large portions of time are spent in the sub-graphs (typically $> 50\%$), and a single application has a few such subgraphs (the black lines in Figure 5.8 indicate end of a sub-graph). In terms of end-to-end performance, we generally see a 2× speedup. GRC and MGN show the least speedups because their subgraph speedup is modest (1.4×), while their sub-graph coverage in time is $< 50\%$.

Takeaway: *We find Kitsune provides substantial performance opportunity for DL inference with this generally scaling with number of fused operations. For the fusions we observe, DRAM bandwidth is not a bottleneck, suggesting higher performance could be possible without increasing bandwidth.*

[3]Recall our configuration of NeRF is based on the original paper which uses hidden dimension of 256.

Training Performance

Figures 5.10 and 5.9 show the corresponding results for training, with training broken down further in terms of the forward and backward pass. The forward pass is similar to inference, with the added issue of intermediate activations being stored to main-memory for computing gradients. The backward pass then uses these to compute gradients for trained parameters.

Impressively the trends are similar showing Kitsune's ability to span inference and training. As expected, the doubling of compute+L2 alone provides some speedup, while the double of memory bandwidth provides further benefit (because of generally higher memory traffic for training). Most of NERF's and some of BERT's back-propagation sub-graphs are DRAM bandwidth-bound and so increasing SM count and L2 bandwidth does not provide much speedup without additional DRAM bandwidth.

Considering end-to-end speedup, we see two trends. As expected, the backward pass takes about 2× the time of the forward pass. Less fractional time of the backward pass is spent in spatial mode, especially for BERT and DLRM. For BERT, the backward pass for attention is not spatially fused, so only the linear layer backward passes in the feed forward network and Q, K, V layers are fused. For DLRM, the backward pass for the feature interaction which is not spatially fused takes substantial runtime, causing an Amdahl's law effect on training back-backpropagation. End-to-end speedups range from only 1.3× to as high as 2.2×.

Takeaway: *Kitsune still enables performance gains for Deep Learning training, with lower improvements due to smaller fusions in the backward pass compared to forward. Because of Kitsune's ability to parallelize reduction operations, training can scale more easily to large on-chip resources and DRAM bandwidth compared to the parallelism-limited bulk-synchronous baseline.*

Comparing to Vertical Fusion

Due to the limitations outlined in §5.1, effectiveness of Vertical Fusion is substantially lower than Kitsune for inference, with MGN showing the best speedup (1.45×) with geo-mean 1.17× (Figure 5.8). Since it only applies for the forward pass, training speedups are even lower (Figure 5.10). Related works like Welder, for inference, have reached similar findings: when applied to production settings of running with TensorCore and meaningful batch-size (like 32 or larger), speedups over

un-optimized PyTorch (worse than our baseline) is 30% or so, with no speedup over TensorRT on Nvidia V100 [160]. Those works target additional scenarios like FP32 based computation (thus eliding our overlap opportunity) and edge-case scenarios like batch-size=1, which are less important in production data-center deployment. Philosophically they target improvements through software in the configuration space where GPUs are inefficient (bs=1, fp32 mode etc). We focus on production scenarios: batched training and inference using TensorCores to address inefficiencies.

CTA Assignment Complexity

We now delve into some details of Kitsune's inner workings. First we look at CTA assignment which is one potential source of software/hardware complexity. To better understand the complexity in achieving efficient CTA scheduling on the GPU for Kitsune spatially fused sub-graphs, we study the space of pairings of SIMT-heavy and Tensor-heavy operators across our applications. Figure 5.11 contains three plots – one for each of three key SM resources: SIMT core, TensorCore, and (SM) Bandwidth utilization. We determine all possible pairings of CTA code an SM could co-execute across our applications and sub-graphs, resulting in 206 unique pairings (for this simulation, we consider a hypothetical 1-SM GPU with bandwidth scaled down). We plot for each possible pair of spatially fused kernels, their combined need for these three resources. Note that greater than 100% sometimes occurs, indicating a pair where slowdown would occur if these two operations are co-scheduled to the same SM. We observe 139 pairs have total resource utilization that can be accommodated by co-execution. CTA behavior oblivious scheduling would thus perform poorly if the CTA-scheduler did not include meta-information to ensure dissimilar CTAs get co-scheduled.

Queue Performance

Kitsune's high performance queue is critical to our execution model. We characterize our queue by measuring SM-SM bandwidth with varying payload sizes for 54 queues (108 CTAs for the 108 SMs of the A100 GPU). We examine queue management overhead by measuring the performance of data transfers with and without synchronizing atomics enabled. Figure 5.12 (top) shows the results of this study. The orange points allow data to race and would not be considered correct execution. With 128-256 KB payloads, aggregate bandwidth reaches 2 TB/s (37 GB/s/queue). Beyond 256

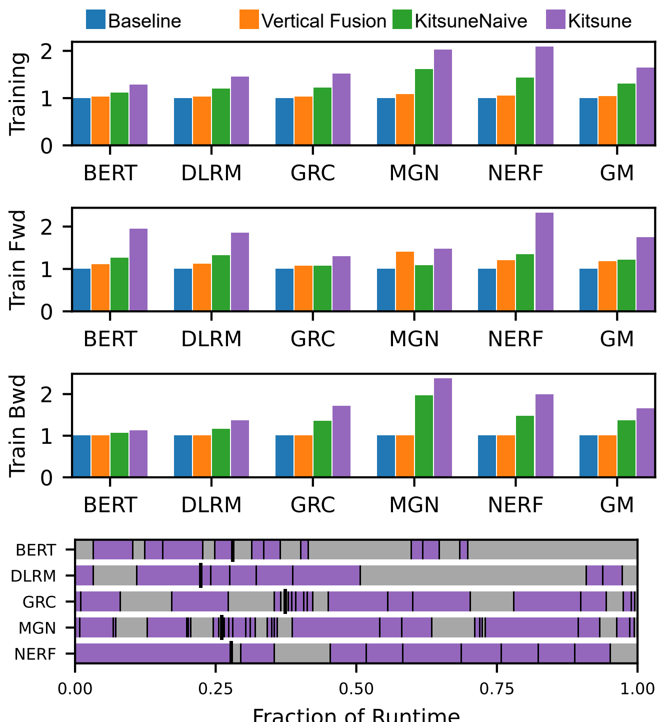


Figure 5.10: Training End-to-end Speedup over Bulk-Sync.

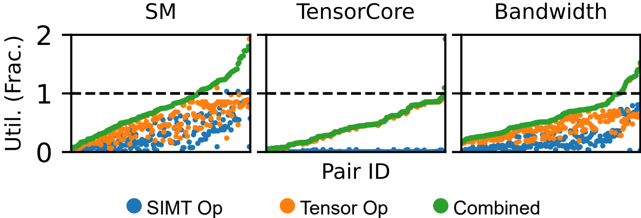


Figure 5.11: All pairings of SIMT- and Tensor-heavy ops onto one SM. We show the individual and combined usage of each resource. Pairings are ordered by increasing combined usage for each resource / pairings are not the same ID across charts.

KB, we see a drop in performance due to queue sizes reaching the L2 capacity, causing accesses to spill out to HBM (Limiting us to 1.5 TB/s for A100). Synchronization overhead is large for small queue sizes: 12× reduction in bandwidth for 1KB payloads. With larger payloads this reduces: synchronization overhead is less than 63% for ⩾64KB payloads. Based on additional measurements, we find for payloads less than 128KB, only 1.6 atomics are executed per spin-loop on average, further demonstrating low contention. *Overall, we find our atomics-based L2 resident queue provides substantial inter-CTA communication bandwidth even in the presence of contention for payloads ranging between 64-256KB.*

The queue's effectiveness hinges on the hardware's atomics performance. The results of an atomicAdd microbenchmark where the # variables and number of contenders is shown in Figure 5.12 (bottom) measured in an A100 GPU. When under no contention, the A100 sustains 100 M atomics / sec / CTA. With double buffering and well-balanced producer/consumer rates, two atomics per warp are required to transfer some data between two communicating CTAs. Considering 32-128 KB payloads and 4 warps / CTA, this implies an upper bound of 385-1541 GB/s *per queue*. This far exceeds L2 and HBM bandwidth (≈61 GB/s per SM), meaning synchronization will not be the bottleneck as we observed above.

