Cross-stack Optimizations for Sequence-based Models on GPUs

by

Suchita Pati

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: 02/08/2024

Final Oral Committee:

Matthew D. Sinclair (Advisor), Assistant Professor, Computer Sciences
Mikko Lipasti, Professor, Electrical & Computer Engineering
Joshua San Miguel, Assistant Professor, Electrical & Computer Engineering
Shivaram Venkatraman, Assistant Professor, Computer Sciences
Shaizeen Aga, Principal Member of Technical Staff, AMD Research & Advanced Development

To Mummy, Papa, and Chima for their love, support, and sacrifice.

ACKNOWLEDGMENTS

There have been many people instrumental in making this PhD research successful. First and foremost, I would like to sincerely thank my advisor Prof. Matthew D. Sinclair for guiding and supporting me through graduate school. Matt mentored me from my early days as a master's student, encouraging me to pursue a PhD and providing me the freedom to follow my research interests. I have learned and grown in many ways beyond research under his guidance. His meticulous feedback on papers sharpened my writing over the years, his patience through the numerous practice sessions helped me improve my presentation abilities, and our many simulator debugging sessions made me a patient developer. His passion to teach, consistent availability and willingness to help are also things I will always admire. And most importantly, his understanding and support during the challenges encountered along my PhD journey has been key to its successful completion.

I would also like to express my gratitude to my collaborators and mentors, Dr. Shaizeen Aga and Dr. Nuwan Jayasena. The Young Architect workshop Shaizeen co-organized introduced me to the broader architecture community, which played a crucial role in my decision to pursue a PhD. The internship opportunity with Nuwan and Shaizeen at AMD Research during my early PhD days was quite productive and their guidance has been pivotal in shaping my research ever since. Learning about near-memory computing from the experts, and integrating it into my PhD work, has been an enriching experience. Their invaluable industry insights have also ensured this dissertation's relevance and impact. I am also fortunate to have worked with Dr. Mahzabeen Islam. Her attention to detail was immensely helpful towards the final stages of my PhD.

I would like to express my deepest appreciation to other members of my committee – Prof. Mikko Lipasti, Prof. Joshua San Miguel and Prof.

Shivaram Venkataraman. Their feedback and questions during my preliminary exam and defense helped improve the research and thesis. I am also grateful to Mikko and Josh for their enjoyable computer architecture courses (752 and 757, respectively) as well as for their recommendations that helped me continue on as a PhD student. And Shiv's insightful comments on machine learning and systems helped made this thesis stronger.

Next, I thank all members of the HAL Research group including Kyle Roarty, Preyesh Dalmia, Rohan Mahapatra, Reese Kuper, Rajesh Shashi Kumar, Rutwik Jain, Brandon Tran, Tanmay Anand, Vishnu Ramadas, and many others. I appreciate their feedback on my research during the various group meetings and the opportunity to learn from the diverse set of projects they worked on. Outside the group, I would like to thank all the exceptional seniors and peers across the architecture labs at UW-Madison, including Gokul Subramanian Ravi, Swapnil Haria, Pratyush Mahapatra, Di Wu, Shyam Murthy, Abhishek Bhattacharyya, and many others. I thoroughly enjoyed our discussions and appreciated their advice at different stages of graduate school. Overall, it was wonderful to be part of an incredible architecture research community.

A PhD wasn't even on my radar until I met my mentor and manager at AMD India, Dr. Kanishka Lahiri. He guided me as I began my career as an engineer fresh out of undergrad, introduced me to architecture research, and supported me as I explored the idea of graduate school. His faith in my abilities as a researcher and his guidance have helped shape my career in ways I never imagined. For that, I will always be grateful to him. I would also like to thank Dr. John Kalamatianos for mentoring me as a research intern during my first year of grad school and recommending for a PhD.

Many others have helped me during my PhD. I would like to take this opportunity to express my appreciation to all of the CS, CSL and ISS staff for their help and support. Specifically, Angela Thorp and Stacey Sykes for

helping with all the administrative processes related to graduate school. I would like to thank Dr. Gabe Loh and Dr. Ganesh Dasika, and others who were part of the AMD Author Program process for helping improve the quality of the papers.

I have been fortunate to have been surrounded with extremely supportive friends during all the highs and lows of the PhD. Gokul Subramanian Ravi's work ethic always inspired me. He kept me motivated and confident, helping me push through tough times. I am thankful to Varsha Swaminathan for being the most caring apartment-mate I could ask for. She helped make our small apartment a home away from home. I am thankful for having constant cheerleaders Shuvam Gupta and Archana Dhyani. Their energy kept me going on long work days. I am thankful to Aaditya Chandrasekhar, Anjali, and Anthony Rebello for all their help and for always being available for a chat. I also thank Aishwarya Rengarajan, Ashwin Varadarajan, Nikhil Agarwal, Pratyush Mahapatra, Yash Trivedi, Ragini Rathore, Vigneshwar Ravisankar, and Nivvetha Srinivasan for supporting me in many ways through graduate school and making life in Madison wonderful.

I cannot begin to express my thanks to my wonderful family. My late grandparents Ram Narayan Pujari, Damodar Pati, and Kalyani Pati supported my decision to pursue a PhD and have been instrumental in all my career choices. I wish they were here to see this day. My grandmother, Subasini Pujari, continues to offer unwavering support to this day. I'm grateful to my cousin, Abhijit Hota, for being my family away from home, and helping me settle in Madison and Bay Area. Additionally, I appreciate the support of my aunt, uncle and cousin, Dr. Sujata Pujari, Dr. Datteswar Hota, and Aditi Hota, for their help in smoothly navigating through medical challenges.

I do not have enough words to thank my parents, Susmita Pati and Chitta Ranjan Pati. They have always supported my decisions, despite

their own expectations and desires, often defying traditional social norms. They have also been a source of motivation. My mum with her energy, enthusiasm, and determination to get things done, served as a boost even through phone calls from thousands of miles away. It helped me get through those endless nights of deadlines. Dad set an example very early in my life that anything is possible with sincerity and hard work. These values have proven invaluable during my PhD. His keen interest in my research, and wisdom were also very helpful, despite his experience in a vastly different domain. Finally, this acknowledgement is incomplete without mentioning my elder sister and brother-in-law, Rachita Pati and Raghavender Yadati. Rachita has been my go-to person in times of crisis, not only because she can help me the best, but also because conversations with her are often filled with laughter. She has also made defying norms easy for me. Finally, a big thank you to Raghav for bringing joy into our lives and for filling the void of my absence back home. My family's encouragement, love, and most importantly, faith in me have helped me through rejections, and at some of my lowest points during the PhD. For this, I will be forever indebted to them.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Advancements in the field of machine learning has made deep neural networks (DNNs) ubiquitous. Their application in the domain of natural language processing (NLP) with *sequence-based models* (models which process sequence of data) has been particularly remarkable and has led to powerful tools such as ChatGPT. This has been a result of advanced model architectures (e.g., Transformers), improved training techniques, as well as the transformative change in the scale of both model and dataset size. Training such models however can be extremely computationally expensive; some of the largest sequence-based models today take over a month to train on ~4500 GPUs, which are the primary workhorses for DNNs. Thus, this dissertation attempts to reduce these costs by identifying and leveraging cross-stack opportunities to maximize the models' use of GPU resources.

Identifying such opportunities requires accurate profiling and characterization of these models. However this is challenging due to long executions times of model training, constantly evolving models, as well as large resource requirements (for large-scale distributed setups). Thus, we first devise a mechanism, *SeqPoint*, to create a short but representative execution profile from thousands of heterogeneous training iterations of sequence-based models. Next, we do a detailed characterization of Transformer models on GPUs. Our characterization focuses on algorithmic understanding of the model and their hardware implications, also including the impact of their ever-evolving behavior. Finally, we devise mechanisms to efficiently study several Transformer models in different types of distributed, multi-device setups (the de-facto setup in which they are trained/deployed today). Our characterization shows that while there has been considerable improvements done at each layer of the computing stack to improve these models, lack of information from other layers

prevents them to reach their maximum potential.

First, gradient descent weight updates of these models are often extremely memory-bound and can under-utilize modern accelerators. This is because weight updates require accessing a substantial amount of data, typically several times the size of the parameters themselves, and entail very few computations. We leverage this algorithmic understanding to offload weight updates to near-memory compute units while executing the remaining operations on the GPU compute units. By significantly reducing data movement and enabling very high bandwidth access to data, near-memory compute improves model efficiency and performance. Second, we find that even with extremely well tuned BLAS libraries, concurrently executing multiple matrix multiplications (GEMMs) seldom improves GPU throughput. This is because operator libraries are tuned offline assuming isolated execution. We devise GOLDYLOC, which selects GEMM kernels optimized for the global resources available during execution and minimizes resource contention during concurrent executions. It further introduces a dynamic logic to control the amount of concurrency for improved performance. Finally, our multi-GPU characterization reveals that often inter-device communication kernels in distributed setups are serialized with compute kernels, causing sub-optimal performance scaling and idle network/compute resources. To overcome this, we devise T3 which hides the communication cost by enabling fine-grained overlap of communication with their producer computations. This overlap is done transparently in hardware, minimizing programmer overheads and furthermore uses DMA engines and near-memory compute units for communication to reduce resource contention with the producer computations.

Overall, by providing detailed characterization of these increasingly important models and accelerating them via a tighter flow of information between the application, libraries and hardware, this thesis contributes to

the synergistic evolution of machine learning and systems which has been key to the rapid and disruptive advancements in machine learning.

1 INTRODUCTION

The field of Machine Learning (ML), particularly deep neural networks (DNNs), has played a transformative role in society, showcasing significant accuracy improvements across diverse tasks including speech recognition [282, 292], language modeling [41, 62, 226], machine translation [87, 273], multi-modal understanding [3, 39, 270], image classification [64, 88, 133, 149, 257, 267, 268], recommendations [179] and autonomous agents [150]. These improvements are attributed to advancements in model architectures [273], increase in model parameters or size as well as the scale of the datasets the models are trained on. This became especially true for *sequence*-based models or models which process sequence of information such as text, audio, and video. From the adoption of *recurrent* networks for language tasks to the emergence of *attention*-based Transformer models, alongside training techniques that enabled training on vast datasets, including the entire internet, these models have demonstrated applicability across multiple domains (vision, video) [38] and have paved the way for advancements in artificial general intelligence [238].

Consequently, sequence-based models have become a significant driver for future hardware requirements. While GPUs have been the primary computing platform for DNNs due to the strong combination of programmability, performance, and energy efficiency they offer, they have also undergone substantial enhancements. To meet the surge in the computational demand in training and deploying sequence-based models (due to scaling model size, datasets, and applicability), the prevailing strategy has been to increase the GPU's computational capabilities. This is done by increasing GPU cores (streaming multiprocessors or SMs for NVIDIA and compute units or CUs for AMD) and memory bandwidth. Furthermore, specialized hardware enhancements, including TensorCores [188], Matrix Core Engines [20], and Transformer engines [196] have been introduced.

These efforts have resulted in GPU FLOPS more than doubling with each generation [9, 21, 25, 188, 194, 200]. To further meet the computational demands, powerful nodes featuring multiple GPUs interconnected with high-bandwidth links have also emerged [26, 199]. This has enabled large-scale distributed setups of these models. For instance, some of the largest sequence-based models today train for over a month on approximately 4500 GPUs [263]. Given these increasingly large hardware resources availed by these models, the fundamental question is: do these models indeed utilize all these resources well? And if they do not, how can we improve these systems and make their executions efficient? *Thus, this dissertation answers these questions by identifying inefficient execution phases and leveraging cross-stack opportunities to improve sequence-based models' use of hardware resources while also accelerating them.*

In Chapters 3, 4 and 5 of the dissertation, we focus on the profiling and characterization of contemporary sequence-based models on GPUs to pinpoint inefficiencies in their execution. Compared to Convolutional Neural Networks (CNNs) and Multi-layer Perceptrons (MLPs), profiling sequence-based models' training executions can be challenging due to their input sequence length-dependent iterations and large datasets with varying input sequence lengths. To overcome this, we first devised a sampling mechanism called SeqPoint (summarized in Section 1.1.1 and detailed in Chapter 3) to enable quick and accurate profiling and characterization of recurrent-based models, the state-of-the-art sequence-based model at the time. We next also do a detailed characterization of Transformer models on a GPU, which succeeded recurrent networks in several sequence-based tasks (summarized in Section 1.1.2 and detailed in Chapter 4). Unlike prior studies, we take into account the several flavors of Transformer models that were introduced by studying the impact of evolving model parameters as well as training techniques. Finally, recognizing the exponential scaling of model size that followed, requiring different types of large-scale dis-

tributed setups, we also characterize multi-GPU execution of Transformer models. We overcome the challenges with exhaustively profiling them (in terms of time, effort and resource availability) by developing algorithmic and empirical strategies. These helped us evaluate the inter-GPU communication costs in executing very large as well as futuristic Transformer models (summarized in Section 1.1.3 and detailed in Chapter 5).

Overall, these analyses reveal three primary inefficient primitives prevalent in sequence-based models. First, not all model operations are GPU-amenable. Model executions consist of memory-bound weight update algorithms that require significant data movement between memory and GPU compute units (CUs) and leave CUs underutilized. Second, GPU-amenable matrix-multiply operations or GEMMs can also underutilize CUs and while sequence-based models have abundant opportunities to concurrently execute independent GEMMs, they seldom provide expected benefits. Finally, multi-GPU model executions have extended serialized inter-GPU communication phases which limit throughput scaling and result in idle compute resources. Notably, while these inefficient operators are prominently observed in sequence-based models, they represent fundamental primitives, although not always dominant, in most other DNNs as well. In addition, these studies reveal opportunities to leverage information from across the stack (application, libraries, runtime and hardware) to improve execution efficiencies.

In Chapters 6, 7 and 8 of the dissertation we mitigate the identified inefficiencies via cross-stack optimizations. First, instead of having entire end-to-end application offloaded to a single accelerator such as a GPU, we demonstrate the efficacy of using algorithm understanding to selectively offload memory-bound weight update algorithms to emerging compute-enhanced memories. This strategy improves efficiency of these memory-bound phases by leveraging the high-bandwidth access to data that 3D-stacked memories enable while also reducing data movement

between GPU compute units and memory (summarized in Section 1.2.1 and detailed in Chapter 6). Second, we address sub-optimal concurrent General Matrix Multiply (GEMM) executions on GPUs with GOLDYLOC (summarized in Section 1.2.2 and detailed in Chapter 7). Unlike current GPU systems, GOLDYLOC extends the GEMM library to include kernel implementations optimized for shared resource environments during concurrency. It additionally uses runtime information to both select the appropriate kernel implementation as well as to dynamically control the amount of concurrency to exploit. Together, these improve overall GPU resource utilization and throughput. Finally, we tackle the challenge of serialized communication in T3 (summarized in Section 1.2.3 and detailed in Chapter 8) by leveraging application understanding about their preceding (producer) operations and overlapping them with communication in a fine-grained manner. T3 uses a configurable hardware track and trigger mechanism to mitigate software complexities of interleaving. It further leverages near-memory computing and DMA engines to minimize resource contention arising from the overlap. This enables communication costs to be largely hidden, and improves the utilization of both compute units and inter-GPU links. Overall, these optimizations highlight the need for information flow between different layers of compute abstractions (application, libraries, runtime, and hardware) to improve GPU resource utilization. Below we provide a summary of each of the chapters:

1.1 Analysis of Sequence-based Models on GPUs

1.1.1 SeqPoint: Identifying Representative Iterations of Sequence-based Neural Networks

Detailed profiling and characterization of DNN training remains difficult as these applications often run for hours to days on real hardware. Prior works have exploited the iterative nature of DNNs to profile a few training iterations to represent the entire training run. While such a strategy is sound for networks like CNNs, where the nature of the computation is largely input independent, we observe in this work that this approach is sub-optimal for sequence-based neural networks (SQNNs) such as RNNs. The amount and nature of computations in SQNNs can vary for each input, resulting in heterogeneity across iterations. Thus, arbitrarily selecting a few iterations is insufficient to accurately summarize the behavior of the entire training run.

To tackle this challenge, we carefully study the factors that impact SQNN training iterations and identify *input sequence length* as the key determining factor for variations across iterations. We then use this observation to characterize all iterations of an SQNN training run (requiring no profiling or simulation of the application) and select representative iterations, which we term *SeqPoints*. We analyze two state-of-the-art SQNNs, DeepSpeech2 and Google's Neural Machine Translation (GNMT), and show that SeqPoints can represent their entire training runs accurately, resulting in geomean errors of only 0.11% and 0.53%, respectively, when projecting overall runtime and 0.13% and 1.50% when projecting speedups due to architectural changes. This high accuracy is achieved while reducing the time needed for profiling by 345× and 214× for the two networks compared to full training runs. As a result, SeqPoint can enable analysis

of SQNN training runs in mere minutes instead of hours or days.

1.1.2 Demystifying Transformers: System Design Implications

Transformer-based networks [273], a successor of RNNs, became the preferred algorithm for natural language processing. These networks, along with transfer learning, gave rise to models like the Bi-directional Encoder Representation from Transformer (BERT) [62], which marked a shift towards deeper knowledge transfer by applying massive pre-trained models to different tasks. Understanding Transformer models' underlying behaviors is vital to designing efficient accelerators for them. Thus, we study the computationally and time-intensive training phase of Transformer models and identify how its algorithmic behavior can guide future accelerator design. We focus on BERT and identify key operations which are worthy of attention in accelerator design. In particular, we focus on the manifestation, size, and arithmetic behavior of these operations which remain constant irrespective of hardware choice. To capture future Transformer trends, we also show and discuss implications of these behaviors as networks and inputs get larger. Moreover, we study the impact of key training techniques like distributed training, checkpointing, and mixed-precision training. The key takeaways from this analysis are:

- Optimizer updates are very memory intensive. Their runtime scales linearly with transformer layer count and quadratically with layer size and thus are important to optimize for.
- GEMMs dominate Transformer runtime but have heterogeneity. Some GEMMs are smaller and thus may not fully utilize accelerators and may also be memory-bound. GEMM proportion also increases with layer size.

- Non-GEMMs (add, multiply, scale, reduce) are memory-bound and a considerable proportion of runtime. Their proportion drops with increasing layer size as they scale only linearly with it (unlike GEMMs and updates, which are quadratic).
- Reducing precision makes optimizing memory-intensive operations crucial. At lower precision, GEMMs speed up more than non-GEMMs due to faster arithmetic and reduced memory traffic. Furthermore, updates use higher (FP32) precision data to maintain accuracy and remain unaffected.
- Tensor Slicing is bottlenecked by communication as the latter is serialized with computations. Its cost increases with device count.