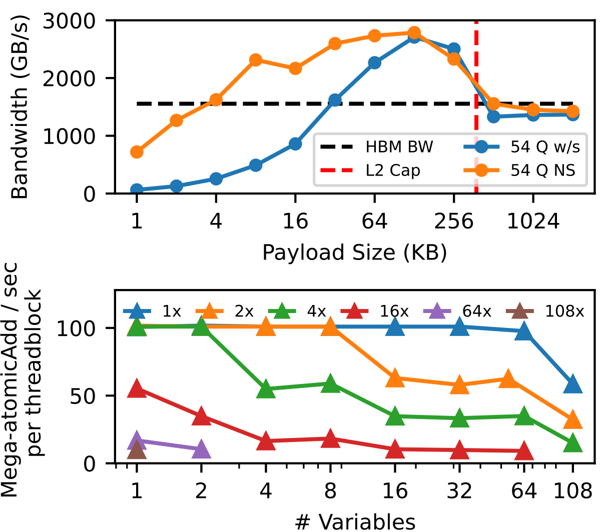


Figure 5.12: Performance of GPU atomics (top) and our synchronized queue (bottom).

5.4 Kitsune Software Prototype

In this section we evaluate a pure-software prototype of Kitsune which we call KitsuneSW. We implement several subgraphs from the forward passes of our selected applications. The coverage our implementation in terms of number of ops and runtime, batch size we select for each application, and analytic traffic reduction is summarized by Table 5.4. The rest of this section discusses methodology, composability of our library and performance of this pure-software implementation.

App.	Batch Size	# Pipe / # Ops		Pipe Lat. / App Lat. (ms)		Subgraph / App Traffic (MB)		Subgraph Compute %
BERT	32 K	10 / 36	(28%)	0.82 / 3.29	(25%)	321 / 691	(46%)	(11%)
DLRM	2 K	16 / 21	(76%)	0.34 / 0.71	(48%)	74 / 159	(46%)	(7%)
MGN	1.3 M	90 / 473	(19%)	13.7 / 30.7	(45%)	864 K / 2.3 M	(37%)	(11%)
NERF	65 K	24 / 24	(100%)	1.64 / 1.64	(100%)	1,395 / 1,408	(99%)	(16%)
GC	40 K	128 / 320	(40%)	137 / 298	(46%)	34 K / 129 K	(27%)	(34%)

Table 5.4: Dataflow coverage of applications in terms of number of operators, time and traffic.

Methodology

Evaluation Platform. We use an NVIDIA A100 40GB PCIe GPU based server (2x AMD EPYC 7282 16-Core motherboard, 512 GB DDR4 DRAM). We additionally measure sustained HBM usage with NVIDIA NSIGHT Compute (NCU) [122], and use nvidia-smi to monitor the power draw of the GPU while running experiments. For full application performance, we compute end-application speedup based on the coverage in Table 5.4 and measured performance for the full application and subgraph performance and dataflow speedup.

Composability Result

We find KitsuneSW to be amenable to fusing many operators. Table 5.4 show total dynamic operations run in dataflow mode on KitsuneSW (column 3) ranges from 10-128. Across the subgraphs, there are 24 unique operator shapes and 18 CUDA kernels used. Implementing dataflow subgraphs – I.e. expressing the dependence semantics, identifying kernel functions and inserting our library calls into original kernels – ranged from fusing 6 to 24 operators and took roughly two person-weeks of work. Considering NeRF: 24 original operators translated to 14 pipeline stages after fusing Linear+ReLU including two concat operations for skip connections. In contrast, approaches such as NVIDIA CUTLASS typically only consider fusing activation functions such as ReLU with things like Linear or Convolution. Fusing complex operations such as LayerNorm is not supported in a straightforward way.

Performance and Traffic Savings

We first study the performance and traffic benefits of Kitsune for our selected subgraphs. We observe these subgraphs are "hard" for GPUs: only 7%-34% of peak compute is achieved (shown

Subgraph	DRAM Utilization		
	Bulk-Sync.	KitsuneSW	Ratio
mgn_mlp	32.5%	10.2%	3.2×
dlrm_bot	5.0%	0.5%	9.9×
dlrm_top	8.1%	1.9%	4.3×
bert_ffn	24.7%	2.4%	10.3×
nerf_all	33.3%	0.9%	39.1×
gc_nodes	33.1%	7.7%	4.3×
gc_edges	33.3%	7.9%	4.2×

Table 5.5: Measured traffic reduction from NVIDIA NCU.

in Table 5.4). Figure 5.13 summarizes, for each subgraph, the measured speedup (orange bars on A100). We note that bulk-synchronous uses state-of-art optimizations provided by PyTorch and CuDNN. Table 5.5 shows the DRAM utilization of bulk-synchronous and KitsuneSW measured by NCU (we measure this for KitsuneSW but expect traffic reduction to apply also to Kitsune). Across the subgraphs we see superior performance to native PyTorch – ranging from 1.8× (gc_edges) to 45× (dlrm_bot), and substantial reduction in DRAM utilization, ranging from 3.2× (mgn_mlp) to 39× (nerf_all).

Discussion of Performance. The largest performance improvements occur for small GEMM shapes (N, K $\approx$ 128 $-$ 256) with improvement dropping off for larger shapes (N, K $>$ 1024). This is because at smaller GEMM sizes, keeping data on-chip helps overcome the relatively low arithmetic intensity of small shapes by reducing the latency to fetch data – and our codesigned queue structures further serve to reduce this latency. For larger GEMM shapes with high data reuse / AMI, neither HBM memory bandwidth nor latency is the bottleneck for performance so Kitsune only sees small performance gains.

Discussion of Traffic. For memory traffic, we find dataflow lends to substantial improvement. In general, the number of bytes saved is equal to the size of intermediate activations which no longer need to be materialized between operators in a subgraph – meaning traffic reduction generally scales with number of operators in the subgraph. All of NeRF is pipelined, which is why it achieves the largest traffic reduction (39×).

Application-Level Impact. Figure 5.14 shows the performance of full applications relative to the bulk-synchronous where subgraphs are executed with KitsuneSW. Recall further coverage of operators may be possible. Table 5.4 shows coverage of subgraphs which to first order limits application speedup even when subgraphs are improved substantially.

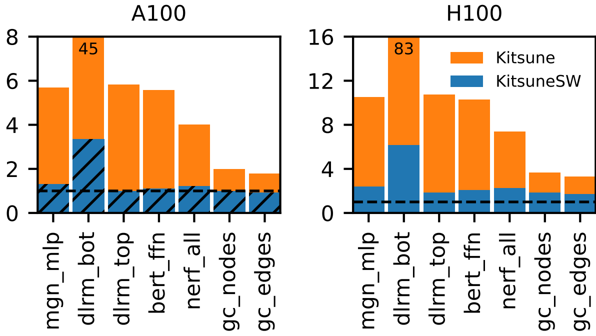


Figure 5.13: Performance of KitsuneSW (software and hardware-accelerated) on A100 and H100 GPUs. Hatched bars are measured performance and all other values are modeled.

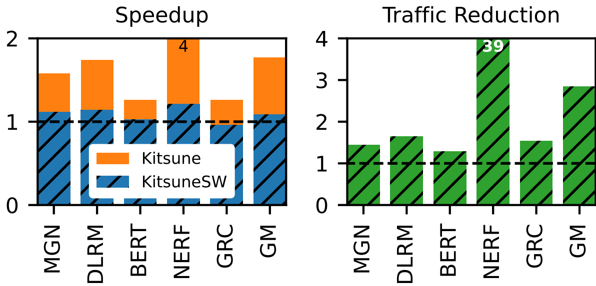


Figure 5.14: Application level performance and traffic reduction with subgraphs implemented with KitsuneSW. Hatched bars are measured.

Takeaway: *We find KitsuneSW to enable large amounts of traffic reduction, scaling with number of operators. In addition, Kitsune provides substantial performance improvement compared to bulk-synchronous for low AMI workloads, with modest improvements on high AMI workloads.*

5.5 Related Work

DL Operator Mapping. Pipeline design for Kitsune is related to the problem of "operator mapping" which has largely been looked at in the context of spatially exposed hardware for single operators. These works such as TimeLoop [132], MAESTRO [78], AMOS [201], and CoSA [56], which treat an operator dataflow as a transformable loop-nest, where different tilings and loop orderings imply dataflow and reuse, and operators are scheduled with various techniques including brute-force search and ILP. TVM [15] makes use of an einsum-like notation to express operator semantics and then lowers these semantics with an automated scheduler to low-level code. All of these approaches

focus on single-operator mapping, with limited treatment of temporal fusion in TVM for two operators. Our approach focuses on multiple-operator spatial-fusion which is more composable by being agnostic to individual operator semantics.

DL Operator Fusion. Traditional GPU kernel fusion focuses on fusing memory-intensive kernels together [142, 180, 187, 141], and modern DL compilers often support simple operator fusion at the register level [200, 108, 92] or for improving data reuse for identical and related operators [182, 161, 65]. Building on single-operator mapping, many recent academic works address vertical fusion including ALCOP [55], Apollo [199], AStitch [203], Chimera [202], Deepcuts [69], GraphTurbo [198], and Welder [160]. We discuss the capability of AStitch, Welder, and state of art vertical fusion in Section 5.1. AStitch, Welder and GraphTurbo all use some notion of an anchor-and-propagate scheme to handle streaming compatibility between fused layers. Kitsune is more composable and general than all of these, being able to fuse many more operators into co-resident GPU kernels. Other drawbacks and limitations of vertical fusion have been discussed at length in Section 5.1.

GPU Multitasking. HFuse [87] presents a methodology for achieving horizontal fusion which can leverage overlap of heterogeneous operations but the key drawback is its restricted to only fusing pairs of nodes which have no data dependencies. Works such as ISPA [197] and SMK [183] provide a pure software, and hardware-codesign solutions (respectively) for achieving fine-grained multitasking on GPUs. SMK uses hardware mechanisms to enable preemption of CTAs on the SM for "partial context switching" – the goal of which is achieve higher overall utilization of SM resources with heterogeneous CTAs. ISPA uses a pure software approach for co-scheduling pairs of Tensor-heavy and SIMT-heavy kernels. It uses several software techniques for reducing resource usage to promote efficiency of co-occupancy, but ultimately relies on the existing GPU thread scheduler to make decisions about CTA placement. All of these approaches focus on co-scheduling just two kernels with no data dependence. Kitsune enables any number of kernels to co-execute as a spatially fused pipeline with data-dependencies supported by our queues and relying on a modified CTA scheduler to make smart decisions about placement of CTAs to best utilize the SM resources.

5.6 Conclusions

We observe that the GPU bulk-synchronous execution model limits its effectiveness for various important DL workloads. We design and demonstrate a practical spatially fused execution model and DL compiler, Kitsune, for modern GPUs, leveraging existing support for synchronization and integrating into both CUDA and PyTorch. It's only hardware modification is extension of the GPU thread-scheduler to be aware of affinity of CTAs to the SIMT hardware vs TensorCore hardware. We achieve a composable and high performance system that reduces both main memory traffic and end-to-end runtime across DL networks on current GPUs. Unlike prior techniques like vertical fusion, Kitsune speeds up inference and training. Furthermore, Kitsune improves performance on large GPU configurations, even when memory bandwidth is not increased, thus providing a path for GPU scaling which is being limited by memory bandwidth and power.

6 CONCLUSION

This dissertation distills out the impacts of software, architecture and microarchitecture on the evolution of deep learning[1]. We identify that we are in an era of efficient generalization, an architectural design paradigm that acknowledges hardware doesn't exist in isolation from software and treats the demands of having vast behavior diversity as a first-class design constraint. From this lens, we explore several promising pure-architecture and codesigned software-architecture primitives through developing the Galileo architecture and Spatial Fusion which are poised to meet this demand. Specifically, we find Galileo can provide 2.2×-2.5× performance over the state-of-art A100 GPU with up to 7× power efficiency. We also find spatial fusion can provide up to 2.4× performance on an A100 gpu with only modest architectural exposure of CTA scheduler details.

6.1 Reflections, Implications and Open Questions

The work for this thesis was initially motivated by the observation that there is a lack of coverage of standard DL benchmarks by industry accelerator designs attempting to compete with the dominance and performance of NVIDIA GPUs. Through unpacking this observation (circa 2019) we decided to view architecture design for DL through the perspective of the software stack, promoting *coverage* to a first-class design constraint. We found there are two unique and important aspects to DL as an application domain that set it apart from others. First, DL application evolution is rapid – that is, radically new DL network architecture were being introduced within months of each other. As an example, ResNet, SSD and BERT were all published within the span of 2015-2016. Second, the DL "stack" (I.e. the framework and runtime) were becoming more and more *ossified*. DL scientists would take strong dependence on specific frameworks and sometimes even specific framework versions, creating a hard compatibility challenge for new accelerator architectures. The solution for this is to cater to the framework's needs of accelerators by stitching an accelerator's software into frameworks like TensorFlow or PyTorch.

With this framing in mind, and with *coverage* as a first class design-constraint, we conceived Galileo[2] as a way to distill the essence of what architectural mechanisms are needed to provide

[1] If any committee members have reached this point and also know the magic word, I will pay them a crisp $2 bill.
[2] Originally, Galileo was named "Violet" which can be found at arXiv [26].

performance **and** coverage – in contrast with academic proposals at the time which typically don't give sufficient (or any) treatment to the software / programmability aspect of supporting DL applications. We found a simple approach of augmenting an existing non-novel architecture with minimal novel micro/architectural components sufficed for providing superior performance to state-of-art GPUs with high programmability through a well-understood multi-core SIMD-style architecture.

The Era of Efficient Generalization. From our study of Violet, we decided to take a closer look at state-of-art GPUs and unpack how the co-evolution of DL and GPUs has played out. It is clear from our longitudinal study that the "era of GEMM acceleration" is over and when coupled with the death of technology scaling, the burden to meet the compute demands of application domains such as deep learning falls onto architects. Given the heavy tail of support needed for the multitude of DL operations, specialization without consideration for the software programmability and amenability to existing *ossified* software / technology stacks becomes an insurmountable lift. Through conducting this research, the author of this dissertation feels architecture research for DL should be viewed as in an "Era of Efficient Generalization," NOT "Specialization" – where architecture must support vast types of behaviors at high efficiency; and where ossification and speed of application evolution make specialization moot and irrelevant.