1.1.3 Computation vs. Communication Scaling for Future Transformers on Future Hardware

Scaling of neural network models has increased the reliance on efficient distributed training techniques. Accordingly, like other distributed computing scenarios, it is important to understand *how compute and communication will scale relative to one another as models scale and hardware evolves?* A careful study which answers this question can better guide the design of future systems which can efficiently train future large models. Accordingly, we comprehensively analyze compute vs. communication (Comp-vs.-Comm) scaling for future Transformer models on future hardware, across multiple axes (algorithmic, empirical, hardware evolution).

We first perform an algorithmic analysis of compute and communication operations in Transformer models. Our algorithmic analysis shows that the complexity of compute operations is often higher than communication volume (data size). We call this *compute's edge* over communication. A compute-dominated execution profile is often a positive edge because compute (FLOPS) scaling has received considerably more attention than

communication (bandwidth) scaling, and often optimizations are employed to overlap communication with useful compute. Thus, compute's edge also helps hide communication costs. However, model scaling and memory capacity trends are stressing this edge.

We quantify this edge by empirically studying how Comp-vs.-Comm scales for future models on future hardware. This approach has several challenges, including requiring studying many model/hardware evolution scenarios. Our empirical strategy addresses these challenges by (a) designing controlled experiments (guided by our algorithmic analysis), (b) executing only certain regions-of-interest (ROIs), and (c) using operator-level models which we show accurately (<15% error) project operator runtime trends for varying hyperparameters. These enable us to study hundreds of future models/hardware scenarios at 2100× lower profiling costs. Our experiments show that communication will be a significant portion (40-75%) of runtime as models and hardware evolve. Moreover, communication that is often hidden by overlapped computation in today's models cannot be hidden in future, larger models. Overall, this work highlights communication's increasingly large role as models scale, discusses promising techniques to potentially tackle communication, and discusses how our analysis influences their potential improvements.

1.2 Improving Sequence-based Models' Efficiency on GPUs

1.2.1 Processing Optimizer Updates in Memory

Our characterization reveals how memory-bound gradient descent updates of billions of Transformer parameters can under-utilize modern accelerators like GPU. To overcome this, we *offload updates to near-memory compute units* [215] while still executing the compute-bound GEMMs on the

GPUs. Mapping a sequence of operations to memory requires few expensive synchronizations with GPU compute units, and provides increased data access bandwidth along with concurrency of multiple DRAM banks. Thus, it accelerates weight updates, by 3.8× for a popular Transformer, BERT. Finally, it considerably reduces (~13×) expensive data movement from DRAM to GPU compute units.

1.2.2 GOLDYLOC: Global Optimizations and Light-weight Dynamic Logic for Concurrency

Concurrently executing multiple operations can help improve the device's compute utilization, especially with small and low-utilizing computations observed in our characterization of sequence-based networks. However, effectively harnessing it on GPUs for important primitives such as general matrix multiplications (GEMMs) remains challenging. GPU libraries exhaustively optimize kernel implementations for performance/efficiency of key operators like GEMMs. However, this tuning assumes the availability of all GPU resources, assuming each kernel executes in *isolation* and can utilize all GPU resources. This approach is highly efficient when kernels execute in isolation, but causes slowdowns when executed concurrently with other operators due to resource sharing and contention. Moreover, concurrency can only be *statically* exposed and controlled from within an application. This does not take into consideration the dynamic execution environment (e.g., varying input size, multiple applications) – often exacerbating contention. These issues limit performance benefits from concurrently executing independent operations.

Accordingly, we propose GOLDYLOC. GOLDYLOC augments kernel tuning to identify efficient kernels for both isolation and *global* resource environments resulting from varying degrees of concurrent execution. To find the latter GOLDYLOC tunes kernels offline with *resource constraints* which emulates various shared resource environments. Similar to the

baseline where kernels have unique properties per GEMM input, tuning for concurrency also makes unique trade-offs per input to efficiently share resources. Moreover, we also augment the GPU's command processor (CP) to *dynamically* control concurrency using a predictor (trained offline) which selects the type and degree of concurrent GEMMs to execute given the available independent GEMMs and their inputs. This includes detecting when sequential execution is preferred. Overall, GOLDYLOC improves performance of concurrent GEMMs on real hardware by up to 2.5× (43% geomean per workload) over sequential execution and up to 2× (18% geomean per workload) over statically controlled and isolated tuned concurrent executions on GPUs.

1.2.3 T3: Transparent Tracking & Triggering for Fine-grained Overlap of Compute & Collectives

Extended phases of inter-device communication can reduce the scaling efficiency of DNNs in large distributed setups. While some distributed techniques can overlap, and thus, hide this communication with independent computations, techniques such as Tensor Parallelism (TP) inherently serialize communication with model execution. One approach to hide this serialized communication is to interleave it with the producer of the communicated data (usually a GEMM) in a *fine-grained* manner. However, enabling this fine-grained overlap in current systems either requires expensive fine-grained synchronization [107] or changes to GEMM kernels which can be disruptive to GPU software infrastructure [278]. Furthermore, overlapped compute and communication contend for both compute units and memory bandwidth, reducing overlap's efficacy [107, 278].

To overcome these challenges, we propose T3 which applies hardware-software co-design to transparently overlap serialized communication while minimizing resource contention with compute. T3 *transparently fuses* producer operations with the subsequent communication via a sim-

ple configuration of the producer's output address space to initiate communication directly on the producer's store, requiring minimal application changes. At the hardware level, T3 uses uses a light-weight and programmable hardware *tracker* to track the producer/communication progress and *trigger* communication using pre-programmed DMA commands, requiring no additional GPU compute resources. It further uses *compute-enhanced memories* [125, 145] to atomically update memory locations on stores thus freeing GPU compute units and reducing memory traffic from communication's attendant compute. As a result, T3 reduces resource contention, and efficiently overlaps serialized communication with computation. For important Transformer models like T-NLG, T3 speeds up communication-heavy sublayers by 30% geomean (max 47%) and reduces data movement by 22% geomean (max 36%). Furthermore, T3's benefits persist as models scale: geomean 29% for sublayers in ~500-billion parameter models, PALM and MT-NLG.

1.3 Contributions

The main contributions of this dissertation are:

- **Fast & Accurate Profiling:** we devise a systematic mechanism to profile sequence-based models which otherwise took several days to run on native hardware. By identifying few representative training iterations to profile, SeqPoint made fast and accurate characterization of these models possible.

- **Identified Inefficiencies in Transformer Executions on GPUs:** We provide a detailed characterization of the, then emerging, Transformer networks to identify and expose inefficiencies in their execution on state-of-the-art GPUs. This also includes impact of varying hyperparameters and training techniques to incorporate the ever-

evolving field of ML. The observations from this study inspired several other pieces of the dissertation.

- **Operator-level Details for Accurate Evaluations:** Our study of Transformers also provides details on the manifestation and size of all its constituent operators, which were instrumental in our evaluations of subsequent proposals. As an example, it helped us derive the size of matrix-multiplication operations (GEMMs) which were used to evaluate both GOLDYLOC and T3. Absence of such a study can cause works to build accelerators with matrix-vector engines for models which actually perform matrix-matrix operations [89].

- **Algorithmic Communication Costs**: We devise a mechanism to algorithmically project the relative importance of computation and communication in large-scale distributed setups. This provides a system-agnostic analysis of communication costs, as well as helps understand how model evolution influences them.

- **Project Communication Costs in Large Distributed Setups**: We devise a strategy for a practical empirical analysis of several Transformer models on large-scale distributed setups and demonstrate how inter-device communication will play an increasingly large role as models scale.

- **Appropriate operator-accelerator mapping**: We showcase the potential of offloading memory-bound weight update algorithm to near-memory compute units while still executing compute-bound GEMM on GPUs.

- **Efficient Concurrent GEMM Execution on GPUs**: We show how important primitives like GEMMs, that GPUs heavily optimize for, can underutilize resources even when run concurrently on a single GPU. The key reason for these inefficiencies is the indifference of GPU

GEMM libraries toward the execution environment as well as and inability for GPU runtimes to control the amount of concurrency. We develop GOLDYLOC which provides kernel implementations that are aware of global resources in shared environments. It also allows GPU scheduler to control the number of concurrent kernels based on dynamic information about the size and count of independent operations. This tighter flow of information between the hardware, runtime and libraries improves overall throughput and efficiency.

- **Efficient Fine-grained Overlap of Communication with Compute**: We show how serialized communication in distributed DNN setups can leave GPUs idle for an elongated period of time, causing poor application throughput scaling and wasted cycles in datacenters. We propose T3 which efficiently fuses and overlaps computations with communication in a fine-grained manner. Furthermore, it does so without disrupting complex software infrastructure, a key challenge in prior works. Thus by leveraging the producer-consumer relationship in the algorithm, we improve overall efficiency of a distributed setup.

- **Support for DNNs in GPU Architecture Simulators**: We also extended and released support for the widely-used, popular GPU simulator, GPGPU-Sim [36], to run DNNs [147].

1.4 Outline

The dissertation is organized as follows: in Chapter 2, we provide all the required background for this dissertation. Chapters 3, 4, and 5 detail our work on the profiling mechanism, characterization of NLP models, and analysis of communication in distributed multi-GPU setups. The key takeaways from this characterization motivate our proposals in the next three chapters. Chapter 6 discusses accelerating memory-bound weight

updates using near-memory computing. Chapter 7 provides details of our work on efficient concurrent execution of GEMMs (GOLDYLOC). Chapter 8 demonstrates our work on fine-grained interleaving of compute and communication using T3 for distributed ML. Finally, Chapter 9 summarizes the document, and provides reflections and future work based on this dissertation.

2 BACKGROUND

This chapter covers the background for the entire thesis. Relevant background for each chapter is specified below:

- **SeqPoint:** DNN Training (Section 2.1.1), Batching & Minibatches (Section 2.1.3), RNNs: Recurrence-based Networks (Section 2.2.1).
- **Characterizing Transformers:** Transfer Learning: Pre-training & Fine-tuning (Section 2.1.2), Transformers (Section 2.2.2), Distributed Computing for DNNs (Section 2.3), Gradient Descent Optimizers (Section 2.4.2).
- **Computation vs. Communication Analysis:** Transformers (Section 2.2.2), Distributed Computing for DNNs (Section 2.3), Collective Communication (Section 2.4.3)
- **Near-memory Optimizer Updates:** Gradient Descent Optimizers (Section 2.4.2), DRAM (Section 2.5)
- **GOLDYLOC:** RNNs: Recurrence-based Networks (Section 2.2.1), Transformers (Section 2.2.2), GEMMs: GEneral Matrix Multiplications (Section 2.4.1)
- **T3:** Transformers (Section 2.2.2), Distributed Computing for DNNs (Section 2.3), Collective Communication (Section 2.4.3)

2.1 Deep Neural Networks (DNN)

2.1.1 DNN Training

Most DNNs have large numbers of tunable parameters (or *weights*) that are learned using large amounts of data during a *training* phase. Once

trained, the network can be deployed to operate on new inputs, which is referred to as *inference*. During training, an input (e.g., an image) is fed into the network and is propagated forward through a collection of layers that form the network until an output is generated. Each layer takes the output of the previous layer, computes on it and feeds its result or *activation* to the next layer. At the end of this *forward propagation*, the generated output is compared to the known correct output to compute an error. The error generated in the forward pass is propagated backwards through the layers of the network during a *backward propagation* of the network. During this back propagation, each layer generates the *input gradient* (or error) to propagate to the previous layer, as well as a *weight gradient* that is used to update the tunable parameters of the layer (to minimize the error).

2.1.2 Transfer Learning: Pre-training & Fine-tuning

In *transfer learning*, a model trained for a particular task is reused for different tasks. Transfer learning has been widely adopted in the language domain. Language models have a long *pre-training* phase where the model learns the language using large unlabeled datasets (e.g., Wikipedia), independent of any target task. Once pre-trained, they are *fine-tuned* during which they are trained on a labeled dataset for a specific task with minimal model changes. As an example, a single pre-trained model can be tuned independently for 11 different tasks [62].

2.1.3 Batching & Minibatches

To improve hardware utilization (particularly on parallel platforms such as GPUs) and to improve the stability of convergence, the training phase is often performed in groups of inputs known as *minibatches* or, simply, *batches*. Figure 2.1(top) illustrates an example of forming batches of size four for a text-based training set. The number of inputs in a batch is referred to as the batch size. In batch-based training, all inputs of a batch

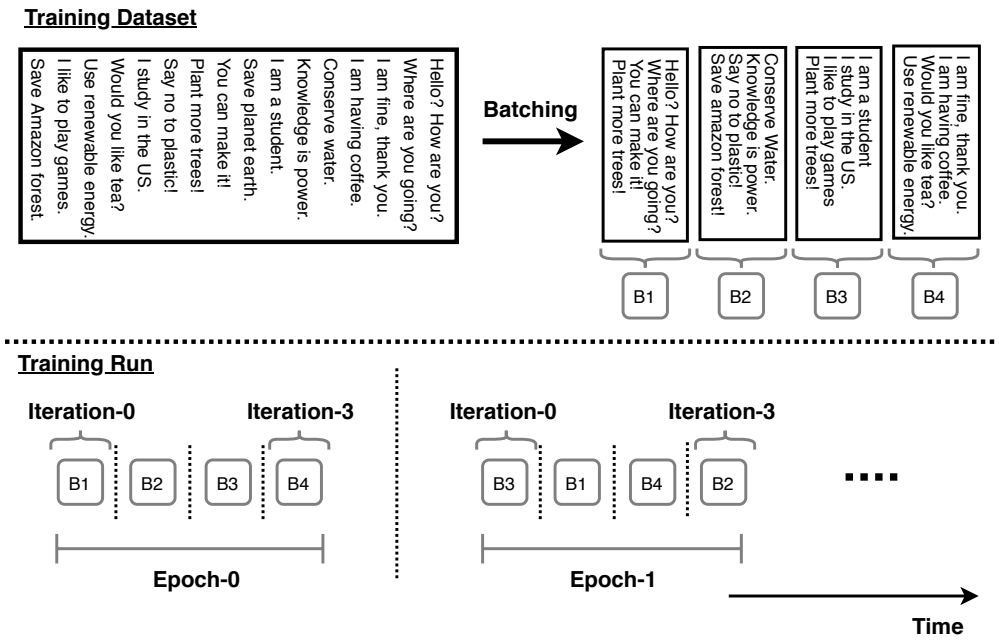


Figure 2.1: Training phase of sequence-based models.

perform the forward traversal using the same set of weights, and then the backward pass is performed for all inputs of the batch, computing the corresponding errors and updating the weights. The forward and backward traversal of the input through the network is referred to as an *iteration*. A set of iterations making up a single pass through the entire dataset is referred to as an *epoch*, as illustrated by Figure 2.1(bottom). Training of a network typically consists of multiple epochs (i.e., multiple passes over the entire training set) until a convergence criterion is met.

2.2 Sequence-based Networks

Sequence-based networks are a class of DNNs which, unlike other DNNs (e.g., CNNs), process a sequence of information (e.g., a sentence) to learn the tokens (e.g., words) and relationship between them (e.g., context of a sentence). This makes them a good fit for natural language tasks including

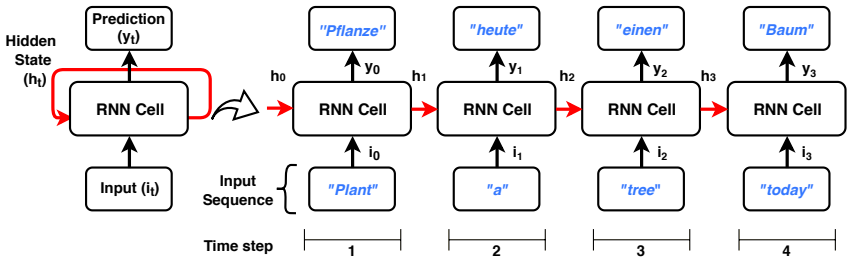


Figure 2.2: Left: A single layer of an RNN. Right: Unrolled RNN processing an input sequence.

language modeling [62], speech recognition [90], and translation [273]. There are two main types of sequence-based networks, recurrent and Transformer networks that we detail in Sections 2.2.1 and 2.2.2, respectively.

2.2.1 RNNs: Recurrence-based Networks

Recurrent neural networks (RNNs) are a class of sequence-based networks which process sequences. As shown in Figure 2.2, a given layer in an RNN processes each input token (e.g., word in a sentence) or i_t a time. Similar to other DNNs, it propagates the output to the next layer. However, unlike other DNNs, it also produces a hidden state h_{t-1} which is updated each time a token is processed as shown by the equation below:

$$h_t = f\,(W\,i_t + U\,h_{t-1} + b\,)$$

where W is the input-hidden weight matrix and U is the recurrent weight matrix and b is a bias term. Each RNN layer thus loops through all tokens in the input sequence, updating the hidden state which allow them to capture and remember information (e.g., context of a sentence) across multiple tokens. This however introduces sequential dependency between token processing within a layer. The number of tokens (or times-steps) represents the *sequence length* (SL) of the input to the RNN. Finally, RNNs can be of different types depending on the layer type. Vanilla, Long Short Term Memory (LSTM) [94], and Gated Recurrent Unit (GRU) [52] are

three widely-used types of RNNs, which differ in how they process input and hidden states as well as the parameters they store. The iterative processing of one token at a time makes them well suited to real-time sequence processing tasks such as speech recognition.

RNN's input and hidden state processing manifest as matrix multiplications (GEMMs). While hidden state processing across tokens within a layer have dependencies and are processed independently, input processing of tokens may be combined into a single or few GEMMs [31].

2.2.2 Transformers

Transformers [273] are another class of sequence-based models which have increasingly replaced RNNs to become the general-purpose architecture for a wide range of tasks and domains. Recent work has shown that many different modalities are using Transformers as their base model (e.g., 41% of text, 22% of image) [38]. The basic building block of these Transformers is an *encoder* or *decoder* layer which is repeated multiple times (Figure 2.3(a)). They also have an input embedding layer that provides the first layer with an input representation/vector of token as well as an output classification layer. As shown in Figure 2.3(b), each encoder/decoder block contains an attention layer and a fully connected (FC) layer, both of which are followed by a residual connection and layer normalization. The encoder and decoder blocks are similar, except the decoder's attention GEMM input is masked to consider only past tokens, which causes different computational inference behavior but does not affect training.