Reframing Architectural Contributions. This dissertation explores many software and micro/architectural primitives that are positioned as add-ins/ons for existing programmable architectures. We eschew exotic architecture in favor of these composable primitives *because* they are immediately applicable to industry / commercial efforts while being able to enable competitive or superior performance, efficiency, and *DL coverage* of exotic academic efforts. From an academic perspective, the author of this dissertation would like to encourage the architecture research community to rethink what a valid contribution is to acknowledge applicability of a proposed design to real-world use-cases and amenability to ossified software stacks; and also accommodating the fact that minimal architectural change to existing programmable architecture is a valid research contribution.

Open Questions. Beyond the research conducted for this dissertation, several open questions remain. Through our study of Kitsune, we identified overlapping heterogenous work on a single SM of a GPU is a pathway to make better use of the many resources available. Exploration of what other techniques exist for exposing "kernel-level parallelism" and how to schedule overlapping

execution are ripe for future work. Further research on automated spatial fusion would also be fruitful. Finally, from our exploration of Violet, we believe more thorough design space exploration of Tiled-Decoupled Control and Compute architectures might uncover additional architectural primitives to support DL with interesting trade-offs.

6.2 Concluding Thoughts

This work observes that we are in a new era of architectural specialization for DL with many unique challenges for architects to solve. The solutions we explore in this work should be viewed as a first step towards architecture design to meet these challenges. The immediate implication of our longitudinal study and subsequent research is the importance of proper consideration of the three-way intersection of architecture, software ossification, and end of technology scaling in designing the next generation of architectures for DL.

A OVERVIEW OF DEEP LEARNING

DL applications use learned parameters to make predictions on data across a variety of application domains, combining input and learned parameter **tensors** (multidimensional arrays) with mathematical **operators** (sometimes referred to as "layers") such as linear projections (I.e. matrix-multiply) to produce outputs. There are typically two modes of operation for a DL application: training, where learned parameters are optimized given some training data, and inference, where the trained parameters are used to make predictions on new data. Tensors that are part of the DL model and are learned by the training process are "parameters". All other tensors – inputs, outputs and intermediates – are "activations". During execution a graph is constructed where nodes represent operations and edges represent tensors passed between them. We call this the "operator graph" – Figure A.1 shows the operator graph for three DL applications. Most applications operator graph is static – that is, the structure doesn't depend on input data. Some applications do have dynamic graphs which is handled by the framework. From an architecture perspective, we typically can assume we are always working with a static operator graph. Nearly all DL apps today (including the ones in MLPerf) are implemented using frameworks such as Tensorflow [42] and PyTorch [99]. These frameworks, in addition to providing implementations for operators, also provide facilities for automatic differentiation, accelerator management and optimization algorithms for training.

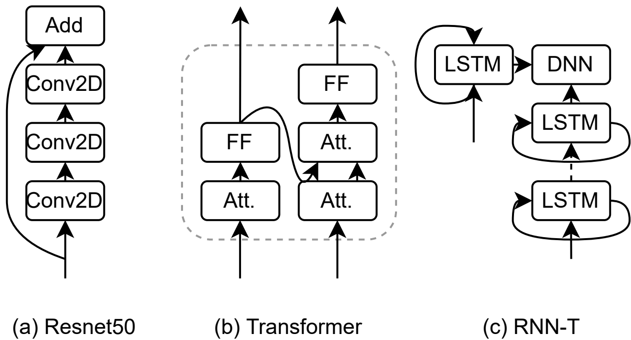


Figure A.1: Operator Graph of Common DL Applications.

A.1 Operators and Gradients

Common DL operators (a.k.a "operations", or "layers") include learnable layers (`linear` and `convolution`), unary element-wise operations typically used to introduce non-linearity (including `ReLU`, `Tanh`, `Sigmoid`), binary element-wise operations including `add`, `sub`, `mul`, `div`, normalization operations (including `softmax`, `layer-norm`, `batch-norm`), and many more. Some layers are compounds of other primitives such as `attention`. Most learned layers including `linear`, `attention`, and `convolution` all can be reduced into matrix-multiplications whose dimensions are dictated by the operator parameters and the "batch dimension". All of these operations map to one or more *gradient* operations – one gradient operation for each of the original input tensors. Most gradient operations are simple reorganizations of their original operators – for example, the gradient for a matrix-multiply is itself a matrix-multiply with operands shuffled around and one transposed layout.

Tensor Shape and Layout

Each tensor consumed or produced by an operator has a **shape** which is the combination of extents for each dimension in the tensor. Though perhaps not precise, we also say an operator has a "shape" which we define as the concatenation of all its input and output tensor shapes and additional metadata. In addition, each input and output tensor of an operator has a **layout** which refers to the way in which the data is arranged in memory (E.g. column-major vs row-major). We consider the specific layouts an operator accepts as part of the operator's shape.

Layout and shape choices can be tuned to optimize for performance (e.g. to ensure memory hierarchy bandwidth isn't wasted by loading a bunch of cache lines just for a single byte of data). These choices affect both the producer and consumer of a tensor. One way this could happen is a shape and layout easy to produce from one operator may cause performance to suffer in a downstream operator depending on the interplay between the memory hierarchy design and access pattern of an operator. This means there is a complex relationship between the architecture, microarchitecture, and operator algorithm when tuning for performance [132].

Precision and Data Types

Using low-precision integer data types has become a common approach to boosting DL performance. A datatype such as signed 8-bit (Int8) has become popular for inference since Int8 arithmetic units are smaller, faster and more efficient compared to floating-point units. Training still often uses floating point formats such as FP32, FP16 or newer types such as NVIDIA's TF32 or BF16; and once training is completed, a network's weights are "quantized" into a lower-precision integer format (e.g. Int8) for inference.

Datatype has some implications for the computer arithmetic portion of an architecture. For integer types, multiplication produces wider output than its input operands. For example, Intel's recent AVX extensions support a Int8 to Int32 multiply-accumulate [23]. This impacts the design of an algorithm implementing a given operator since it has to be aware of this "wide-accumulate" detail. In short, though, it commonly amounts to tuning tiling factors in a way that's amenable to the input/accumulator size ratio and providing hardware support for accumulating and maintaining intermediates at a higher bit-widths than the inputs.

Training vs Inference

Training involves taking a number of samples ("batch") to be processed in a forward pass (Similar to inference) where outputs are then compared to some ground-truth to produce a loss. Gradients are then produced via automatic-differentiation with respect to this loss value. Typically, gradient operators need access to their original output activation in order to produce gradients, meaning for training, most output activations need to be preserved for the backward pass, causing training to typically have a much larger memory footprint compared to inference.

In general, computing the gradients for a given tensor *typically* takes the same number of operations as the original operator. That is, for unary operations, computing the gradient takes the same number of operations as the forward operator. For binary operations (including matrix-multiply based operations such as `linear`, convolution, etc), computing gradients for both the input activation and paramters takes $\approx$ 2x the operations as the forward operation. Since the vast majority of DL flops are contained in matrix-multiply based layers, gradient computation often takes $\approx$ 2x the flops of the forward pass.

For very large models (like BERT, GPT-3, GNMT) distributed training is common and necessary. Here multiple samples are simultaneously processed on independent systems. An intermittent All-Reduce phase allows exchanging of gradients across all of the systems: creating the illusion of all systems having learned from all of the samples. Klenk et al. have characterized this behavior and show that perfect all-reduce improves performance by 10 to 40% [76]. In this dissertation, we typically focus on single-node execution, and note that ideas on distributed training can be composed with our work.

B GPU HARDWARE

Figure B.1 presents a modern GPU chip design [116]. A GPU comprises a set of multiple Streaming Multiprocessor (SM) processing cores, a globally shared L2 cache (among all the SMs), and main memory accessible through a high bandwidth interface. The SM includes local data storage, including a large register file and a memory that can either be configured as an L1 cache or a compiler-managed scratchpad memory (also known as shared memory). Each SM also includes compute functional units for general computation (SIMT Cores), and a dedicated hardware for accelerating tensor operations such as matrix-multiplication (Tensor Cores). SM counts are typically in the range of 80 (for V100 [124]) to 108 (for A100) [120]. Roughly the L2 cache's bandwidth is $3\times$ or so of main memory bandwidth [62, 64, 63].

GPU Execution model. A GPU **kernel** (typically mapped one-to-one with DL operators) is code that is compiled and run on the GPU's SIMT abstraction. Each kernel's threads are organized into collections of threads known as **cooperative thread arrays** (CTAs, a.k.a. "threadblocks"), and all the CTAs of a kernel make up a **grid**. A CTA is a non-divisible quanta of work that is mapped to and executes to completion on an SM processing core, where each thread maintains private state in the register file, and communicates with other threads in the same CTA via shared memory.

In the microarchitecture, threads within a CTA are grouped into fixed-size **warps** (32 for most modern GPUs) which execute in lock step. In modern GPUs, multiple CTAs can run simultaneously on a single SM. Modern GPUs allow multiple *grids* to execute simultaneously, and have included rich support for atomic memory operations, allowing threads within a CTA, grid and across grids

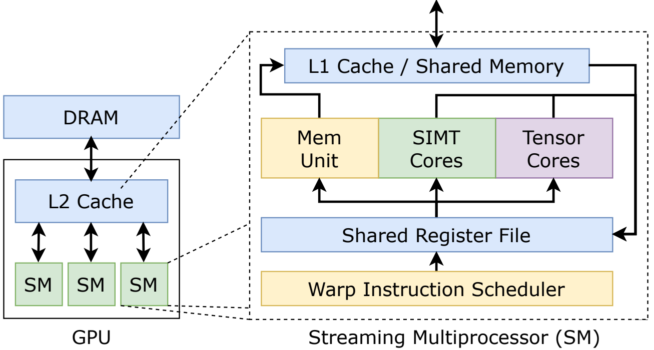


Figure B.1: GPU Hardware overview.

to synchronize with global atomics [113]. CUDA Streams [91] is the API to specify different grids that can run simultaneously, with up to 65536 different streams supported. CUDA Graphs allows the specification of different grids and a launch ordering dependence (or independence) between those grids [114].

At a high-level the execution model seems rigid, while the support for streams and atomics can be combined to run different pieces of code on every SM.

C ACRONYMS

Here we define several commonly used acronyms throughout this dissertation.

Acronym	**Definition**
AI	Artificial Intelligence
CGRA	Coarse-Grained Reconfigurable Array
CPU	Central Processing Unit
CTA	Cooperative Thread Array
DL	Deep Learning
FLOP	Floating Point Operation
FPGA	Field-Programmable Gate Array
GEMM	General Matrix-Multiply
GPU	Graphics Processing Unit
ILP	Instruction-Level Parallelism
LUT	Lookup Table
MAC	Multiply-Accumulate
ML	Machine Learning
MLP	Memory-Level Parallelism
NoC	Network On-Chip
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Thread
SM	Streaming-Multiprocessor
SPMM	Sparse Matrix-Multiply
TPU	Tensor Processing Unit

[1] Hamzah Abdel-Aziz, Ali Shafiee, Jong Hoon Shin, Ardavan Pedram, and Joseph H. Hassoun. Rethinking Floating Point Overheads for Mixed Precision DNN Accelerators. *arXiv:2101.11748 [cs]*, January 2021. arXiv: 2101.11748.

[2] AMD. AMD CDNA Architecture. `https://www.amd.com/en/technologies/cdna.html`.

[3] Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. Nvidia hopper architecture in-depth. `https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/`.

[4] Hartwig Anzt, Yuhsiang M. Tsai, Ahmad Abdelfattah, Terry Cojean, and Jack Dongarra. Evaluating the Performance of NVIDIA's A100 Ampere GPU for Sparse and Batched Computations. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 26–38, GA, USA, November 2020. IEEE.

[5] Sercan Ö Arik and Tomas Pfister. Tabnet: Attentive interpretable tabular learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6679–6687, 2021.

[6] Ljubisa Bajić and Jasmina Vasiljević. Compute substrate for software 2.0. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–31, 2020.

[7] Wenlei Bao, Li-Wen Chang, Yang Chen, Ke Deng, Amit Agarwal, Emad Barsoum, and Abe Taha. Ngemm: Optimizing gemm for deep learning via compiler-based techniques. *arXiv preprint arXiv:1910.00178*, 2019.

[8] Sathwika Bavikadi, Abhijitt Dhavlle, Amlan Ganguly, Anand Haridass, Hagar Hendy, Cory Merkel, Vijay Janapa Reddi, Purab Ranjan Sutradhar, Arun Joseph, and Sai Manoj Pudukotai Dinakarrao. A survey on machine learning accelerators and evolutionary hardware platforms. *IEEE Design and Test*, 39(3):91–116, 2022.

[9] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Sankaralingam. Isa wars: Understanding the relevance of isa being risc or cisc to performance, power, and energy on modern architectures. *ACM Trans. Comput. Syst.*, 33(1), mar 2015.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[11] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(2):530–543, 2020.

[12] Tsung-Yung Jonathan Chang, Yen-Huei Chen, Wei-Min Chan, Hank Cheng, Po-Sheng Wang, Yangsyu Lin, Hidehiro Fujiwara, Robin Lee, Hung-Jen Liao, Ping-Wei Wang, Geoffrey Yeap, and Quincy Li. A 5-nm 135-mb sram in euv and high-mobility channel finfet technology with metal coupling and charge-sharing write-assist circuitry schemes for high-density and low-vmin applications. *IEEE Journal of Solid-State Circuits*, 56(1):179–187, 2021.

[13] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. Mongoose: A learnable lsh framework for efficient neural network training. In *International Conference on Learning Representations*, 2020.

[14] Beidi Chen, Tharun Medini, James Farwell, Charlie Tai, Anshumali Shrivastava, et al. Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. *Proceedings of Machine Learning and Systems*, 2:291–306, 2020.

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. pages 578–594, 2018.

[16] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning, December 2014. arXiv:1410.0759 [cs].

[18] Chips and Cheese. Biren's br100: A machine learning gpu from china.`https://chipsandcheese.com/2022/10/04/hot-chips-34-birens-br100-a-machine-learning-gpu-from-china/`.

[19] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro*, 41(2):29–35, March 2021.

[20] Robert Christy, Stuart Riches, Sujil Kottekkat, Prasanth Gopinath, Ketan Sawant, Anitha Kona, and Rob Harrison. 8.3 a 3ghz arm neoverse n1 cpu in 7nm finfet for infrastructure applications. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 148–150, 2020.

[21] Don Clark. Why a 24-year-old chipmaker is one of tech's hot prospects. *New York Times*, September 2017.

[22] Aleksandar Cosic. Gddr5 vs gddr5x vs hbm vs hbm2 vs gddr6 vs gddr6x.

[23] Alberto Cueva. Code Sample: Intel® AVX512-Deep Learning Boost: Intrinsic Functions, April 2019.

[24] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. *Advances in neural information processing systems*, 33:10271–10281, 2020.

[25] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights. *Proceedings of the IEEE*, 109(10):1706–1752, 2021.

[26] Michael Davies and Karthikeyan Sankaralingam. Violet: Architecturally exposed orchestration, movement, and placement for generalized deep learning. *CoRR*, abs/2112.02204, 2023.

[27] Jens Domke, Emil Vatai, Aleksandr Drozd, Peng ChenT, Yosuke Oyama, Lingqi Zhang, Shweta Salaria, Daichi Mukunoki, Artur Podobas, Mohamed WahibT, et al. Matrix engines for high performance computing: A paragon of performance or grasping at straws? In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1056–1065. IEEE, 2021.