The evolution of Transformer models has largely focused on changing Transformer block type (encoder vs. decoder, or both), increasing the number of Transformer blocks, and/or increasing layer widths. This is true for all Transformer models; starting with the model BERT [62] (with 0.3 billion parameters), to its most recent successor, MT-NLG (with 540

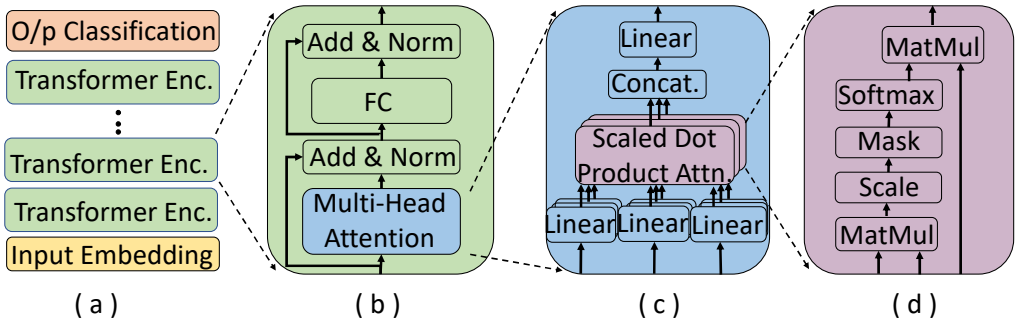


Figure 2.3: BERT hierarchical model breakdown.

billion parameters), and many others in between [41, 58, 138, 154, 226, 256, 266, 284]. Thus, while Transformer models have become larger with different hyperparameters, their fundamental computational components are largely the same. Therefore, throughout this dissertation, we use BERT as our baseline model and change its hyperparameters to study/evaluate larger Transformer models.

2.2.2.1 Attention

Attention (Figure 2.3(c, d)) is an essential component of Transformer-based models (e.g., BERT) described in Section 2.2.2. Given an input sequence, attention networks output a representation of the sequence such that each output token of the sequence is encoded with *weighted* information from all (or a subset, for masked attention used by decoder layers) other tokens in the sequence. This all-to-all encoding of information enables attention to process all tokens independently (unlike sequential RNNs), but also quadratically increases computations with increasing length of the input sequence.

2.2.2.2 Transformers Layer Manifestations

The attention sub-layer and fully connected (FC) sub-layer (as shown in Figure 2.4(a)) manifest as matrix multiplication operations (GEMMs). The residual connections and layer normalizations which manifest as

element-wise operations and are often fused [66, 72, 264, 275] with the GEMMs. As shown in Figure 2.4(b), these GEMMs entail multiplication of layers' weight matrices by an input matrix (with each vector representing an input token). During training, the input matrices contain multiple tokens from one or more (if batched) input sequence(s). During inference, there are two execution phases: a *prompt* phase to process all tokens in the input sequence(s) and a *token generation* phase to iteratively process and generate one token at a time for each input sequence [211]. The prompt phase operations are similar to those in training, while the generation phase has GEMMs with small input matrices or matrix-vector operations (GEMVs) if there is no batching.

2.2.2.3 Transformer Hyperparamters

Transformer models are defined by several hyperparameters that we use throughout the dissertation. *Layer count* denotes the number of Transformer encoder/decoder layers in the model. The *hidden dimension* of an attention sub-layer is the layer width and is usually same as the *embedding size* which is the size of each input token/vector. *Intermediate dimension* is the layer width of the feed-forward, fully-connected sub-layer, and is usually 4× the hidden dimension. Finally, the inputs to a model are dictated by the *batch-size* as well as the length of the inputs, *sequence length* (described in Section 2.2.1).

2.3 Distributed Computing for DNNs

Most Transformer models' memory capacity requirements exceed a single device. Thus, they employ distributed techniques and use multiple accelerators (e.g., GPUs) collaboratively. Furthermore, the aggregate computational capacity of multiple devices also accelerates training by enabling the processing of large input datasets in parallel. Thus, since

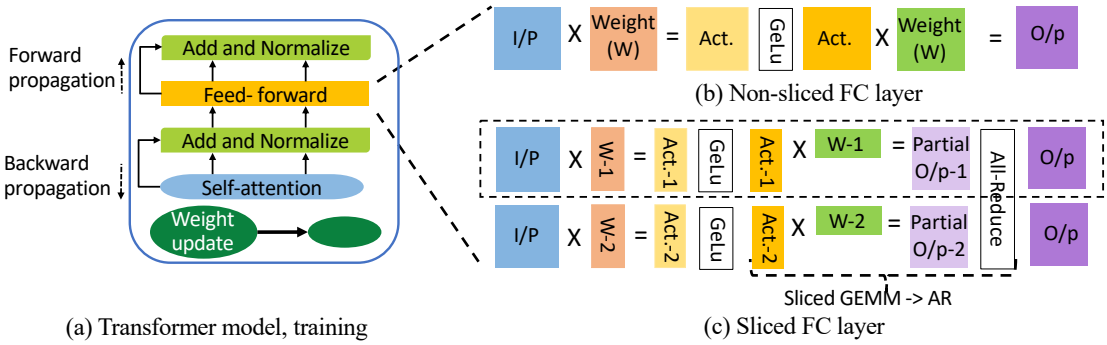


Figure 2.4: (a) Transformer (b) Fully-connected (FC) layer (c) Tensor-sliced FC layer with all-Reduce on the critical path.

Transformers and their datasets (usually large corpora of unlabeled text) have increased by several orders of magnitude in size, distributed techniques are often mandatory and increasingly require many devices. This scaling will only increase for future models.

2.3.1 Distributed Techniques & Associated Communication

DNNs, and specifically, Transformers employ many distributed techniques, each with associated *communication* between devices. *Data parallelism* (DP) trains model replicas on multiple devices, each on a disjoint set of the dataset, and requires communication and reduction of layer gradients across all devices every iteration. *Tensor parallelism* (TP) [256] and *pipeline parallelism* (e.g., GPipe) [99] are two types of *model parallelism* which slice a single model across multiple devices. While the former slices each layer requiring activations to be communicated and reduced across devices, the latter partitions the model layer-wise requiring peer-to-peer transfer of activations. ZeRO-based optimizations [230] also slice model weights across devices or offload them to slower but larger (e.g., CPU) memo-

ries, and require the corresponding weights to be gathered before layer executions. Finally expert parallelism [128] partitions mixture-of-expert (MoE) models [69, 228] such that each device hosts a single expert and requires exchange of input data based on input-to-expert mapping. In this dissertation, we focus on DP and TP, two of the most effective and widely adopted distributed techniques described in detail in Sections 2.3.2 and 2.3.3.

2.3.2 Data Parallelism

The most common and straightforward distributed ML technique is ***data parallelism***, in which the model is replicated on multipled (D) devices, with the input dataset partitioned amongst them. Each device iterates over its own dataset (using a mini-batch of b) and trains its model while synchronizing with all the other devices every iteration.[1] During synchronization, local gradients from all devices are averaged and re-distributed using an *all-reduce* collective and each model updates its parameters using these accumulated gradients. This enables large mini-batch ($D * b$) training, otherwise not feasible with a single device's memory capacity.

2.3.3 Tensor Parallelism

Tensor Parallelism (TP) [256] effectively increases the memory capacity available to a model by splitting the model across M devices (illustrated in Figure 2.4(b) & (c)). It splits a model layer (or tensor) across devices such that each device holds and thus operates on a subset of layer parameters. This slicing causes each device to generate only a partial layer activation (and error) during training's forward (and backward) passes, which require an all-reduce to generate the final layer output (Figure 2.4(c)).

[1] This does not hold for asynchronous training, which converts fine-grained synchronization into data accesses but may increase convergence time [60].

Furthermore, a layer's forward and backward executions are dependent on another layer's all-reduce of activations and errors.

2.4 Important DNN Operations

This section describes some of the important DNN primitives that dominate model execution as we show in Chapters 4 and 5.

2.4.1 GEMMs: GEneral Matrix Multiplications

2.4.1.1 GEMM's dominance

A prominent computation that GPUs accelerate are highly parallelizable GEMM operations. Most of a DNN's execution manifest as GEMMs [89, 216, 225]. While networks also manifest other operations, including element-wise adds and multiplies, activation functions, and layer normalization [33], they are often fused with preceding operations (commonly with GEMMs) using kernel fusion [66, 72, 264, 275] and tensor contractions [1, 126, 127, 180, 255] to avoid redundant memory traffic and reduce kernel launch overheads. Figure 2.5(a) shows a common DNN setup: DNNs have a series of layers, each of which executes as a GEMM between the input and the layer's weight matrix.

2.4.1.2 GEMM Operation

As shown in Figure 2.5(b), a GEMM multiplies two input tensors A and B of size MxK and NxK, respectively, to generate an output tensor C of size MxN. This involves $2 * M * N * K$ floating point multiplies and adds. The values of M, N and K are usually dictated by model hyperparameters such as layer width, batch-size, and/or input length (sequence length). Additionally, the input tensors may be used transposed or non-transposed or both (e.g., transposed in forward propagation but non-transposed in

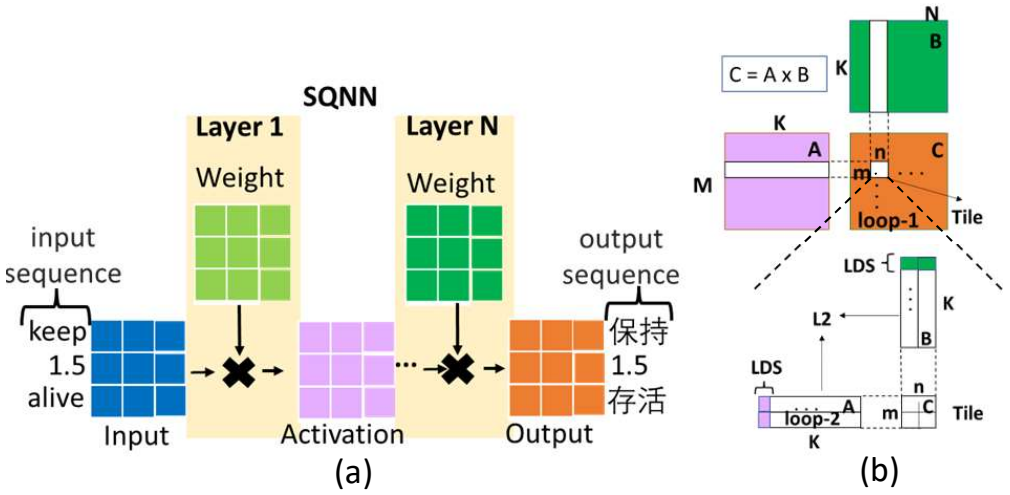


Figure 2.5: (a) Toy DNN GEMM computation. (b) High-level GEMM implementation on a GPU.

backprop). We represent the transpose of A and B input tensors by T1, T2 (e.g., 1,0 implies only one of them is transposed).

2.4.1.3 GEMM GPU Implementation

In GPU GEMM implementations C is often blocked/tiled (*Tile* in Figure 2.5(b)) with each work group (WG) usually responsible for a single tile (loop 1). Each thread in the WG multiplies and accumulates a row(s) with its respective column(s) within the innermost loop (loop 2). These threads often leverage fast on-chip shared memory or local data share (LDS) to store row/column data. Several optimizations are usually applied, including executing a subset of WGs at a time (which impacts cache reuse), prefetching data from memory to the LDS, and coalescing. Unlike other operations, applying these optimizations make GPU GEMM implementations quite complex, with hundreds of tunable features per size/transpose combination. Thus, to improve performance, vendors rigorously tune implementations for GEMMs of different sizes, corresponding to different layer types or parameters [23].

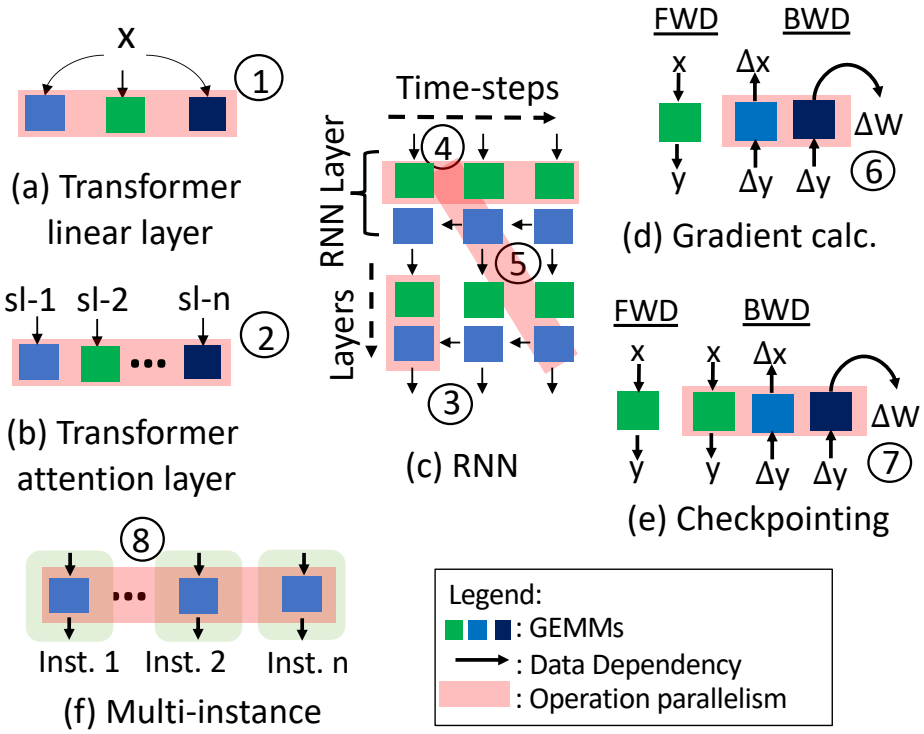


Figure 2.6: ML algorithms with independent operations.

2.4.1.4 Important DNNs with GEMMs

GEMMs are prominent in both RNNs and Transformers but differ in size and therefore, properties. RNNs process one token of the input sequence at a time [52, 94, 248]. The token processing manifest as one or more GEMM(s) and the sequential nature of the algorithm makes the GEMM's input tensor (in Figure 2.5) small, with one of the dimensions equal to the input batch size. Transformers use attention layers [34, 273] to represent a token as the weighted sum of all the input's other input tokens. Thus, Transformers layers process all tokens of an input sequence in parallel as a single operation that manifests as a GEMM. However, each input in a batch must be processed independently, as a separate GEMM, in the attention layer.

DNN Model / OP Type	Intra-model	Backprop	Checkpointing	Multi-instance
Transformers	✓ ①,②	✓ ⑥	✓ ⑦	✓ ⑧
RNNs	✓ ③,④,⑤	✓ ⑥	✓ ⑦	✓ ⑧
CNNs	X	✓ ⑥	✓ ⑦	✓ ⑧
Recommendation	X	✓ ⑥	✓ ⑦	✓ ⑧
Other DNNs	Varies	✓⑥	✓ ⑦	✓ ⑧

Table 2.1: Concurrency opportunities in DNNs; the circled numbers refer to Figure 2.6.

2.4.1.5 Opportunities for GEMM concurrency in DNNs

As shown in Figure 2.6 and Table 2.1, DNNs possess considerable operation parallelism from their model architecture: independent query/key/value generation in the linear layers, and independent (batched) attention computations for unique sequence length (SL) inputs in Transformers (① and ② in Figure 2.6), respectively. Similarly, independent input processing in the time dimension and hidden state processing across layers in RNNs introduce operation parallelism (③, ④, ⑤). Training algorithms also have additional parallelism opportunities that apply to all DNNs (e.g., CNNs, Recommendation) as highlighted in Table 2.1. These include independent weight and input gradient calculations during back-propagation (⑥) and activation recomputing due to checkpointing (⑦). Finally, while not applicable during training (due to large memory capacity requirements), multiple DNN inference instances (⑧) are deployed on the same GPU in production environments which provides additional concurrency opportunities [53, 54, 75, 115, 121, 189, 193, 271, 285].

2.4.2 Gradient Descent Optimizers

Gradient descent is the most common algorithm used to train neural networks. It minimizes an *objective function* (usually the loss) parameterized by the model's parameters. Models today use various algorithms to further

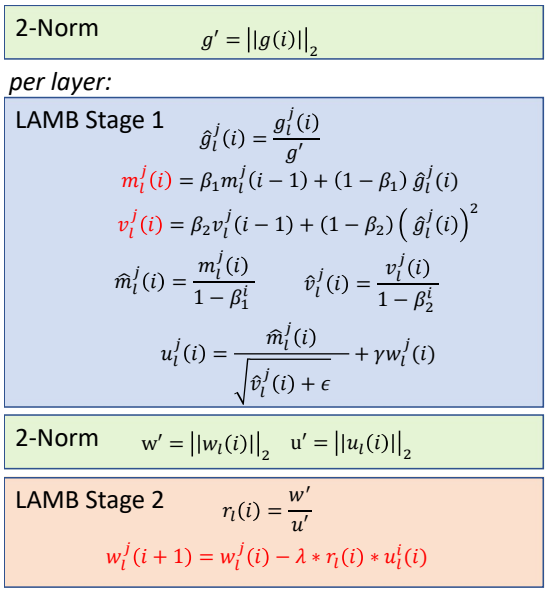


Figure 2.7: LAMB Optimizer Algorithm.

optimize gradient descent to converge faster. These optimizers help derive appropriate learning rates for different model parameters and for different training stages, but at the cost of additional optimizer parameters.