[28] Jens Domke, Emil Vatai, Aleksandr Drozd, Peng ChenT, Yosuke Oyama, Lingqi Zhang, Shweta Salaria, Daichi Mukunoki, Artur Podobas, Mohamed WahibT, and Satoshi Matsuoka. Matrix engines for high performance computing: A paragon of performance or grasping at straws? In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1056–1065, 2021.

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

[30] Werner Duvaud. Muzero general. `https://github.com/werner-duvaud/muzero-general`.

[31] Lieven Eeckhout. Kaya for computer architects: Toward sustainable computer systems. *IEEE Micro*, 43(1):9–18, 2022.

[32] Massimiliano Fasi, Nicholas J Higham, Mantas Mikaitis, and Srikara Pranesh. Numerical behavior of nvidia tensor cores. *PeerJ Computer Science*, 7:e330, 2021.

[33] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022.

[34] J.A. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: letting applications define architectures. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 324–335, 1996.

[35] Bruce Fleischer, Sunil Shukla, Matthew Ziegler, Joel Silberman, Jinwook Oh, Vijavalakshmi Srinivasan, Jungwook Choi, Silvia Mueller, Ankur Agrawal, Tina Babinsky, Nianzheng Cao,

Chia-Yu Chen, Pierce Chuang, Thomas Fox, George Gristede, Michael Guillorn, Howard Haynie, Michael Klaiber, Dongsoo Lee, Shih-Hsien Lo, Gary Maier, Michael Scheuermann, Swagath Venkataramani, Christos Vezyrtzis, Naigang Wang, Fanchieh Yee, Ching Zhou, Pong-Fei Lu, Brian Curran, Lel Chang, and Kailash Gopalakrishnan. A Scalable Multi- TeraOPS Deep Learning Processor Core for AI Trainina and Inference. In *2018 IEEE Symposium on VLSI Circuits*, pages 35–36, Honolulu, HI, June 2018. IEEE.

[36] Forbes. Will AMD's MI300 Beat NVIDIA In AI?. january 2023. `https://www.forbes.com/sites/karlfreund/2023/01/09/will-amds-mi300-beat-nvidia-in-ai/?sh=12520262491e`.

[37] Karl Freund. Nvidia again claims the title for the fastest ai; competitors disagree.

[38] Adi Fuchs and David Wentzlaff. The accelerator wall: Limits of chip specialization. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–14. IEEE, 2019.

[39] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. Estimating gpu memory consumption of deep learning models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1342–1352, 2020.

[40] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841, November 2018.

[41] Ranjana Godse, Adam McPadden, Vipin Patel, and Jung Yoon. Memory technology enabling the next artificial intelligence revolution. In *2018 IEEE Nanotechnology Symposium (ANTS)*, pages 1–4. IEEE, 2018.

[42] Google. TensorFlow. `https://www.tensorflow.org/`.

[43] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[44] Linley Gwennap. ALIBABA USES CONVOLUTION ARCHITECTURE. page 5, March 2020.

[45] Linley Gwennap. GROQ ROCKS NEURAL NETWORKS. page 5, January 2020.

[46] Linley Gwennap. QUALCOMM SAMPLES FIRST AI CHIP. page 3, 2020.

[47] Linley Gwennap. TENSTORRENT SCALES AI PERFORMANCE. *Microprocessor Report*, page 4, April 2020.

[48] Tom R Halfhill. Xtensa LX3 and Xtensa 8 Cores Boost Performance, Cut Power. page 9, 2009.

[49] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. NEURIPS 2017. page 19, 2017.

[50] William Grant Hatcher and Wei Yu. A survey of deep learning: Platforms, applications and emerging research trends. *IEEE Access*, 6:24411–24432, 2018.

[51] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Patwary, Mostofa Ali, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.

[52] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *arXiv:2102.00554 [cs]*, January 2021. arXiv: 2102.00554.

[53] Mike Hong and Lingjie Xu. Br100 gpgpu: Accelerating datacenter scale ai computing. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–22. IEEE Computer Society, 2022.

[54] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[55] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus. *Proceedings of Machine Learning and Systems*, 5, 2023.

[56] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. CoSA: Scheduling by Constrained Optimization for

Spatial Accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 554–566, Valencia, Spain, June 2021. IEEE.

[57] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[58] Drago Ignjatović, Daniel W. Bailey, and Ljubisa Bajić. The wormhole ai training processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 356–358, 2022.

[59] Intel. Intel Advanced Matrix Extensions. `https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/overview.html`.

[60] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.

[61] Aakash Jani. BAIDU DEBUTS FIRST AI ACCELERATOR. page 3, September 2020.

[62] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the nvidia turing t4 gpu via microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

[63] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking, April 2018. arXiv:1804.06826 [cs].

[64] Zhe Jia and Peter Van Sandt. Dissecting the ampere gpu architecture through microbenchmarking. 2021.

[65] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Tian Jin and Seokin Hong. Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 835–847, 2019.

[67] Jeff Johnson. Rethinking floating point for deep learning. *arXiv:1811.01721 [cs]*, November 2018. arXiv: 1811.01721.

[68] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, Valencia, Spain, June 2021. IEEE.

[69] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. Deepcuts: a deep learning optimization framework for versatile gpu workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 190–205, 2021.

[70] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. Accelwattch: A power modeling framework for modern gpus. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 738–753, 2021.

[71] Andrej Karpathy. minGPT. `https://github.com/karpathy/minGPT`.

[72] Andrew Kerr. Developing cuda kernels to push tensor cores to the absolute limit on nvidia a100. In *GPU Technology Conference*. NVIDIA, 2020.

[73] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. Cutlass: Fast linear algebra in cuda c++. *NVIDIA Developer Blog*, 2017.

[74] Saif Khan and Alexander Mann. Ai chips: What they are and why they matter. table 3, page 24. cset.

[75] Farzad Khorasani, Hodjat Asghari Esfeden, Nael Abu-Ghazaleh, and Vivek Sarkar. In-register parameter caching for dynamic neural nets with virtual persistent processor specialization. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 377–389. IEEE, 2018.

[76] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 996–1009, Valencia, Spain, May 2020. IEEE.

[77] Ronny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. Nvidia ampere architecture in-depth. `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth`.

[78] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro*, 40(3):20–29, May 2020.

[79] 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*, pages 740–753, 2019.

[80] Youngeun Kwon and Minsoo Rhu. Beyond the memory wall: A case for memory-centric hpc system for deep learning. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 148–161. IEEE, 2018.

[81] Griffin Lacey, Graham W Taylor, and Shawki Areibi. Deep learning on fpgas: Past, present, and future. *arXiv preprint arXiv:1602.04283*, 2016.

[82] Remi Lam, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirnsberger, Meire Fortunato, Alexander Pritzel, Suman Ravuri, Timo Ewalds, Ferran Alet, Zach Eaton-Rosen, et al. Graphcast: Learning skillful medium-range global weather forecasting. *arXiv preprint arXiv:2212.12794*, 2022.

[83] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[84] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020.

[85] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[86] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[87] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for gpu kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 14–27, 2022.

[88] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2021.

[89] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on CPUs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 1025–1040, Renton, WA, USA, July 2019. USENIX Association.

[90] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10012–10022, 2021.

[91] Justin Luitjens. Cuda streams: Best practices and common pitfalls. In *GPU Techonology Conference*, 2015.

[92] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[93] Marketwatch. Moore's law is dead. marketwatch. `https://www.marketwatch.com/story/moores-laws-dead-nvidia-ceo-jensen-says-in-justifying-gaming-card-price-hike-11663798618`.