While DNNs are compatible with many different optimizers, they have recently used complex optimizers such as ADAM [129] and LAMB [287], which have proven effective for very large effective batch-sizes. Figure 2.7 details the LAMB algorithm, which updates the model parameters at the end of the model's forward and backward gradient calculations, once every (few) iteration(s). It is executed in two stages; in the first stage (LAMBStage1), it determines the update values (u) and learning rate multiplier using additional *momentum* (m) and *velocity* (v) parameters from the past iterations and gradients (g) of the current iteration. In the second stage (LAMBStage2), it updates the model weights (w) using these update and learning rate values. This pair of two stages are executed independently for every layer in the model, with each set accessing independent data (weights, gradients and optimizer parameters of the

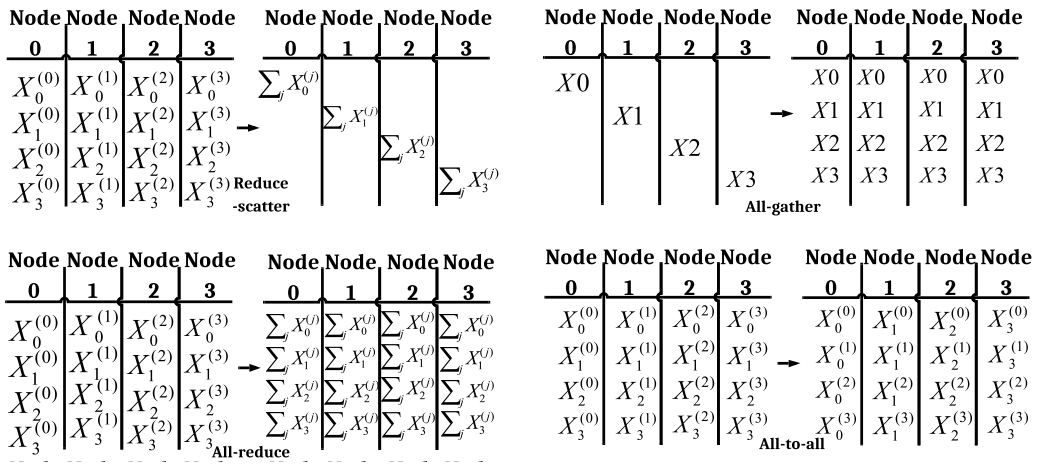


Figure 2.8: Common collective operations used in DNN execution [236].

corresponding layer).

2.4.3 Collective Communication

The communication patterns described in Section 2.3.1 are handled by *collectives* such as *reduce-scatter*, *all-reduce*, *all-gather*, *all-to-all*. As shown in Figure 2.8, each of these involve communication and at times, arithmetic operations (e.g., reduction) on the communicated data. In this dissertation, we focus on the all-reduce collective used in the widely adopted DP and TP setup described in Section 2.3.1. All-reduce can have multiple implementations optimized for different inter-connect topologies and distributed setups. One of the most bandwidth-optimal and commonly used implementation is the *ring* implementation.

2.4.3.1 All-Reduce & Ring Implementation

The all-reduce (AR) collective reduces (element-wise sums) arrays from each of the devices. Although there are multiple implementations of AR, one of the most bandwidth-efficient, and thus most commonly used, implementations is *ring-AR*. Ring-AR consists of a ring reduce-scatter

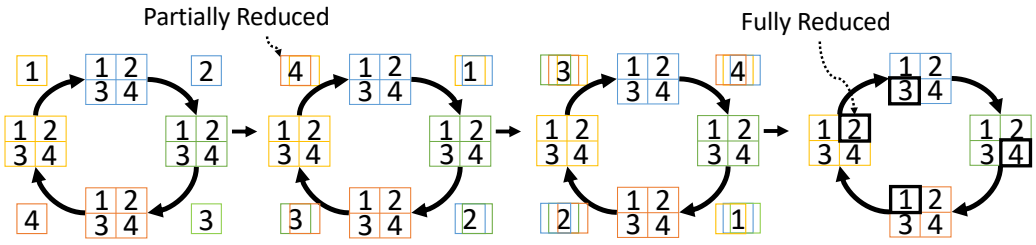


Figure 2.9: Ring implementation of reduce-scatter collective.

(ring-RS) followed by a ring all-gather (ring-AG). As shown in Figure 2.9, ring-RS is done in multiple steps. The arrays are chunked on each device, and during each step, all devices send their copy of a *unique* chunk to their neighbor in the ring. The devices then reduce their local copy of the chunk with the received copy and forward it to their neighbor in the next step. With N devices and the array chunked N ways, this process requires $N-1$ steps until each device has a completely reduced copy of one chunk. Ring-AG is similar but does not have reductions; it also requires $N-1$ steps until each device has all the reduced chunks. We use AR, RS, and AG to refer to the ring implementation of these collectives throughout the dissertation.

2.5 DRAM

2.5.1 Organization

DRAM is organized hierarchically as depicted in Figure 2.10 [6]. The lowest level of this hierarchy is a 2D array of memory cells and several of these are grouped into sub-arrays. The sense amplifiers of the 2D arrays collectively form a local row buffer of the sub-arrays. Several sub-arrays form banks, wherein sense-amplifiers at bank-level form the global row buffer. A typical DRAM chip contains multiple such banks.

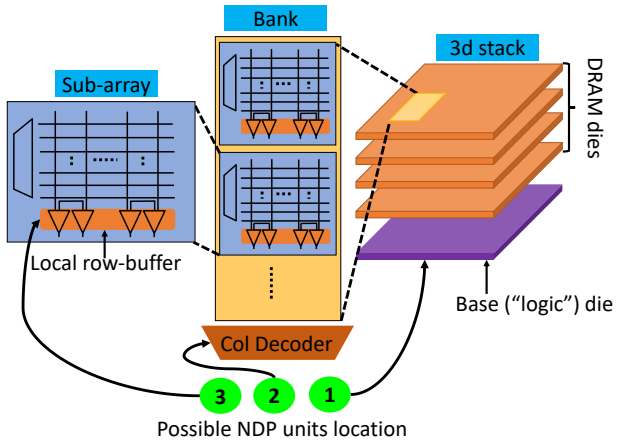


Figure 2.10: DRAM organization and locations for placing NMC units [6].

In mainstream DDR-based memory systems, a single dual-inline memory module (DIMM) consists of one or more sets of memory chips that are accessed in parallel (e.g., a 64-bit data bus may be served by four DRAM chips with 16-bit-wide, or x16 interfaces). Multiple such sets of chips on a DIMM are called ranks and share a single memory interface bus to the host. Chip-select signals in the memory interface identify which rank is accessed on any given operation. Each DIMM is accessed over an independent memory interface from the host and is referred to as a channel. When a DRAM memory access is to be performed, the appropriate channel, rank, and bank are selected based on the physical address of the memory operation. However, before a read or write can take place, the relevant row of the selected bank must be activated. A row is activated by driving the appropriate word line and transferring the values stored in that DRAM row to the local and global row buffers. Once the data from the row is in the global row buffer (row open), a column from that is selected based on the address of the access. To access another row in the same bank, an open row first has to be closed by precharging the bitlines.

2.5.2 Near-Memory Computing (NMC)

Given the inherent hierarchy in the DRAM organization, NMC designers typically face a choice in where to place compute logic which is depicted in Figure 2.10 [6]. Most NMC designs implement compute units near DRAM banks ②. While it requires memory dies changes, incurs area overheads, and restricts data placement (all data needed for a computation should be in the same bank), this design point has considerable bandwidth advantage as compared to the host accelerator which can be as high as 14× [43]. Some NMC designs push computation logic within each sub-array in DRAM ③. It provides the highest memory bandwidth advantage by enabling access to all (or many) sub arrays in parallel. However, such a design incurs high complexity, area, and power costs as well as limited data accessibility due to the large number of ALUs and each ALU being associated with a relatively small amount of memory (data for compute must be located in the same sub-array). Alternatively, they can be implemented on the base die of the HBM stack as indicated by ① in Figure 2.10. These designs, however offer no memory bandwidth advantage.

2.6 Summary

In this chapter, we presented the background essential to understanding this dissertation. We described the basic concepts in deep learning and specifically, sequence-based models. We also include an overview of the distributed setups they use. We detailed the key primitives in these models that the dissertation optimizes for. Finally, we provided an overview of DRAM organization and ways to augment it for near-memory computing.

3 PROFILING SEQUENCE-BASED NETWORKS WITH SEQPOINT

Profiling and characterization of application behavior forms the groundwork that guides optimizations at various levels of the hardware-software stack from architecture to system design to compilers and libraries. Given the importance of understanding program behavior, there exist a plethora of tools and techniques that work at various levels of the system stack and provide the necessary insights which help researchers and developers design the next-generation of architectural, system level, and software optimizations. However, for complex workloads such as DNNs, application characterization is difficult due to large datasets and long runtimes of several hours to days on hardware. Furthermore, given the complex software stack such networks are based on, it is often challenging to reproduce their execution environment and run realistic workloads with real-world datasets on architectural simulators.

Prior work address this challenge by harnessing the iterative nature of DNN training to profile a few iterations after warm-up phase [295] While this works well for DNNs such as CNNs where the amount and nature of computations are independent of the inputs, it is inadequate for the increasingly important class of sequence-based neural networks (which we refer to as SQNNs), such as RNNs and Transformers. The amount and nature of computations in SQNNs vary with the inputs, resulting in heterogeneous iterations during training.

We tackle this challenge by exploiting the underlying features of the algorithms to identify a small subset of the training iterations that can accurately summarize the overall DNN training run. To this end, we characterize the factors that affect the execution profile of training iterations for two popular RNN-based SQNNs from the MLPerf [168] suite: DeepSpeech2 (DS2) [29] and Google's Neural Machine Translation (GNMT) [280]. Our

characterization shows that the *input sequence-length* of an iteration (Section 2.2.2.3) is the key factor that leads to variations in an iteration's execution profile. As such, exercising a small, curated set of sequence lengths in the training dataset can enable us create a representative execution profile of a long training run.

In order to select such a set of sequence lengths, we exploit the insight that inputs of similar sequence lengths have similar execution profiles. In tandem with this observation, we extend ideas from the well-known SimPoint [254] approach to cluster sequence lengths together. Then, we pick a representative sequence length from each cluster, which we call ***SeqPoint***. Similar to SimPoints, we assign weights to SeqPoints and use the weighted sum/average of SeqPoints to project the behavior of the overall training run.

We compare SeqPoint-based projections to measurements of full training runs and show that it can accurately summarize the entire training while significantly reducing the number of profiled iterations Moreover, SeqPoint can be used as a stepping stone to simulate complex SQNNs on architecture simulators. This chapter is based on the paper, *SeqPoint: Identifying Representative Iterations of Sequence-based Neural Networks*, published in ISPASS 2020 [217].

The relevant background for this chapter is provided in Chapter 2. The rest of this chapter is organized as follows. In Section 3.1, we describe challenges with profiling SQNN training. Next, in Section 3.2 we provide a summary of our study to find factors that affect the execution profile of DS2 and GNMT. In Section 3.3, we describe our proposal SeqPoint. We evaluate the efficacy of SeqPoints in Section 3.4 and characterize GNMT in Section 3.5. In Sections 3.6 and 3.7 we discuss SeqPoints's applicability/extensions and related work, respectively. Finally, we summarize and conclude in Section 3.8.

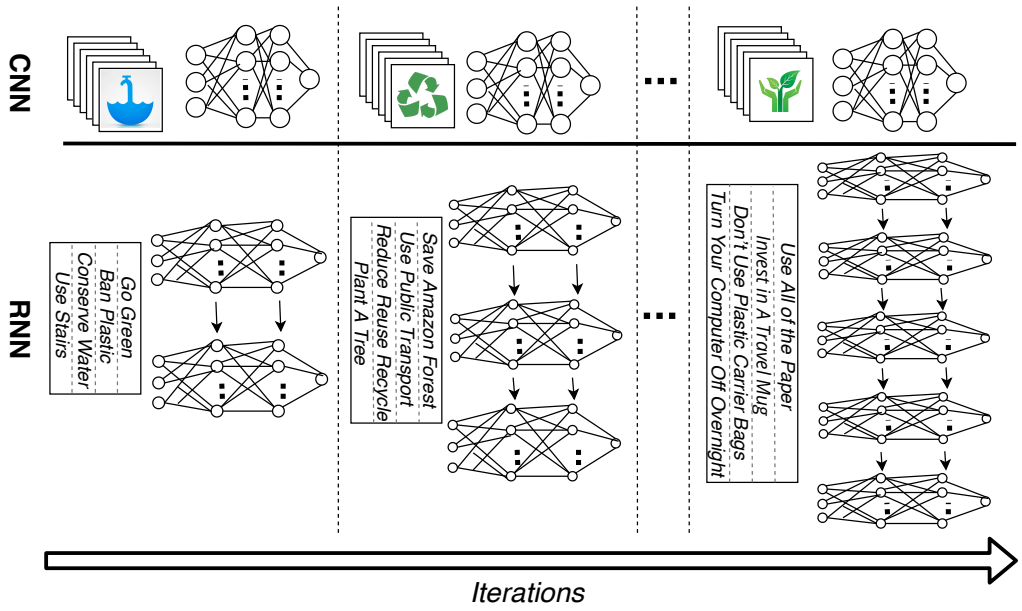


Figure 3.1: Comparing iterations of CNNs and SQNNs.

3.1 Challenges

To address the challenge of long execution times prior works identified representative portions of program execution and used their characteristics to guide whole program optimization [218, 254]. While selecting representative portions is difficult because the behavior of programs change over time, past work [295] has exploited the iterative nature of DNNs (Section 2.1.3) to pick a few iterations of the training phase as representative of the entire training run. Although this strategy is sound for CNNs, where the characteristics of the computation is largely the same across different inputs, it is sub-optimal for SQNNs where the amount and nature of computation can vary with each input. Figure 3.1 depicts this fundamental difference between CNNs and SQNNs such as RNNs. As discussed in Section 2.1.3, the training phase of DNNs is a collection of iterations, each with its input batch of data. While the input batch does not affect the computation performed for CNN training, the unroll factor

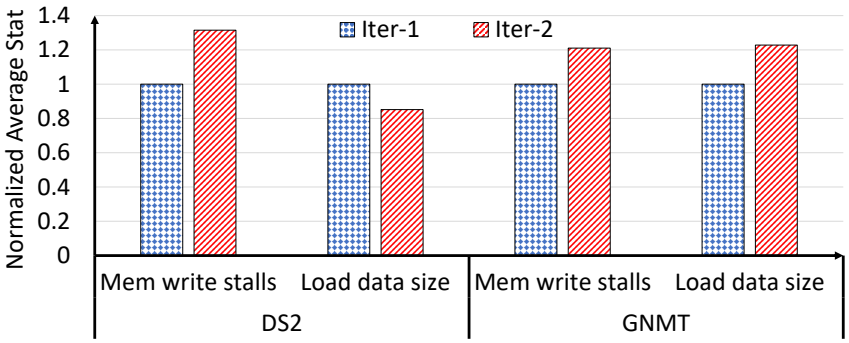


Figure 3.2: Architectural statistics for two training iterations.

of RNNs (Section 2.2.1) is dictated by the input batch. This leads to heterogeneous iterations for SQNNs with differing computations unlike the homogeneous iterations of CNNs training.

The heterogeneity of iterations in SQNNs is manifested in their architectural behavior as depicted in Figure 3.2. In the figure we compare a few hardware performance counter metrics (averaged across all operations) for two representative iterations of DS2 and GNMT (methodology discussed in Section 3.4). Specifically, we show performance counter data for the memory system behavior (read memory traffic, memory write stall behavior). These statistics differ by about 22% and 30% across iterations for DS2 and GNMT. Thus, generalizing the entire training run based on a few arbitrarily selected training iterations will likely be inaccurate since the selected iterations either may not represent iterations that have a major impact on the overall training run or will have different behavior from other iterations.

Due to this heterogeneity, a potential strategy to truly characterize the training phase of an SQNN is to profile a single training epoch instead of the entire training run. This suffices as different training epochs are largely homogeneous and encompass all possible iterations as discussed in Section 2.1.3. However, even a single training epoch for complex DNNs such as DS2 and GNMT can possibly run hours to days on real-world

datasets making profiling an entire epoch impractical.

Finally, unlike prior works [73, 291] which primarily focus on specific layers within SQNNs, we focus on characterizing the overall training phase of end-to-end networks. An end-to-end SQNN, such as DS2, often comprises several heterogeneous layers in a specific configuration (e.g., convolution, batch-normalization, and GRU). Thus, characterizing individual layers often misses out on interactions between such heterogeneous layers. In summary, existing mechanisms to profile and characterize SQNN training phase remain either inadequate or impractical. We aim to tackle this challenge by identifying representative iterations whose characteristics can accurately summarize the behavior of the entire training phase for SQNNs.

3.2 Characterizing iteration execution profile

The discussion in Section 3.1 illustrated that the heterogeneity of training iterations in SQNNs makes it difficult to select arbitrary training iterations and consider their behavior representative of the training run. Thus, we must carefully select iterations that are representative of the behavior of the entire training run. Accordingly, we analyze the applications to deduce key factors that decide an iteration's execution profile and use this to select representative iterations for the training phase.

3.2.1 Execution Profile

The execution profile of an iteration is directly related to the computations it executes. As training of SQNNs is typically executed on accelerators such as GPUs [295], in this work, we discuss the execution profiles in the context of GPU computations. Computation on a GPU is typically invoked as 'kernels' (analogous to functions in CPU parlance). As such, the execution

		M	K	N	
				sl-1	sl-2
GNMT	GEMM-a	36549	1024	6016	576
	GEMM-b	1024	36549	6016	576
DS2	GEMM-a	29	1600	25728	3776
	GEMM-b	1600	29	25728	3776

Table 3.1: Dimensions for the same GEMM operation across two iterations.

profile of an iteration comprises the distribution of the invoked kernels as well as their respective runtimes.

3.2.2 Factors Determining Execution Profile

3.2.2.1 Sequence Length

As discussed in Section 2.1, the computations in an SQNN iteration largely decide its execution profile (i.e., the kernels and their runtimes). These computations in turn are determined by network dimensions (e.g., number of layers, hidden state size) and inputs to an iteration. As such, an iteration's execution profile is largely dictated by the network dimensions and inputs to the iteration.

Throughout a training run, the network dimensions stay constant. However, inputs vary per iteration, and are dictated by batch size and, for SQNNs, the length of the input sequences. Although the *sequence length* (SL) may vary for each input of a batch, most SQNNs will pick a single SL for the entire batch (usually the longest SL in the batch) and pad the remaining inputs. Accordingly, while batch size is kept constant throughout the training run, the input SL varies per batch based on the specific inputs.

Input SL can affect the execution profile of an iteration in the following different ways:
First, some layers (attention, fully connected classifier) in a heterogeneous

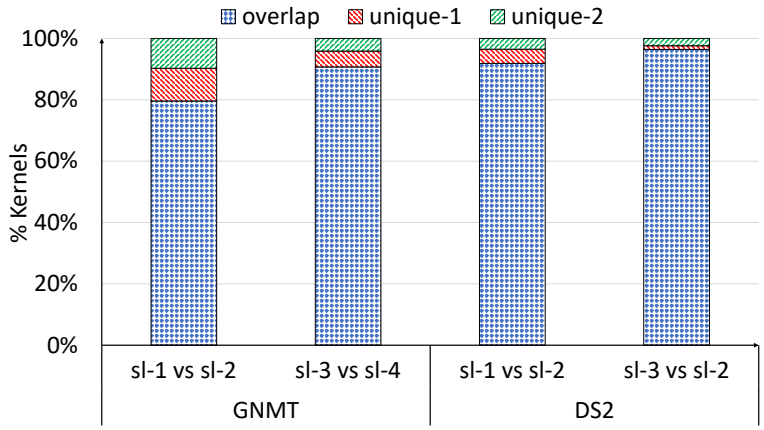


Figure 3.3: The number and types of kernels invoked differ based on sequence length.

SQNN process the entire input sequence causing the inputs to such layers (and their operations) to differ across iterations with different SLs. Table 3.1 depicts the input matrix sizes (M, N, K) for two such GEMM operations (GEMM-a, GEMM-b) across two iterations. The matrix dimensions differ and consequently, their runtime and contribution to the overall execution profile differ. The rest of the layers (GRUs, LSTMs) usually process one token of the sequence at a time, and hence have fixed-size inputs across iterations.

Second, due to the variation in input sizes of operations, different kernels (optimized for certain input sizes) may get invoked across different iterations. Figure 3.3 illustrates this with a pair of iterations from both GNMT and DS2. It shows the proportion of unique and common (overlap) kernels and while there can be several kernels common to both the iterations, there are up to 20% of kernels which are unique to either iteration.

Third, some layers in a heterogeneous SQNN are executed a fixed number of times per iteration (e.g., attention, convolution, fully connected), and some are executed as many times as there are tokens in an input sequence or SL times (e.g., due to unrolling of RNN layers described in Section 2.2.1).

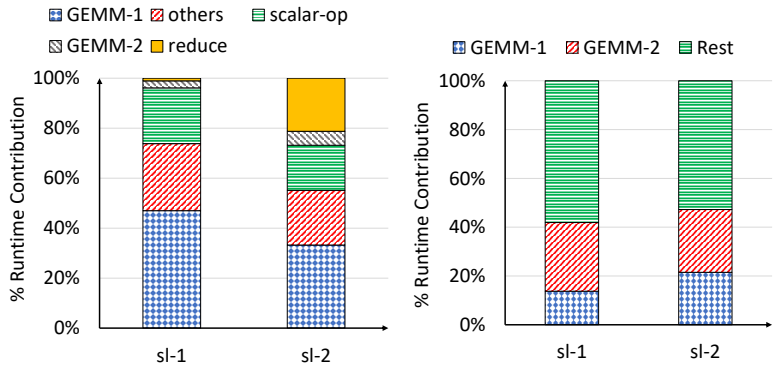


Figure 3.4: Kernel distribution differs based on sequence length [Left: GNMT, Right: DS2]

This implies that the proportion of these layers, and thus, of their respective kernels, change across iterations. This in combination with the points above, causes the kernel distribution of operations to differ across iterations as depicted in Figure 3.4. For example, the runtime contributions of kernels "GEMM-1" and "reduce" differ significantly based on the iteration's SL in GNMT.

Thus, we make the following key observations of how SL impacts the execution profile of SQNN training iterations:

Key observation 3.1: *Sequence length can differ across iterations and dictates the proportion of operations in an iteration.*

Key observation 3.2: *The total number and type of kernels invoked differ based on the iteration's sequence length.*

Key observation 3.3: *A given kernel can have different input dimensions across iterations and, thus, contribute to the overall execution profile differently.*

3.2.2.2 Training Dataset

Multiple datasets are often available to train a given DNN. As discussed in Section 2.1.3, the dataset dictates the number of iterations within a single training epoch and also the iteration inputs. As such, we observe that

representative training iterations are largely tied to the underlying training dataset. Furthermore, the training dataset stays constant across all epochs of a single training run. Epochs may only differ in the order in which the samples in the dataset are processed (and thus the order of heterogeneous iterations). Thus, considering iterations within one epoch is sufficient for identifying a representative set of iterations for the entire training.

Key observation 3.4: *Since the training dataset is constant during training, considering iterations within a single training epoch is sufficient to generate a representative training phase.*

3.2.2.3 Iteration Temporal Placement

As discussed above, the input SL is the key determinant of execution time of an iteration. Thus, in the absence of data-dependent optimizations (e.g., exploiting sparsity, which we do not consider in this work), the behavior of all iterations with a given SL will largely stay the same.

Key observation 3.5: *Unless data-dependent optimizations are used, considering iterations corresponding to unique sequence lengths suffices to generate a representative training phase.*

3.2.2.4 Vocabulary

The vocabulary of a dataset in sequence-based networks refers to the unique set of symbols (e.g., words) that appear in a given dataset. The vocabulary size remains fixed across iterations of a training phase and has a considerable effect on the execution time (lookup time when converting symbols to vectors, input dimensions to operations). Therefore, while sampling training iterations (which may refer to a subset of the dataset), it is important to keep the vocabulary size unchanged to preserve the representativeness of the iterations.

Key observation 3.6: *Since the dataset's vocabulary determines a considerable fraction of the per-iteration execution time, we must use the full vocabulary size*

of the original dataset.

3.2.3 Non-Training Phase Computations

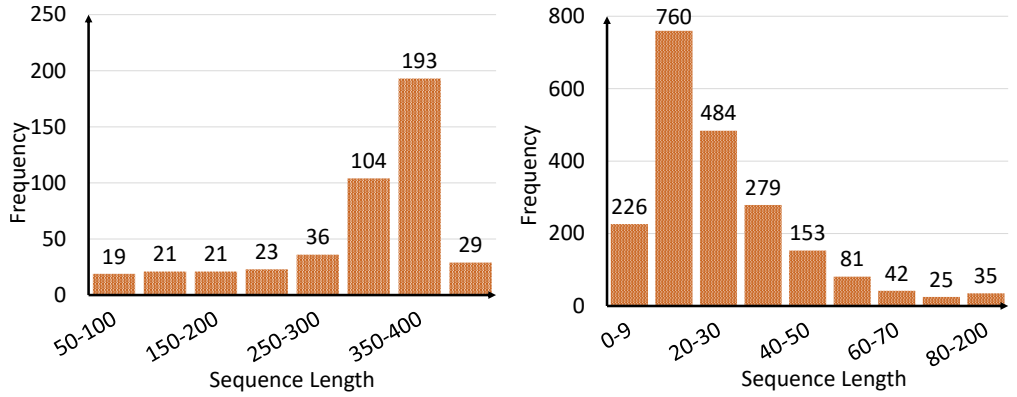
While DNN training largely comprises training iterations, there are also other computations.

3.2.3.1 Evaluation Phase

DNN training includes an evaluation phase at the end of every epoch to determine if a desired level of accuracy has been reached and training can be terminated. The evaluation phase has an independent dataset associated with it and is typically very small compared to the training phase. Unsurprisingly, empirically we observe that it only takes up to 2-3% of the total training time and thus can be ignored when creating a representative training run.

3.2.3.2 Autotune

Most high-level ML software frameworks employ an 'autotune' phase at the beginning of a training run to identify the optimal kernel to run for each computation in the network. It is usually an expensive process and affects the runtime of the first iteration (CNNs) or epoch (SQNNs). However, since autotune only runs once, we can easily ignore it when creating a representative training run.



(a) DS2 (b) GNMT

Figure 3.5: Histogram of SQNN sequence lengths.

3.3 SeqPoint: Representative Iterations for SQNNs

3.3.1 Challenge: Large Sequence Length Space

In Section 3.2 we analyzed SQNN training and identified several key factors that affect identifying representative iterations. In particular, SL is the key determining factor for variations in execution profile of training iterations. Thus, to select representative iterations of an SQNN training phase, in theory we could include all unique SLs in the training run. However, as Figure 3.5 shows, this is challenging because representative datasets for complex SQNNs like DS2 and GNMT have a large number of unique SLs. Consequently, including all unique SLs would lead to a representative set with up to half of all iterations in an epoch (e.g., DS2 with the LibriSpeech [207] 100 hours dataset). Moreover, the SLs in a given training run are also a function of the batch size. Since most SQNNs pick a single SL (often the maximum SL within the batch of inputs) for an iteration, smaller batch sizes have more unique SLs. Thus, simply selecting all unique SLs is not sufficient, and additional work is needed to identify

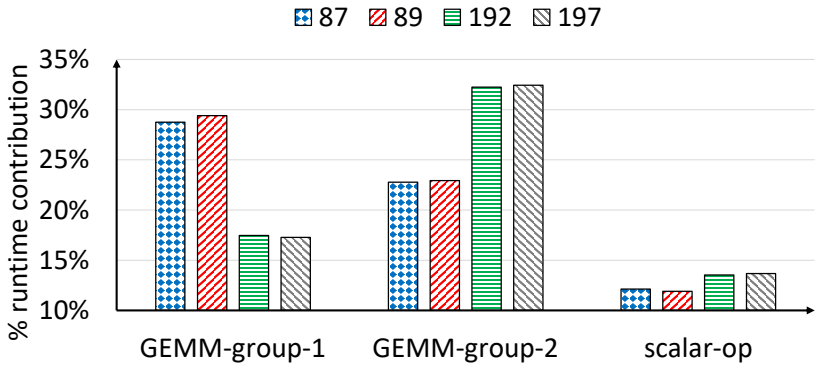


Figure 3.6: Execution profile with varying sequence length for GNMT.

a smaller set that retains the representativeness of using every unique SL.

3.3.2 SeqPoint Overview

Although SQNNs have a large number of unique SLs (Section 3.3.1), each with a unique execution profile (Section 3.2.2.1), similarly sized SLs have similarity in their execution profiles. Figure 3.6 shows that SLs that are close to each other (e.g., 87 and 89, or 192 and 197), have similar kernel distributions. Furthermore, Figure 3.7 shows that similarly sized SLs also have similar runtimes. We propose to exploit this similarity to create a smaller, yet still representative set of training iterations, taking inspiration from the well-known SimPoint methodology [254].

SimPoint divides program execution into slices and represents each slice with an architecture-independent metric: basic-block vector (BBV) which comprises basic blocks and their counts. It then uses clustering over the BBVs and selects a single representative of each cluster termed as SimPoint. In addition, it assigns weights to each SimPoint. Program behavior under SimPoint is then the weighted average of behaviors of individual SimPoints.

In a similar vein, we exploit similarity in SLs to bin them and select a single SL as the representative of each bin, which we term as SeqPoint.

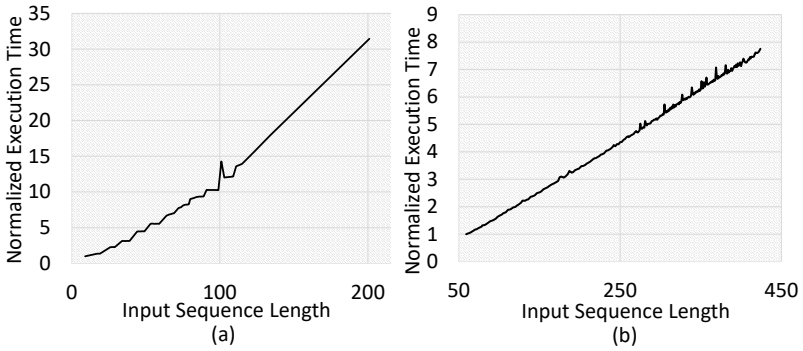


Figure 3.7: Runtime vs sequence length for (a) GNMT and (b) DS2.

In addition, similar to SimPoint, we also assign weights to each of the SeqPoints. The behavior of the entire training run is then a weighted average of all the SeqPoints. Overall, we use a SimPoint-like strategy to create a small, representative subset of the overall training run that is practical to profile and analyze.

3.3.3 SeqPoint Mechanism

Figure 3.8 depicts our SeqPoint mechanism. As illustrated in the flowchart, we first execute a single epoch of the SQNN training with the desired network, dataset, and batch size and log all the unique SLs exercised along with the runtime of the respective iterations (①). We also log the training time of the epoch. If desired, to control the training duration, the user can set a threshold, n, which decides the number of unique SLs to be included in the representative training run. If the total number of unique SLs is less than this threshold ($n = 10$ for our purposes), we include all unique SLs as SeqPoints.

However, if the number of unique sequence lengths is more than n, we bin the observed SLs into k buckets ($k = 5$, initially) each corresponding to a different SL range (②). Our binning of contiguous sequence length ranges is driven by the the fact that SLs in close proximity to one another behave similarly (Section 3.3.2).

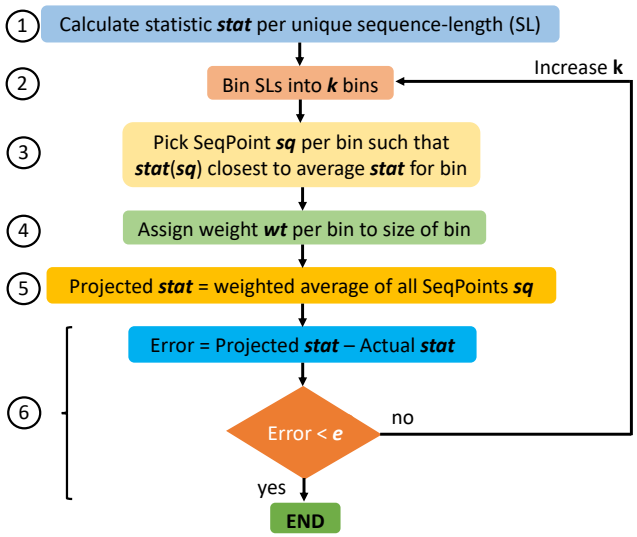


Figure 3.8: SeqPoint overview.

Next, we pick as the representative from each bin the SL whose runtime (s) is closest to the average runtime of the bin and consider it as a SeqPoint (③). This choice exploits the fact that iteration runtime is a good enough proxy of the program execution behavior, as shown in Figure 3.7.

In the absence of binning, we assign each SeqPoint a weight (w) equal to the frequency of its occurrence in one epoch of the training phase. In the presence of binning, the weight assigned is the size of the bin the SeqPoint belongs to (④).

Next, to evaluate the accuracy of the selected SeqPoints, we calculate the weighted sum of the runtimes of each SeqPoint as follows (⑤):

$$\text{Predicted Statistic} = w_1 * s_1 + w_2 * s_2 + .. + w_k * s_k \qquad (3.1)$$

If the error between the predicted and actual runtime exceeds an error threshold e (specified by the user, ⑥), we increment k by one and repeat steps ② to ⑥ until the threshold is met. Note that, to predict statistics that are ratios (e.g., throughput, IPC, etc), the value in Equation 3.1 should be normalized by the sum of all weights.

Overall, given the architectural independence of the SeqPoint method-

ology, once the SeqPoints for a given combination of model, dataset, and batch size are identified, they can be used to profile the SQNN on any system setup. Further, while we focus on runtime, the methodology can use any other statistic (or collection of statistics) that varies with SL.

3.4 Evaluation

3.4.1 Hardware & Profiling Setup

Our system consists of an AMD Ryzen™ Threadripper [13] CPU and a Radeon™ Vega Frontier Edition GPU [17][1]. The GPU has 64 compute units (CUs) and 16GB of HBM2 [110]. Our software stack comprises TensorFlow [19] built on top of the AMD ROCm platform[18], and calls into MIOpen [15] and rocBLAS [14], AMD's high-performance machine learning libraries. We use the Radeon Compute Profiler [16], a performance analysis tool, to gather kernel runtimes and other GPU performance counter data.

3.4.2 Networks and Inputs

We study two state-of-the-art SQNNs: GNMT, which is used for machine translation, and DS2, which is used for speech recognition. GNMT has three main components: (a) an encoder with seven uni-directional and one bi-directional Long Short Term Memory (LSTM) layers, (b) a decoder with eight unidirectional LSTM layers, (c) an attention network, which is a feedforward network connecting the encoder and decoder and (d) a fully-connected layer. DS2 has five bi-directional Gated Recurrent Unit (GRU), two convolutional, one fully-connected, and one batch-normalization lay-

[1]Given the fast-evolving GPU space with improved ML-specific optimizations built into each generation, we use a setup with the latest available GPU for ML at the time. Thus, the GPU used in this chapter differs from those in Chapters 4, 7, 5 & 8.

Config	GCLK	#CU	L1 $	L2 $
#1	1.6 GHz	64	16 KB	4 MB
#2	**852 MHz**	64	16 KB	4 MB
#3	1.6 GHz	**16**	16 KB	4 MB
#4	1.6 GHz	64	**0 KB**	4 MB
#5	1.6 GHz	64	16 KB	**0 MB**

Table 3.2: Configurations used to evaluate SeqPoint

ers. We use the IWSLT 2015 [47] and LibriSpeech [207] datasets with a batch size of 64 for GNMT and DS2 respectively.

3.4.3 Methodology

Hardware configurations: To show the efficacy of SeqPoint, we evaluate its ability to project both the overall program execution behavior and execution speedups under various hardware configurations for the two SQNNs detailed above. Table 3.2 lists the hardware configurations we study. We create five different configurations by varying GPU core frequency (GCLK) and number of active CUs, and by enabling or disabling its L1 and L2 caches. Further, we use total training time as a proxy for program execution behavior and study speedups in terms of increase in training throughput (samples/s).

SeqPoints: We generate the SeqPoints and their weights for GNMT and DS2 following the steps detailed in Section 3.3.3. Our methodology identified 15 SeqPoints for GNMT and 8 for DS2, respectively. Note that SeqPoints only need to be identified once, and we do so using config #1. Subsequently, only the SeqPoints are executed on the other configurations. Therefore, representative execution profiles of GNMT and DS2 training can be generated by executing only 15 and 8 iterations, respectively.

SeqPoint alternatives: We compare SeqPoint to other alternatives and prior approaches in our evaluation.