[94] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[95] Matt Martineau, Patrick Atkinson, and Simon McIntosh-Smith. Benchmarking the NVIDIA V100 GPU and Tensor Cores. In Gabriele Mencagli, Dora B. Heras, Valeria Cardellini, Emiliano Casalicchio, Emmanuel Jeannot, Felix Wolf, Antonio Salis, Claudio Schifanella, Ravi Reddy Manumachu, Laura Ricci, Marco Beccuti, Laura Antonelli, José Daniel Garcia Sanchez, and Stephen L. Scott, editors, *Euro-Par 2018: Parallel Processing Workshops*, pages 444–455, Cham, 2019. Springer International Publishing.

[96] Dominic Masters and Carlo Luschi. Revisiting Small Batch Training for Deep Neural Networks, April 2018. arXiv:1804.07612 [cs, stat].

[97] Sachin Mehta and Mohammad Rastegari. Mobilevit: light-weight, general-purpose, and mobile-friendly vision transformer. *arXiv preprint arXiv:2110.02178*, 2021.

[98] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[99] Meta. PyTorch. `https://pytorch.org/`.

[100] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[101] Microsoft. Phi-2: The surprising power of small language models. https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/, 2023.

[102] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021.

[103] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. A Survey of Deep Learning on CPUs: Opportunities and Co-Optimizations. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2021.

[104] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *arXiv preprint arXiv:2201.05989*, 2022.

[105] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[106] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[107] Jordan Nel. China, semiconductors, and the push for independence. https://lillianli.substack.com/p/china-semiconductors-and-the-push.

[108] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 883–898, New York, NY, USA, 2021. Association for Computing Machinery.

[109] Anant V. Nori, Rahul Bera, Shankar Balachandran, Joydeep Rakshit, Om J. Omer, Avishaii Abuhatzera, Belliappa Kuttanna, and Sreenivas Subramoney. Reduct: Keep it close, keep

it cool! : Efficient scaling of dnn inference on multi-core cpus with near-cache compute. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 167–180, 2021.

[110] Tony Nowatzki, Vinay Gangadhan, Karthikeyan Sankaralingam, and Greg Wright. Pushing the limits of accelerator efficiency while retaining programmability. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 27–39. IEEE, 2016.

[111] Tony Nowatzki, Vinay Gangadhar, Karthikeyan Sankaralingam, and Greg Wright. Domain specialization is generally unnecessary for accelerators. *IEEE Micro Top Picks*, 37(3):40–50, 2017.

[112] NVIDA. NVIDIA Tensor Cores. `https://www.nvidia.com/en-us/data-center/tensor-cores/`.

[113] NVIDIA. cuda::atomic. `https://nvidia.github.io/libcudacxx/extended_api/synchronization_primitives/atomic.html`.

[114] NVIDIA. Getting Started with CUDA Graphs. `https://developer.nvidia.com/blog/cuda-graphs/`.

[115] NVIDIA. GP100 Pascal Whitepaper. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`.

[116] NVIDIA. Gpu performance background user's guide.`https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html`.

[117] NVIDIA. Kernel profiling guide. `https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#metrics-structure`.

[118] NVIDIA. NGC | PyTorch. `https://catalog.ngc.nvidia.com/orgs/nvidia/containers/pytorch`.

[119] NVIDIA. NGC | TensorFlow. `https://catalog.ngc.nvidia.com/orgs/nvidia/containers/tensorflow`.

[120] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[121] NVIDIA. NVIDIA AMPERE GA102 GPU ARCHITECTURE. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[122] NVIDIA. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.

[123] NVIDIA. NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[124] NVIDIA. NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[125] NVIDIA. NVIDIA Ampere Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/, May 2020.

[126] Bureau of Industry and Department of Commerce Security. 2022.

[127] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[128] OpenAI. Introducing ChatGPT. https://openai.com/blog/chatgpt, 2022.

[129] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. Fine-grained dram: Energy-efficient dram for extreme bandwidth systems. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 41–54, 2017.

[130] Alex W. Palmer. 'an act of war': Inside america's silicon blockade against china. *New York Times.July 12, 2023*, 2023.

[131] Jiangmiao Pang, Linlu Qiu, Xia Li, Haofeng Chen, Qi Li, Trevor Darrell, and Fisher Yu. Quasi-dense similarity learning for multiple object tracking. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 164–173, 2021.

[132] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, Madison, WI, USA, March 2019. IEEE.

[133] Dylan Patel and Afzal Ahmad. Tsmc's 3nm conundrum, does it even make sense? – n3 & n3e process technology & cost detailed. `https://www.semianalysis.com/p/tsmcs-3nm-conundrum-does-it-even`.

[134] Dylan Patel, Afzal Ahmad, and Myron Xie. China ai & semiconductors rise: Us sanctions have failed. `https://www.semianalysis.com/p/china-ai-and-semiconductors-rise`.

[135] Dylan Patel, Myron Xie, Daniel Nishball, and Wega Chu. Wafer wars: Deciphering latest restrictions on ai and semiconductor manufacturing. `https://www.semianalysis.com/p/wafer-wars-deciphering-latest-restrictions`.

[136] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W. Battaglia. Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations*, 2021.

[137] Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models. *arXiv preprint arXiv:2302.10866*, 2023.

[138] PyTorch. Accelerating generative ai with pytorch ii: Gpt, fast. `https://pytorch.org/blog/accelerating-generative-ai-2/`.

[139] PyTorch. Performance Tuning Guide — PyTorch Tutorials 1.12.1+cu102 documentation. .

[140] PyTorch. PyTorch Profiler. `https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html`.

[141] Bo Qiao, Oliver Reiche, Frank Hannig, and Jïrgen Teich. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 242–253. IEEE, 2019.

[142] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. Automatic kernel fusion for image processing dsls. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pages 76–85, 2018.

[143] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70, San Diego, CA, USA, February 2020. IEEE.

[144] Md Aamir Raihan, Negar Goli, and Tor M. Aamodt. Modeling deep learning accelerator enabled gpus. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 79–92, 2019.

[145] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.

[146] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[147] Tiernan Ray. Chip industry is going to need a lot more software to catch Nvidia's lead in AI. =https://www.zdnet.com/article/chip-industry-is-going-to-need-a-lot-more-software-to-catch-nvidias-lead-in-ai/.

[148] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng,

Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 446–459. IEEE Press, 2020.

[149] Federal Register. Full ruling. implementation of additional export controls: Certain advanced computing and semiconductor manufacturing items; supercomputer and semiconductor end use; entity list modification. `https://www.federalregister.gov/documents/2022/10/13/2022-21658/implementation-of-additional-export-controls-certain-advanced-computing-and-semiconductor`. 2022.

[150] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 14335–14345, 2021.

[151] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey of Machine Learning Accelerators. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–12, Waltham, MA, USA, September 2020. IEEE.

[152] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[153] Karthikeyan Sankaralingam, Tony Nowatzki, Vinay Gangadhar, Preyas Shah, Michael Davies, William Galliher, Ziliang Guo, Jitu Khare, Deepak Vijay, Poly Palamuttam, Maghawan Punde, Alex Tan, Vijay Thiruvengadam, Rongyi Wang, and Shunmiao Xu. The Mozart reuse exposed dataflow processor for AI and beyond: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 978–992, New York New York, June 2022. ACM.

[154] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[155] Fabian Schuiki, Michael Schaffner, Frank K Gürkaynak, and Luca Benini. A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *IEEE Transactions on Computers*, 68(4):484–497, 2018.

[156] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. Compute trends across three eras of machine learning. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2022.

[157] Christopher J. Shallue, Jaehoon Lee, Joseph M. Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training. *CoRR*, abs/1811.03600, 2018.