Frequent, Median, Worst: Prior work [293] used a single iteration as a proxy for the entire training run. By harnessing our insight that SL is a key

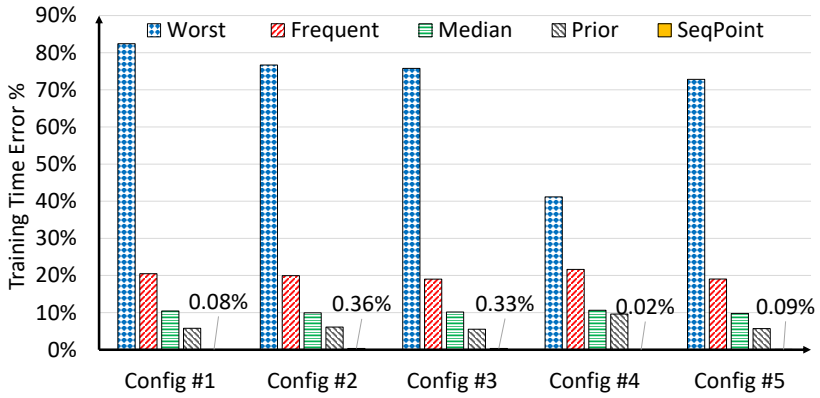


Figure 3.9: Error in total training time projections for DS2.

factor which causes heterogeneous iterations in SQNNs, we devise three strategies to select a single iteration as a representative. *Frequent* selects the most frequently occurring SL, as it has the most likelihood of being picked in a random selection. *Median* selects an iteration with the median SL. Finally, *worst* selects an iteration with the worst case error to provide a bound on possible error when arbitrarily selecting a single iteration.
Prior: *Prior* uses a sampling based approach [295] that samples 50 iterations after a fixed warmup period.

3.4.4 Projecting Program Execution Behavior

As discussed in Section 3.4.3, we use total training time as a proxy for program execution behavior. Figure 3.9 and Figure 3.10 show the error in projecting the total training time of DS2 and GNMT incurred by SeqPoint and its alternatives (calculated by multiplying average iteration time with the number of iteration in an epoch) for the configurations in Table 3.2.

As Figure 3.9 and Figure 3.10 depict, while we identified SeqPoints using only config #1, they can accurately project training time across a variety of system parameters resulting in geomean errors of 0.11% and 0.53% for DS2 and GNMT, respectively, across the configurations evaluated. This shows that our methodology allows for the SeqPoints to be identified once

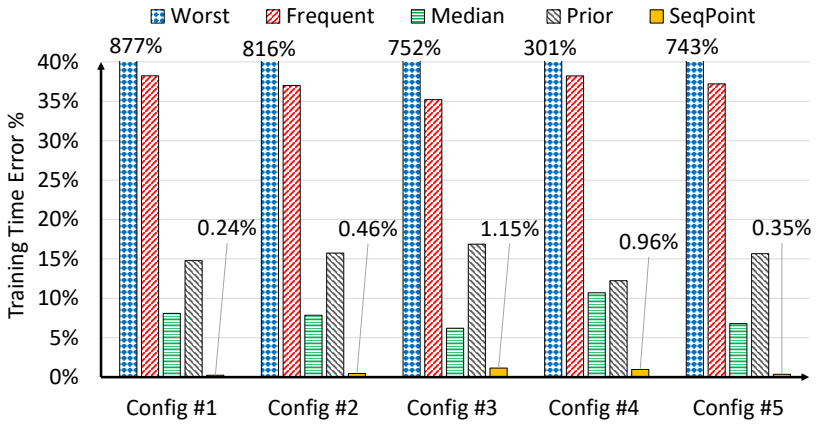


Figure 3.10: Error in total training time projections for GNMT.

and be used repeatedly to make accurate program behavior projections across a wide range of architecture and software stack variants.

Moreover, Figure 3.9 and Figure 3.10 show that SeqPoint alternatives which use a single training iteration to make projections have higher errors. For example, *frequent*, despite being the most frequent SL, has high error (20-35%) and thus is not very representative of the full training run. This is due to the fact that the most frequently occurring SL is not necessarily representative of the distribution of training iterations. Similarly, selecting *median* results in an error as high as 10%.

Despite the projection errors, both *frequent* and *median* were careful selections for a representative iteration based on our understanding of the underlying SL distribution. Selecting an arbitrary, fixed iteration is fraught with higher risk of projection errors as illustrated by *worst* in both the figures.

Figure 3.9 and Figure 3.10 show that *prior* results in lower errors (about 6%) for DS2 for certain configurations, but performs poorly both for other configurations and GNMT in general. *Prior*'s low error for certain configurations is a consequence of an artifact of DS2's computation: DS2 sorts SLs in the first training epoch, leading to *prior* selecting a set of iterations whose runtimes dominate the training run. Nevertheless, SeqPoint out-

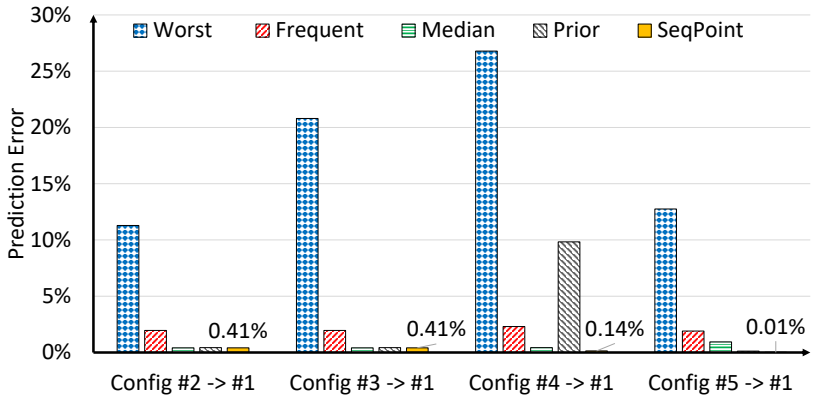


Figure 3.11: Error in performance speedup projections for DS2.

performs *prior* by over 5% and, more importantly, does so while running one-third and one-sixth of the iterations as compared to *prior* for GNMT and DS2, respectively.

3.4.5 Projecting Performance Speedups

We next evaluate SeqPoint's ability to project speedups as we vary hardware configurations. To do so, we plot the error (delta) in projecting percentage throughput (samples/s) change between config #1 and other configurations under study.

Figure 3.11 and Figure 3.12 show that SeqPoint outperforms all studied alternatives in projecting speedups with geomean errors of 0.13% and 1.50% for DS2 and GNMT, respectively. This further demonstrates SeqPoint methodology's ability to be representative of the entire training run.

Among the SeqPoint alternatives that select single iterations, we observe that while *median* and *frequent* perform worse than SeqPoint, their errors are sometimes within acceptable margins (e.g., 2.5% for DS2). This is due to the fact that both *median* and *frequent* select SLs which are exercised often. Combined with the SL distribution skew in DS2 (Figure 3.5), this enables them to accurately predict the relative variation across config-

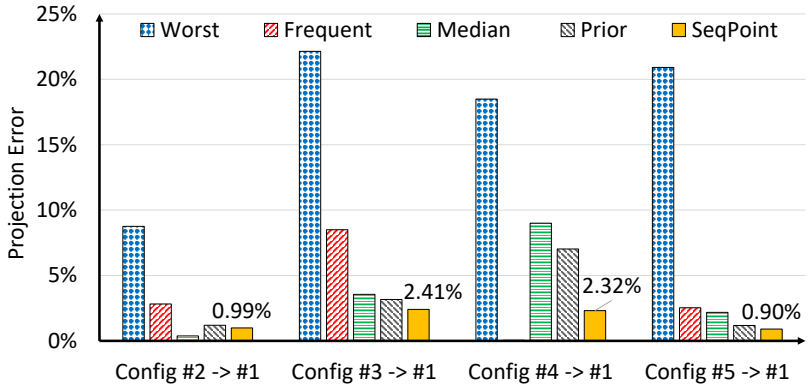


Figure 3.12: Error in performance speedup projections for GNMT.

urations reflected in the speedups. With a more uniform SL distribution, as is the case for GNMT, *median* and *frequent* exhibit higher errors of up to 9%. Further, as in Section 3.4.4, *worst* shows the perils of selecting an arbitrary training iteration: errors as high as 22% and 27% for GNMT and DS2 respectively.

We observe in Figure 3.11 and Figure 3.12 that *prior* does as well as SeqPoint in all cases except when predicting for config #4 for DS2. First, note that, while SeqPoint carefully picks points in SL space and weights them, *prior* simply picks a subset of the SL space. Depending on the overlap between these two sets it is possible, though not certain, for *prior* to select SLs which together are representative of overall training speedup for a specific hardware configuration.

For some configurations, however, *prior* can have higher errors as is the case for config #4 to #1 uplift. The region *prior* picks its iterations from is depicted by ① in Figure 3.13. The figure also shows that region ②, of which ① is a subset, has a constant (and given the skew in Figure 3.5(a), also close to overall) uplift for all configs *but* config #4, thus, leading to higher errors for *prior* in projecting config #4 to #1 uplift. This further underscores the need to carefully select iterations from the SL space as our proposed SeqPoint methodology does.

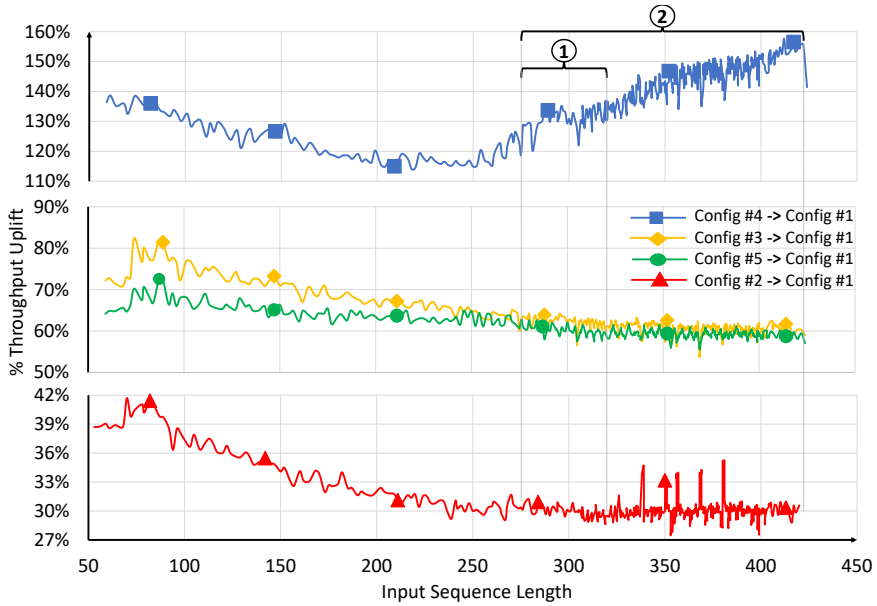


Figure 3.13: Sensitivity to GCLK, CU count, L1 cache and L2 cache of different sequence length iterations in DS2.

3.4.6 Profiling Speedups

A key benefit of SeqPoint is that it reduces the time required to profile an end-to-end SQNN model training from hours/days to mere minutes/seconds while being extremely accurate. By carefully selecting representative iterations, SeqPoint reduces profiling overheads by 40× and 72×, for GNMT and DS2 respectively. Moreover, given each SeqPoint is an independent iteration, they can be executed in parallel (on different machines) which further speeds up profiling by 214× and 345×, for GNMT and DS2 respectively.

Finally, while we have evaluated SeqPoint only for smaller datasets (LibriSpeech's 100 hours dataset [207] and IWSLT15 dataset [47] for DS2 and GNMT, respectively), applying SeqPoint to larger datasets such as the LibriSpeech 500 hours and WMT16 [40], which we observed to have similar SL ranges to the evaluated shorter datasets, can lead to much higher speedups than what we observe for these smaller ones.

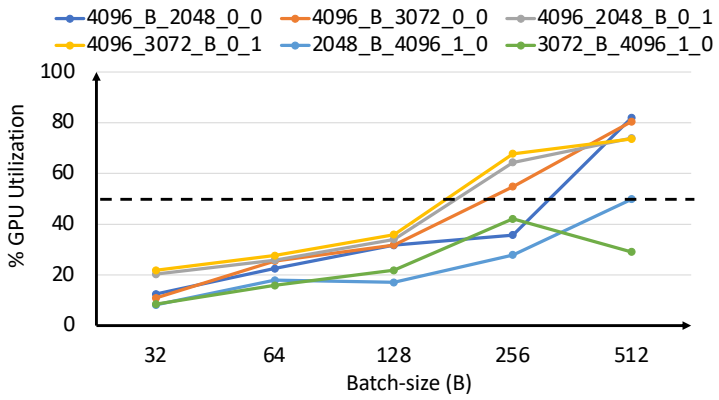


Figure 3.14: GEMM utilization in GNMT with increasing batch size.

3.5 GNMT Case Study

Characterizing GNMT (using its SeqPoints) reveals that more than 50% of its time is spent executing matrix multiplications (GEMMs). While the remaining time is spent executing element-wise operations, we believe, better software libraries [31] implementing kernel fusion for RNNs can fuse these with the remaining GEMMs. We further find that a majority of the GEMMs in RNNs have a low GPU utilization: 80% of the GEMMs have utilizations between 20-40% even with a batch-size of 64. This is because of sequential processing in RNN layers; layers process only a single token of an input sequence at a time causing GEMMs to have one of the dimensions as just the batch size. Increasing batch size only partially helps; as shown in Figure 3.14, half of the GEMMs still have utilization of $<$50% at a batch size of 256. Furthermore, it may not be possible to achieve such large batch sizes due to both hardware memory capacity limitations and algorithmic convergence.

3.6 Discussion

3.6.1 Enabling Network-level Simulation for SQNNs

Simulating an entire GPU application on a cycle-level simulator [83, 147] is often impractical, especially for long-running SQNN training applications. To aid in successful simulation of long-running applications, prior works have attempted to identify representative regions within applications and porting them to simulators for CPUs [45, 134, 218, 254, 281] and GPUs [98, 118].

Such techniques, however, require some form of program analysis, simulation, or profiling which can have significant overhead of up to 10× to 30×, making them infeasible for SQNN training (which run for days to months on native hardware). In contrast, SeqPoint reduces SQNN training profiling time to a few seconds to minutes by identifying few representative iterations. This, along with recent techniques to identify and simulate only few kernels (or portion of kernels) of a DNN model iteration [32] can enable representative network-level simulations of SQNN training.

3.6.2 Other SQNNs

While our analysis focuses on two SQNNs, SeqPoint applies to other networks as well. An insight of this work is to identify input SL as a key factor which determines the variations in execution profile (kernel distribution) for training iterations. As such, any SQNN consisting of layers whose computation varies with input SL can benefit from SeqPoint methodology to reduce the representative training runtime. A wide swath of networks fall into this category which employ layers including, but not limited to, attention (e.g., Transformer [273], BERT [62], and GNMT [280]), convolution (e.g., ConvS2S [76], DS2 [29]), and other recurrence-based ones (e.g., Seq2Seq [158] and ByteNet [117]).

3.6.3 Sophisticated Clustering of SQNN Iterations

We also considered a more sophisticated approach to tame the SQNN training SL space via k-means clustering [162]. In this approach, we applied k-means clustering to execution profiles of all iterations. However, we observed that our simple methodology to bin SLs (Section 3.3.3) performs as well as k-means clustering and, as such, we use the simpler approach. We believe this to be a consequence of the fact that iteration runtime (which we use) is a good proxy for execution profile of SQNN iterations.

3.6.4 Architecture and Software Independence

The SeqPoint methodology we propose relies entirely on the characteristics of the SQNN model and the dataset it is trained on. Therefore, while our system setup consists of AMD hardware/software stack and the TensorFlow framework, the insights we highlight and the methodology we adopt applies to any other system (e.g., NVIDIA) and/or framework (e.g., PyTorch, Tensorflow). Further, while we demonstrate the efficacy of SeqPoints in the context of GPUs, since SeqPoint uses architecture-independent metrics (e.g., SL), our methodology is also equally applicable to CPUs and other accelerators.

3.6.5 SQNN Inference

While the focus of this chapter has been on SQNN training, our insights can be useful in the context of SQNN inference as well. Our observation that SL is a key factor that dictates variations between SQNN iterations is equally applicable to inference. Further, our methodology to bin SLs to tame the SL space can also help characterize inference runs in order to optimize for them.

3.7 Related Work

As discussed in Section 3.1, prior works [4, 293, 295] assume homogeneity in training iterations of SQNNs, which we show is not the case. By being cognizant of heterogeneity in training iterations, our proposed methodology can generate a short set of representative training iterations (SeqPoints) that have lower error as compared to these prior techniques.

Other works, side-step end-to-end profiling of SQNN training and instead focus on analyzing individual layers [73, 291] using microbenchmarks such as DeepBench [175, 176]. However, real-world networks such as DS2 and GNMT comprise several different types of layers (e.g., convolution, attention), interactions among which remain uncaptured by these prior techniques. In contrast, by considering entire iterations, SeqPoint captures these interactions.

Finally, as discussed in Section 3.6.1, prior works [45, 98, 118, 134, 218, 254, 281] which identify representative portions in applications are unwieldy for long training runs. However, SeqPoint considerably reduces the training run and paves the way for such techniques to be used in architectural simulation of SQNN training.

3.8 Chapter Summary

Profiling and characterization of SQNN training runs remain challenging given their hours-to-days native runs. In this chapter, we showed that prior works that characterize SQNNs are oblivious to the heterogeneity in training iterations and, as such, are ill-equipped to create small, representative training runs that faithfully summarize entire training phases. To address this, we used the insight that input sequence length (SL) is a key factor that dictates the heterogeneity of SQNN training iterations. Then, we designed a new scheme, SeqPoint, that clusters unique SLs and selects a representative point from each cluster. We showed our identi-

fied SeqPoints are representative of the entire training run with low error and reduce the training iterations to be profiled by up to two orders of magnitude for state-of-the-art, end-to-end SQNNs. Using SeqPoints, we studied the dominant GEMM operations in GNMT training and found them to have very poor device utilization which we address in Chapter 7. Overall, we not only make profiling and characterization of SQNN training tractable but also pave the way for network-level simulations for SQNNs.

4 DEMYSTIFYING TRANSFORMERS

In the previous chapter, we devised a methodology to profile and characterize the extremely long-running training phase of RNN-based models, a subclass of sequence-based models (Section 2.1). In this chapter, we focus on Transformers, another important class of sequence-based models (Section 2.2.2).

Transformers [273] have become a popular general-purpose architecture for a wide range of tasks/domains. Recent work has shown that many different modalities are using Transformers as their base model (e.g., 41% of text, 22% of image) [38]. These networks, along with transfer learning, mark a shift towards deeper knowledge transfer by applying massive pre-trained models to different tasks. They use unsupervised learning to train on massively large unlabeled datasets (e.g., Wikipedia), which along with their network architecture, helped them outperform their predecessors on several Natural Language Processing (NLP) tasks. Google's Bi-directional Encoder Representation from Transformer (BERT) [62] is one of the first Transformers introduced for NLP. Several, larger, Transformer models have been introduced since BERT, with its most recent successor being MT-NLG (540 billion parameters), and many others in between [41, 58, 138, 154, 226, 256, 266, 284]. The evolution of these Transformer models however, has largely focused on changing Transformer block type (encoder vs. decoder), increasing the number of Transformer blocks, and/or increasing layer widths (Section 2.2.2). Given their increasing popularity and massive computational demands, understanding their underlying behaviors is vital to designing efficient accelerators for them.

Thus, in this chapter, we characterize the computationally and time-intensive training phase of Transformer models and identify how their algorithmic behavior can guide future accelerator design. We focus on BERT and identify key operations that are worthy of attention in accel-

erator design. While there have been some prior works that characterize Transformers, they miss important details such as the manifestation of layer operations and detailed runtime breakdown amongst all operations [274, 279, 289]. Consequently, some works build accelerators with matrix-vector engines for BERT layers which actually perform matrix-matrix operations [84, 89]. Thus, we focus our characterization on the manifestation, size, and arithmetic behavior of BERT's constituent operations which remain constant irrespective of hardware choice. Our results show that although computations that manifest as matrix multiplications dominate BERT's execution, they have considerable heterogeneity. Furthermore, we characterize memory-intensive computations which also feature prominently in BERT but have received less attention. Additionally, and to capture future Transformer trends, we vary BERT's hyperparameters and analyze the implications of these behaviors as networks and inputs get larger. Finally, we study the impact of key training techniques like distributed training, checkpointing, and mixed-precision training, which are employed to scale network training. More broadly, we identify inefficient execution phases in Transformer-based models that we address later in this dissertation (Chapters 6, 7 and 8). This chapter is based on the paper, *Demystifying BERT: System Design Implications*, published in IISWC 2022 [216].

The relevant background for this chapter is provided in Chapter 2. The rest of this chapter is organized as follows: Section 4.1 details our experimental setup. Section 4.2 provides a detailed runtime breakdown of BERT and analyzes its constituents. Section 4.3, 4.4, and 4.5 describe our observations from sweeping model hyperparameters, employing activation checkpointing, and multi-GPU training, respectively. Table 4.1 summarizes our main takeaways. Sections 4.6 and 4.7 discuss applicability, extensions of this work and other related work. Finally, we conclude with summary in Section 4.8.

Takeaway	Algorithmic Explanation	Sec.
LAMB optimizer is very memory intensive & important to optimize for.	LAMB updates 340M BERT parameters & is the second-highest training time contributor. It reads data worth 4× the model size and has few element-wise operations. LAMB's runtime scales linearly with transformer layer count & quadratically with layer size.	4.2.1 4.2.3 4.3
GEMMs dominate runtime but have heterogeneity.	BERT processes all input sequence tokens in parallel & thus layers manifest as GEMMs, even if mini-batch is one. Linear and Batched-GEMMs in attention are smaller than in FC & thus may not fully utilize accelerators & may also be memory-bound. GEMM proportion also increases with layer size.	4.2.1 4.2.2 4.3
Non-GEMMs are memory-bound & a considerable proportion of runtime.	These constitute element-wise (add, mul, scale) & reduction operations. Their proportion drops with increasing layer size as they scale only linearly with it (unlike GEMMs & LAMB, which are quadratic).	4.2.3 4.3
Reducing precision makes optimizing memory-intensive operations crucial.	GEMMs speedup more than others in half precision due to faster arithmetic & reduced memory traffic. Non-GEMMs only benefit from reduced footprint of reduced precision data. LAMB uses high (FP32) precision data to maintain accuracy and is unaffected.	4.2.1 4.2.3
Tensor Slicing is bottlenecked by communication.	Communication is serialized with computations in tensor slicing. Its cost increases with device count.	4.5.2

Table 4.1: Summary of takeaways

4.1 Experimental Setup

4.1.1 System

Our system consists of an AMD Ryzen™ Threadripper™ CPU [13] and an AMD Instinct™ MI100 GPU [21][1] with 32GB of HBM2 [110]. Our software

[1]Given the fast-evolving GPU space with improved ML-specific optimizations built into each GPU generation, our setup uses the latest available GPU for ML at the time. Thus, the GPU used for studies in this chapter differs from those in Chapters 3, 5 & 8.

stack is built on top of the AMD ROCm™ platform [18] with PyTorch v1.7. Although many accelerators are used to train BERT, we choose GPUs for this study because of their wide availability and popularity for DNN training. However, our takeaways are accelerator agnostic and should be applicable to other GPUs, accelerators, and frameworks suitable for machine learning. Since our goal is to characterize BERT training in a platform independent manner, we focus on relative importance of its operations, as well as the size and nature of operations, which in turn depend on BERT's network architecture, hyperparameters, and selected training mechanism (e.g., mixed precision; model versus data parallelization strategy). Thus, this approach provides fundamental value in guiding architecture design based on deep, algorithmic understanding of the application instead of solely profiling-based analysis, since architectures, both within and across vendors, evolve considerably from one generation to another. We discuss this further in Section 4.6.

4.1.2 BERT Phases

We analyze the BERT pre-training phase. Since fine-tuning requires only minor model tweaks and is similar to the more intensive pre-training, studying pre-training provides a solid understanding of BERT's overall training behavior while focusing on the costliest - most important to accelerate - part. We focus on Phase-1 (n=128) of pre-training with a mini-batch size (B) of 32 and discuss how Phase-2's (n=512) characteristics differ. Finally, we study both single and mixed precision (MP) training to discuss how bottlenecks shift with reduced precision. Tables 4.2a and 4.2b list the acronyms we use to refer to model details and training techniques.

Acronym	Parameter
B	mini-batch size
d_{model}	Hidden Dim.
h	#Attention Heads
d_{ff}	Intermediate Dim.
N	Layer Count
n	Sequence Length

(a) BERT hyperparameters

Acronym	Full Form
FC	Fully-connected
EW	Element-wise
LN	LayerNorm
DR	Dropout
RC	Residual Conn.
MP	Mixed Precision

(b) BERT acronyms

Table 4.2: BERT hyperparameters, GEMMs and acronyms.

4.1.3 BERT Hyperparameters

Although BERT has several configurations [272], we focus on the largest and most accurate one: *BERT Large*. BERT Large model contains 24 Transformer layers (N) with a hidden state size (d_{model}) of 1024, 16 attention heads (h) and an intermediate dimension (d_{ff}, usually $4 * d_{model}$) of 4096. Since these hyperparameters can scale in future models, we also study their impact on BERT's execution profile in Section 4.3. Throughout the remainder of the paper, we use these symbols to refer to the parameters, as shown in Table 4.2a.

4.1.4 Profiling Mechanism

Profiling entire BERT pre-training (with the English Wikipedia dataset [62]) can be impractical. Although CNNs can be characterized by profiling a single training iteration [160, 208, 293, 295], NLP model iterations can be heterogeneous due to varying input sequence length as we showed in Chapter 3. However, BERT's training iterations operate on same-size inputs within a phase. Thus, we profile and study a single training iteration (after a set of warm-up iterations) per pre-training phase. We use rocProf [12] to gather runtime and other performance counter data.

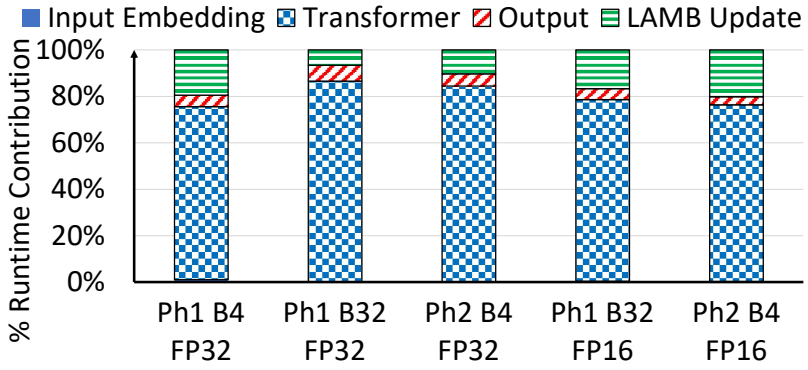


Figure 4.1: Runtime breakdown of BERT pre-training.

4.2 Compute & Memory Demands of BERT Operations

A BERT iteration performs (a) a forward (FWD) phase to process input sequences to produce an output, (b) a backpropagation (backprop, or BWD) phase to calculate the loss in output prediction and weight gradients, and (c) an update phase to update the weights using the gradients. Table 4.3 describes three GEMM operations and activation sizes for each important BERT sub-layer: one for FWD and one each for BWD activation and weight gradient calculation. Throughout we represent a matrix as MxN, a GEMM as MxNxK, and a product of two variables as $a * b$.

4.2.1 Runtime Breakdown

We first present a high-level runtime breakdown amongst different network layers and training phases. In all the runtime distribution plots we consider a layer's FWD and BWD phases together and show weight updates separately. Figure 4.1 shows this for different phases, B, and precisions: Phi-Bj-FPk, where i represents the phase (Phase-1 or Phase-2), j is the mini-batch size, and k is the number of floating-point (FP) bits used in the experiment. Note that, FP16 here refers to mixed precision training

[184] where FWD and BWD use FP16 inputs, weights, and gradients, but updates are in FP32 to maintain accuracy.

As expected, for all the configurations, the Transformer layers dominate (68-85%) the runtime while the output and input embedding layers constitute only a small proportion (3-7%). Interestingly, the LAMB optimizer (Section 2.4.2) is consistently the second highest contributor (7-25%). Its proportion is higher at smaller token counts ($n * B$) (e.g., Ph1-B4-FP32 and Ph2-B4-FP32 have a higher LAMB proportion than Ph1-B32-FP32). This occurs because the FWD and BWD runtimes depend on token count, while the weight update runtime is only proportional to model size. LAMB's proportion also increases with MP training (in Ph1-B32-FP16 and Ph2-B4-FP16). In MP, LAMB still updates a higher (FP32) precision copy of weights, and thus its runtime is not affected. Other operations speedup up due to faster arithmetic and reduced memory accesses at lower precision. Its proportion will further increase with more aggressive quantization [249].

Key observation 4.1: Transformer layers dominate (68-85%) BERT runtime. Output and embedding layers contribution is negligible.

Takeaway 4.1: LAMB updates are the second highest contributor (7-10%) to BERT's training time. Their contribution increases (25%) with decreasing token count per iteration.

Takeaway 4.2: LAMB updates become more important (16-19%) to optimize for with mixed-precision training.

Figure 4.2 presents a hierarchical breakdown of Transformer layers for single (Ph1-B32-FP32) and MP (Ph1-B32-FP16) training (labels represent their contribution to overall training time). The second bar, **Transformer**, shows the runtime breakdown among the Transformer layer's components: the attention layer, the Fully Connected (FC) feed-forward layer, as well as the combined dropout (DR), residual connection (RC), and Layer Normalization (LN) layers. Overall, FC layer has a higher runtime

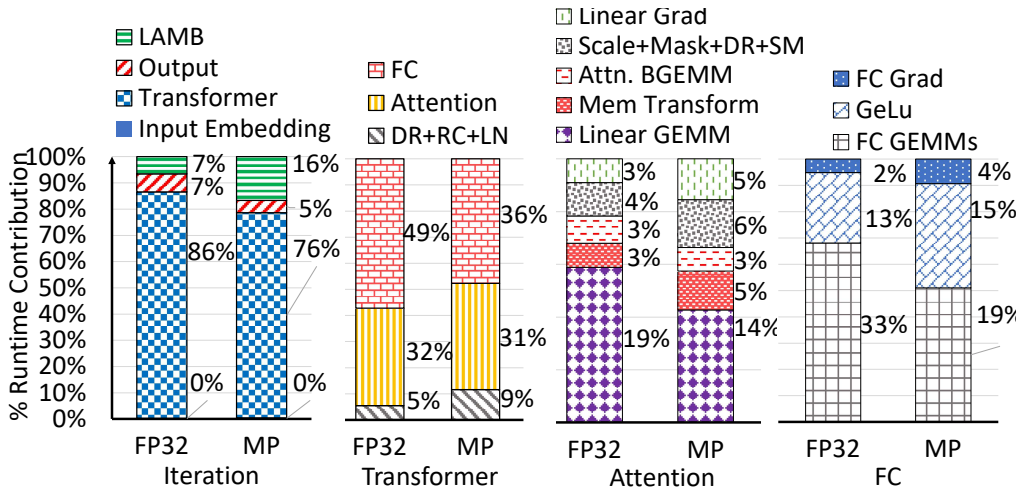


Figure 4.2: Hierarchical breakdown of BERT pre-training runtime. Labels show contribution to overall training time (SM=Softmax in this figure).

contribution compared to the attention layer due to its larger ($4\times$) intermediate dimension (Section 4.1.3). Additionally, the combined DR, RC and LN layers have a smaller, but non-negligible (5% in FP32, 9% in MP) contribution per iteration.

The third bar of Figure 4.2, breakdown of the **Attention** layer, shows that a significant portion of the runtime (22% in FP32, 19% in MP) is spent on *linear operations* (linear in Figure 2.3(c)). These operations are required to project each of the inputs query, key, and value vectors (of length d_{model}) into h different vectors (of dimension d_{model}/h) to be operated on by h attention heads (detailed in Section 4.2.2). The actual attention operation (Figure 2.3(d)) represented by Attn. BGEMM and Scale+Mask+DR+Softmax, constitute a much smaller proportion (7% in FP32, 9% in MP) of the overall runtime. The feed-forward sub-layer, **FC**, of BERT's Transformer layers (last bar in Figure 4.2) consist of two fully-connected connections with a Gaussian Error Linear Unit (GeLu) [92] activation in between. The FC connections (FC GEMMs+Grad) dominate the runtime with GeLu contributing to 13% in FP32 and 15% in MP.

Operation	FWD	BWD Grad. Activation	BWD Grad. Weight
Linear	d_{model} x n*B x d_{model}	d_{model} x n*B x d_{model}	d_{model} x d_{model} x n*B
Attn. Score	n x n x d_{model}/h, B=B*h	n x d_{model}/h x n, B=B*h	d_{model}/h x n x n, B=B*h
Attn. O/p	d_{model}/h x n x n, B=B*h	d_{model}/h x n x n, B=B*h	n x n x d_{model}/h, B=B*h
FC-1	d_{ff} x n*B x d_{model}	d_{model} x n*B x d_{ff}	d_{model} x d_{ff} x n*B
FC-2	d_{model} x n*B x d_{ff}	d_{ff} x n*B x d_{model}	d_{ff} x d_{model} x n*B

Table 4.3: Architecture-agnostic sizes of BERT GEMMs.

Finally, the proportion of linear and FC layers decreases considerably (from 57% in FP32 to 42% in MP) compared to other operations when executed with MP implying they benefit more from the drop in precision. These layers manifest as GEMMs (details in Section 4.2.2) and their speedup can be attributed to both faster arithmetic (Matrix Core Engine [20]) and smaller memory footprint.

Key observation 4.2: Linear and FC layers dominate (57%, FP32) BERT runtime. The rest of the time is spent executing several smaller operations.

Takeaway 4.3: Reducing precision speeds up GEMMs in the dominant linear and FC connections more than other operations, reducing their overall contribution (42% in MP).

Takeaway 4.4: Attention operations constitute a very small proportion (7% in FP32, 9% in MP) of BERT runtime.

4.2.2 GEMM Operations in BERT

Since GEMMs constitute a large proportion (55% in FP32 and 36% in MP) of BERT's iteration time, we next characterize these and their compute requirements. There are three sets of GEMMs in BERT's Transformer layers, corresponding to the attention computations, the linear transform operation, and the fully connected layers.

As illustrated in Figure 4.3 (within the dotted box), the attention head takes the query (q_n) and key (k_n) vectors of all the tokens in the input and calculates the attention score (a_n) between every token pair through the

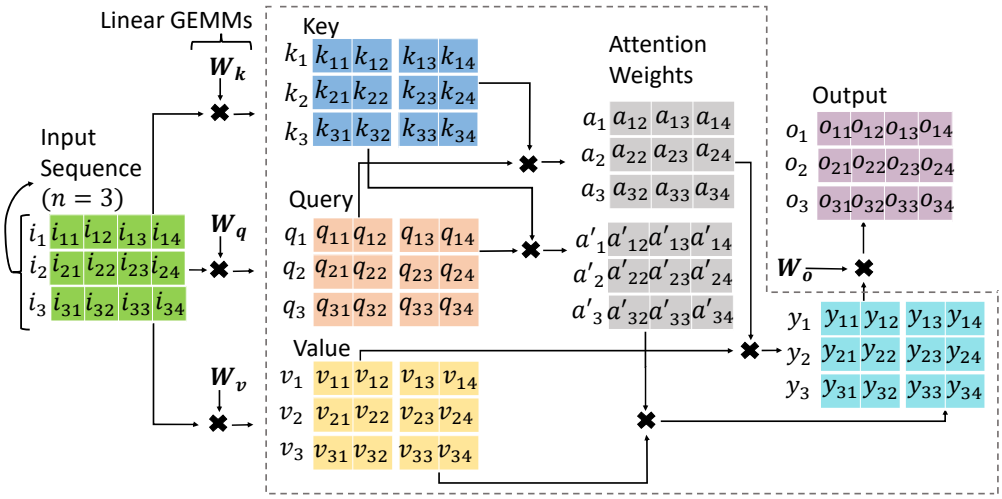


Figure 4.3: Computations in the Attention layer.

product of their respective query and key vectors using a GEMM. Since there are h independent attention heads working in parallel, there are h GEMMs executed in parallel per input sequence (h = 2 in Figure 4.3). Furthermore, since a training iteration operates on a mini-batch (B) of inputs, there are B ∗ h GEMMs invoked as a single batched-GEMM kernel (**Attn. B-GEMM** in Figure 4.2 and Attn. Score in Table 4.3). The attention scores are then used to calculate the weighted sum (y_n) of all value vectors (v_n) in the input sequence, also invoked as a batched-GEMM (Attn. O/p in Table 4.3) with B ∗ h parallel GEMMs. While there are several (B ∗ h) parallel GEMMs in this operation, each of them is quite small (dimensions of n and d_{model}/h).