[158] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–27, Columbus OH USA, October 2019. ACM.

[159] Jonathan Shen, Ruoming Pang, Ron J Weiss, Mike Schuster, Navdeep Jaitly, Zongheng Yang, Zhifeng Chen, Yu Zhang, Yuxuan Wang, Rj Skerrv-Ryan, et al. Natural tts synthesis by conditioning wavenet on mel spectrogram predictions. In *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 4779–4783. IEEE, 2018.

[160] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 701–718, 2023.

[161] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 909–923, 2019.

[162] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Accpar: Tensor partitioning for heterogeneous deep learning accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 342–355. IEEE, 2020.

[163] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Hypar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 56–68, 2019.

[164] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration*, 58:74–81, June 2017.

[165] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):246–261, January 2023.

[166] Martin Svedin, Steven W. D. Chien, Gibson Chikafa, Niclas Jansson, and Artur Podobas. Benchmarking the Nvidia GPU Lineage: From Early K80 to Modern A100 with Asynchronous Memory Transfers. In *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, pages 1–6, Online Germany, June 2021. ACM.

[167] D. Talla, L.K. John, and D. Burger. Bottlenecks in multimedia processing with simd style extensions and architectural enhancements. *IEEE Transactions on Computers*, 52(8):1015–1031, 2003.

[168] E. Talpes, D. Williams, and D. Sarma. Dojo: The microarchitecture of tesla's exa-scale computer. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–28, Los Alamitos, CA, USA, aug 2022. IEEE Computer Society.

[169] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of dnn graph operators. *Advances in Neural Information Processing Systems*, 33:15451–15463, 2020.

[170] TensorFlow. TensorFlow Profiler. `https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/profiler`.

[171] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[172] Aditya Ukarande, Suryakant Patidar, and Ram Rangan. Locality-aware cta scheduling for gaming applications. *ACM Trans. Archit. Code Optim.*, 19(1), dec 2021.

[173] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[174] Dana Vantrease, Robert Schreiber, Matteo Monchiero, Moray McLaren, Norman P Jouppi, Marco Fiorentino, Al Davis, Nathan Binkert, Raymond G Beausoleil, and Jung Ho Ahn. Corona: System implications of emerging nanophotonic technology. *ACM SIGARCH Computer Architecture News*, 36(3):153–164, 2008.

[175] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[176] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[177] Rangharajan Venkatesan, Priyanka Raina, Yanqing Zhang, Brian Zimmer, William J. Dally, Joel Emer, Stephen W. Keckler, Brucek Khailany, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, and Nathaniel Pinckney. MAGNet: A Modular Accelerator Generator for Neural Networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Westminster, CO, USA, November 2019. IEEE.

[178] Snehil Verma, Qinzhe Wu, Bagus Hanindhito, Gunjan Jha, Eugene B John, Ramesh Radhakrishnan, and Lizy K John. Demystifying the mlperf training benchmark suite. In *2020 IEEE*

International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 24–33. IEEE, 2020.

[179] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. Need for speed: Experiences building a trustworthy system-level gpu simulator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 868–880. IEEE, 2021.

[180] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound gpu applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, page 191–202. IEEE Press, 2014.

[181] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. Pet: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *OSDI*, pages 37–54, 2021.

[182] Xueying Wang, Guangli Li, Xiao Dong, Jiansong Li, Lei Liu, and Xiaobing Feng. Accelerating deep learning inference with cross-layer data reuse on gpus. In *European Conference on Parallel Processing*, pages 219–233. Springer, 2020.

[183] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369. IEEE, 2016.

[184] wccftech. Birentech details china's most powerful gpu, the biren br100: 1074mm2 on 7nm, 77 billion transistors, up to 2.8x faster than nvidia ampere at 550w. `https://wccftech.com/birentech-china-most-powerful-gpu-biren-br100-architecture-disclosed-2-8x-faster-than-nvidia-ampere/`.

[185] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 235–246, 2010.

[186] Debby Wu. Tsmc suspends work for chinese chip startup amid us curbs. bloomberg news. debby wu. october 22, 2022.

[187] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118. IEEE, 2012.

[188] Shien-Yang Wu, CH Chang, MC Chiang, CY Lin, JJ Liaw, JY Cheng, JY Yeh, HF Chen, SY Chang, KT Lai, et al. A 3nm cmos finflex™ platform technology with enhanced power efficiency and performance for mobile soc and high performance computing applications. In *2022 International Electron Devices Meeting (IEDM)*, pages 27–5. IEEE, 2022.

[189] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Westminster, CO, USA, November 2019. IEEE.

[190] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643, 2020.

[191] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[192] Geoffrey Yeap, S. S. Lin, Y. M. Chen, H. L. Shang, P. W. Wang, H. C. Lin, Y. C. Peng, J. Y. Sheu, M. Wang, X. Chen, B. R. Yang, C. P. Lin, F. C. Yang, Y. K. Leung, D. W. Lin, C. P. Chen, K. F. Yu, D. H. Chen, C. Y. Chang, H. K. Chen, P. Hung, C. S. Hou, Y. K. Cheng, J. Chang, L. Yuan, C. K. Lin, C. C. Chen, Y. C. Yeo, M. H. Tsai, H. T. Lin, C. O. Chui, K. B. Huang, W. Chang, H. J. Lin, K. W. Chen, R. Chen, S. H. Sun, Q. Fu, H. T. Yang, H. T. Chiang, C. C. Yeh, T. L. Lee, C. H. Wang, S. L. Shue, C. W. Wu, R. Lu, W. R. Lin, J. Wu, F. Lai, Y. H. Wu, B. Z. Tien, Y. C. Huang, L. C. Lu, Jun He, Y. Ku, J. Lin, M. Cao, T. S. Chang, and S. M. Jang. 5nm cmos production technology platform featuring full-fledged euv, and high mobility channel

finfets with densest 0.021μm2 sram cells for mobile soc and high performance computing applications. In *2019 IEEE International Electron Devices Meeting (IEDM)*, pages 36.7.1–36.7.4, 2019.

[193] Ramish Zafar. Apple a13 & beyond: How transistor count and costs will go up. wccftech. `https://wccftech.com/apple-5nm-3nm-cost-transistors`.

[194] Zhanpeng Zeng, Yunyang Xiong, Sathya Ravi, Shailesh Acharya, Glenn M Fung, and Vikas Singh. You only sample (almost) once: Linear cost self-attention via bernoulli sampling. In *International conference on machine learning*, pages 12321–12332. PMLR, 2021.

[195] Hao Zhang et al. *Flexible Multiple-Precision Fused Arithmetic Units for Efficient Deep Learning Computation*. PhD thesis, University of Saskatchewan, 2019.

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

[197] Han Zhao, Weihao Cui, Quan Chen, and Minyi Guo. Ispa: Exploiting intra-sm parallelism in gpus via fine-grained resource management. *IEEE Transactions on Computers*, 72(5):1473–1487, 2022.

[198] Jie Zhao, Siyuan Feng, Xiaoqiang Dan, Fei Liu, Chengke Wang, Sheng Yuan, Wenyuan Lv, and Qikai Xie. Effectively scheduling computational graphs of deep neural networks toward their {Domain-Specific} accelerators. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 719–737, 2023.

[199] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. Apollo: Automatic partition-based operator fusion through layer by layer optimization. *Proceedings of Machine Learning and Systems*, 4:1–19, 2022.

[200] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[201] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *ISCA*, pages 874–887, 2022.

[202] Size Zheng, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, and Yun Liang. Chimera: An analytical optimizing framework for effective compute-intensive operators fusion. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1113–1126. IEEE, 2023.

[203] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. Astitch: enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–373, 2022.