To enable multiple attention heads, the query, key, and value vectors of the tokens are first linearly projected (outside the dotted box in Figure 4.3, left) into h smaller (d_{model}/h) feature vectors. All the token vectors of all the input sequences in a mini-batch are usually combined into a single (B ∗ n) x d_{model} matrix. Thus, unlike in RNNs, a batch size of one does not lead to matrix-vector operations in Transformers. Using the learned

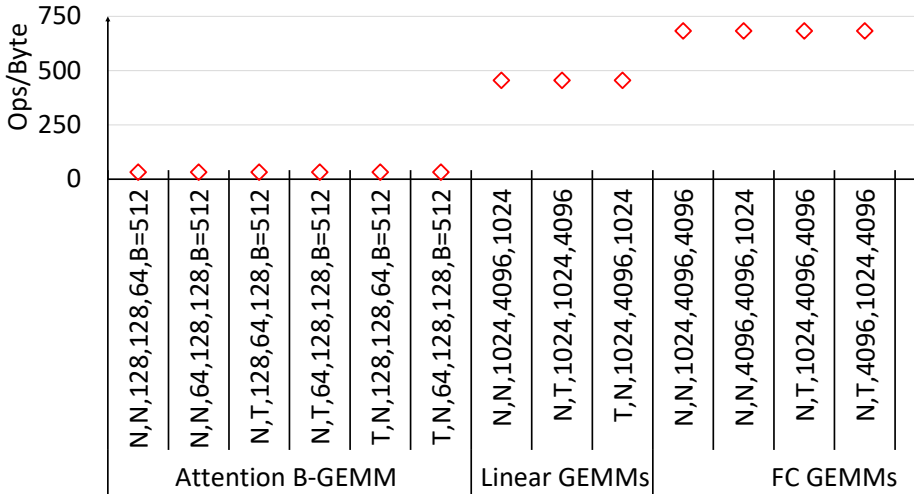


Figure 4.4: Arithmetic intensity of BERT's training GEMMs. It shows that not all of BERT's GEMMs are equal.

Weights (W_k, W_q, W_v), the tokens are linearly projected via three different GEMMs which dominate the attention layer runtime (**Linear GEMMs** in Figure 4.2 and Table 4.3). These GEMM outputs are then split to create the query, key, and value vectors for each of the attention heads. The concatenated outputs of the attention heads are also projected back using W_o (outside the dotted box in Figure 4.3, right). Finally, the FC layers use their learned Weights ($4\times$ the linear weights) to operate on the output of the attention layer. This creates two large **FC GEMMs** (Table 4.3 FC-1 & FC-2) which dominate the execution time of the FC layer (Figure 4.2).

Usually, larger, and squarer GEMMs perform better on modern accelerators by leveraging the highly parallel accelerator's compute power, exploiting data reuse, and better hiding memory latency. However, not all of BERT's GEMMs and B-GEMMs fit this paradigm. We analyze this by using the *arithmetic intensity* (ops/byte) of all GEMMs (labeled as $transposeA, transposeB, M, N, K, [batch]$) in a BERT's Transformer layer (Ph1-B-32-FP32) as shown in Figure 4.4. An algorithm's arithmetic intensity is the number of operations it performs for every byte of data read.

If it performs very few operations on each byte of data, it will likely be bottlenecked by memory bandwidth and vice-versa. It is an important parameter used to gauge if operations benefit from more compute, or higher memory bandwidth. Figure 4.4 shows that while the FC GEMMs are large and extremely compute-intensive, the linear transform GEMMs are not, with 4× smaller matrix dimensions and smaller ops/byte ratios. Furthermore, the attention layer's B-GEMM matrices are even smaller, leading to extremely low ops/byte ratio. We further plot their memory bandwidth requirements normalized to the maximum bandwidth achieved by any BERT operation (i.e., element-wise or EW multiply) in Figure 4.5. Attn. GEMMs have much higher (70%) memory bandwidth requirements compared to the other GEMMs (only 20%), making them memory-bound in contrast to the commonly occurring compute-bound GEMMs in DNNs.

Takeaway 4.5: GEMM dimensions in BERT are a multiple of the input token count (i.e., $B * n$), and layer's hidden size (d_{model} or d_{ff}) and scale with these parameters. Unlike RNNs, a B of one does not lead to matrix-vector operations.

Takeaway 4.6: Not all GEMMs in BERT are equal. Smaller, skinnier GEMMs in BERT's attention layer are memory-bound and can underutilize highly parallel accelerators.

4.2.3 Non-GEMM Operations in BERT

In Section 4.2.1, we observed that 45% (FP32) and 64% (MP) of BERT's training time is spent executing non-GEMM operations. Thus, accelerators for BERT-like models must optimize both GEMMs and these operations. There are four parts of BERT pre-training where we observe these operations: (1) LAMB, (2) scale, mask, dropout (DR) and softmax, (3) GeLU activation, and (4) DR, residual connection (RC), and layer normalization (LN). Input and output layer operations are omitted as they are a small proportion, especially as model sizes grow (Section 4.3.2). Figure 4.5

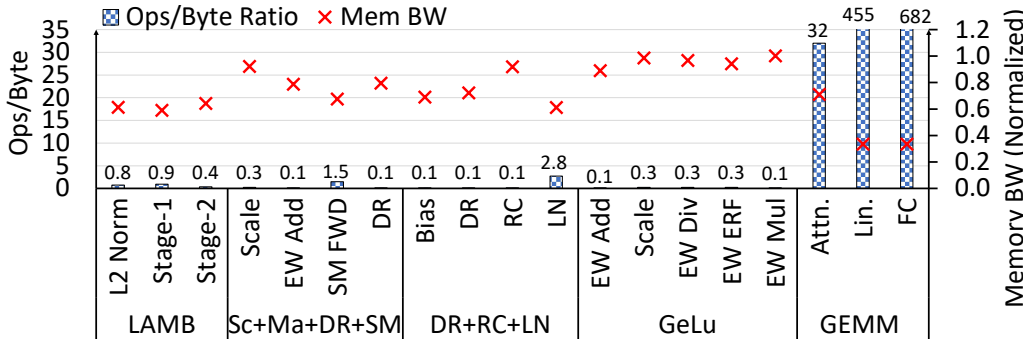


Figure 4.5: BERT op's arithmetic intensity & bandwidth requirements (SM=Softmax in this figure).

includes memory bandwidth requirements and ops/byte ratios of the four phases.

LAMB Updates: The LAMB algorithm described in background Section 2.4.2 is 7-25% of BERT's iteration time, and can further increase with increasing model size (Section 4.3.2), smaller token size per iteration, or MP training. It is implemented as two stages in [184]. LAMBStage1 (Figure 4.5) determines the update values and learning rate multiplier using additional momentum (m) and velocity (v) states from past iterations and gradients of the current iteration (all of which are the same size as the model parameters being updated, shown as MxN of the BWD Grad. Weight GEMMs in Table 4.3). This stage performs multiple EW add, multiply, divide, scale, and square-root operations on these parameters and therefore, has very low arithmetic intensity (Figure 4.5) making it memory intensive. The second stage (LAMBStage2 in Figure 4.5) updates model weights with stage 1's output also using multiple EW operations and has similar memory characteristics to stage 1. These two stages are executed for each layer, and access the corresponding layer's data (weights, gradients, and optimizer parameters). Therefore, each set has no data reuse across kernels (its impact on kernel fusion is discussed in Section 6.1). Moreover, LAMB must perform the L2 Norm (reduction) across all the

model's gradients before it can update any parameter, which serializes the model update with respect to the entire model backprop.

Takeaway 4.7: The memory-intensive LAMB optimizer reads 4× more data than the model size and has few EW operations.

Scale, Mask, Dropout & Softmax: The attention head generates attention scores between token pairs. These scores are normalized and operated on by a mask, softmax, and dropout functions (**Scale+Mask+DR+Softmax** in Figure 4.2) before being used to calculate the weighted representation of each token in the input. The normalization kernel multiplies each element of the input matrix with a constant value. The mask and DR operations, invoked as separate kernels, involve an EW add and multiply of the activation matrix with a mask and DR matrix, respectively. Therefore, all three perform only a single operation on each data read. Finally, softmax performs a series of EW operations on the input matrix, which improves its arithmetic intensity, but it is still not very compute intensive. Thus, these operations have high memory bandwidth requirements (Figure 4.5).

GeLU: GeLu activation [92] is executed between two FC GEMMs and consists of a series of EW add, multiply, divide, and ERF (error function) as shown in Equation 4.1:

$$\mathrm{GELU}(x) = x * \frac{1}{2} * [1 + erf(\frac{x}{\sqrt{2}})] \tag{4.1}$$

When invoked as separate kernels, these operations have very low ops/byte ratios, as shown in Figure 4.5. Along with the large input activation size (output of FC GEMM), this makes these kernels memory bandwidth bound.

Dropout, Residual Connection, & Layer Normalization: Outputs of the FC and attention sublayers are applied the DR, RC, and LN (DR+RC+LN in Figure 4.2) function as shown in Figure 2.3(b) (Add & Norm). DR randomly sets activation elements to zero using an EW multiply. RC does EW addition of the input to the output of a sublayer. Thus, these kernels

have an arithmetic intensity of less than one (Figure 4.5). Finally, LN [33] is a reduction operation and requires calculation of mean/variance of rows/columns, followed by a few EW operations. However, it still has a very low arithmetic intensity as shown in Figure 4.5. Consequently, these kernels are memory bandwidth bound.

While LAMB kernels remain unchanged in MP training (since updates are in FP32), most other memory bandwidth bound kernels speed up by $1.5 - 1.9\times$ in MP. However, this speedup is much smaller than GEMMs, thereby increasing the relative proportion of these operations in MP training. Thus, non-GEMM operations become even more important to optimize for when training with reduced precision.

Takeaway 4.8: BERT has multiple memory-bound elementwise operations that make up to 30% of its (FP32) runtime.

Takeaway 4.9: Optimizing memory-bound operations is even more important for BERT's reduced precision training, where they make up 46% of all operations.

4.3 Effects of Hyperparameter Sweep

Most Transformer-based NLP models have a similar structure to BERT and vary largely in their model/input sizes (discussed in Section 2.2.2). However, Transformer training characteristics can change as models get larger and deeper, with evolving hyperparameters like Transformer layer count, hidden dimension, mini-batch and sequence length. Thus, we next analyze and characterize the impact of these hyperparameters on BERT's execution profile.

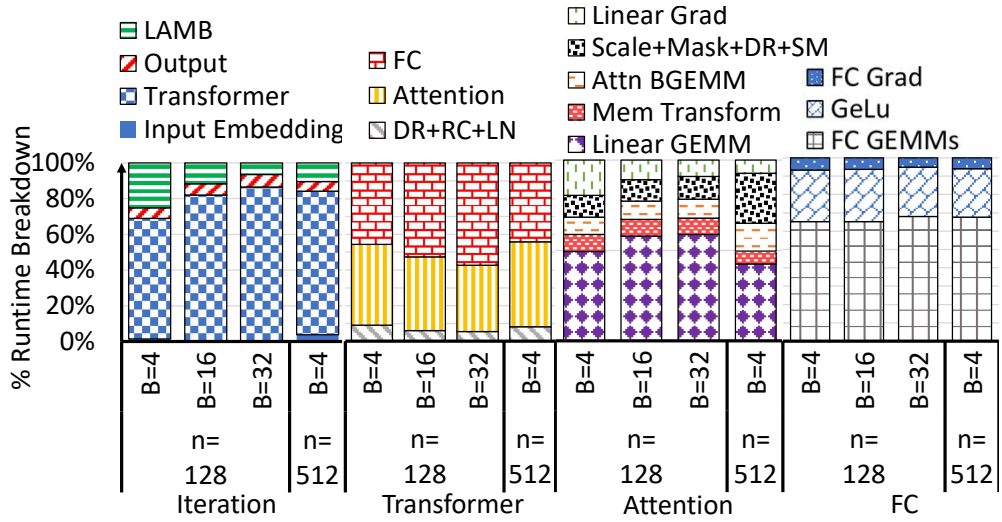


Figure 4.6: Impact of scaling mini-batch size & sequence length (SM=Softmax in this figure).

4.3.1 Input Size: Mini-batch Size (B), Sequence Length (n)

B and n impact training convergence and throughput. Increasing B improves throughput but may hurt convergence, especially in data-parallel training (Section 4.5). Conversely, increasing n improves accuracy but increases training costs. B and n decide the token count processed in a BERT iteration. Thus, increasing them increases the total computations in the forward and backward gradient calculations while keeping the parameter update computation (which only depends on model size) constant. Figure 4.6 highlights this: as B ranges from 4 to 32 LAMB updates constitute 25% to 7% of training time.

Within the Transformer layer the input size's impact varies across layers/operations. The impact depends on the layer type and its relationship with the input. For example, a layer with a MxNxK GEMM has operations proportional to $M * N * K$. Thus, increasing any dimension would linearly scale operation count. Since $B * n$ forms one of the Linear and FC GEMM

dimensions (Table 4.3), their operations scales linearly with B or n. This is similar to its impact on other operations (e.g., EW, reduce) which operate on activations with one of the dimensions as $B * n$. The number of GEMMs in attention B-GEMMs, and thus its runtime, also scale linearly with B (Table 4.3). Thus, the breakdowns of the Transformer layers with a constant n (128) but varying B (from 4 to 32) remains largely the same in Figure 4.6.

The efficiency of operations at each size also impacts the proportions: for higher Bs, attention and DR+RC+LN proportions drop, while FC's increases. Operations with small matrices may not be able to utilize the accelerator's peak throughput and/or memory bandwidth. A smaller $B*n$, along with $4\times$ smaller hidden dimension, can lead to smaller matrices in attention and DR+RC+LN layer as compared to the FC layer. Accordingly, increasing B (from 4 to 16 in Figure 4.6) improves the size of matrices, which improves these layers' throughput more than others, causing their overall runtime proportion to drop. The benefits, however, diminish with further increase in B.

Changing n has a similar impact as B, except attention operations (B-GEMMs in Table 4.3 and Scale+Mask+DR+SM) scale quadratically with n but only linearly with B. Thus, increasing n from 128 to 512 (and changing B from 16 to 4 to keep token count same) increases their proportion from 7% to 17% (B-GEMMs' proportion increases from 3% to 8%) as shown in Figure 4.6. This also implies that, unlike B, Transformer iteration time increases super-linearly with n.

Key observation 4.3: B impacts all layers similarly due to their linear dependence on it. Increasing it sometimes improves throughput.

Takeaway 4.10: Higher n makes attention operations important.

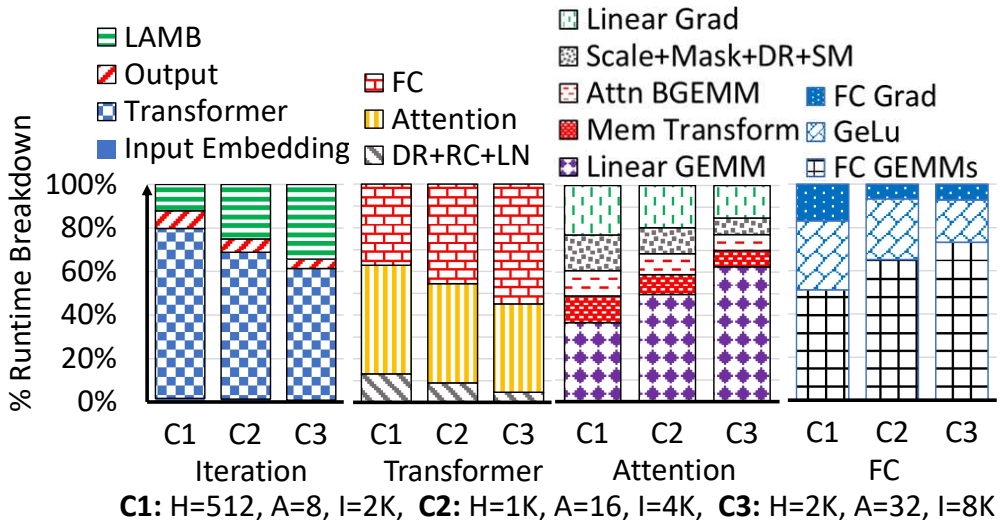


Figure 4.7: Impact of scaling Transformer layer size (SM=Softmax in this figure).

4.3.2 Model Size: Layer Count (N), Hidden Dimension (d_{model})

BERT model size is dictated by N and hidden sizes, d_{model} and d_{ff}. Increase in N linearly scales the count of every operation pertaining to a Transformer layer and LAMB update (parameter count also scales linearly). Intuitively, this does not change the Transformer layer breakdown, but runtime proportions of both the Transformer layers and LAMB update slightly increase as operation count in the input and output layers remain constant. Conversely, increasing layer widths (i.e., d_{model} and d_{ff}) increases the size of weight matrices and input to layer operations. Thus, it changes two of MxNxK GEMM's dimensions (see Table 4.3) and scales GEMM computation count quadratically. Since other layer operations only scale linearly with d_{model} or d_{ff}, the proportion of linear and FC GEMMs increase with increasing layer size. Figure 4.7 highlights this with three different model configurations (C1, C2, and C3, where C2 is the BERT-Large configuration): proportion of these GEMMs in configuration

C3 (i.e., similar to Megatron-LM-BERT with 2× higher d_{model} than C2) is much higher than in C2. Furthermore, the "Transformer" breakdown shows that FC runtime proportion increases compared to the attention layer. This indicates that (similar to changing B) throughput of linear GEMMs increase more than FC GEMMs', causing their runtimes to scale differently. Finally, the proportion of LAMB update increases considerably with larger layers (34% for C3). Unlike the linear scaling with N, parameter count and thus LAMB operations scale quadratically with d_{model} and/or d_{ff} (if $d_{model} = 1024$, layer parameters $= 1024 * 1024$). Thus, optimizing for complex optimizers like LAMB, which thus far had not been studied in detail, is increasingly important as Transformer models grow deeper and larger.

Key observation 4.4: GEMM and LAMB updates scale linearly and remain important as Transformer layer counts increase.

Takeaway 4.11: GEMM and LAMB runtime proportions increase with larger Transformer layers due to their quadratic relationship with layer size.

4.4 Effects of Activation Checkpointing

Activation (or gradient) checkpointing helps overcome device memory capacity issues. Instead of saving all layer activations from the forward pass to use in backprop, it checkpoints a limited set of activations and recomputes the others on demand during backprop. This reduces a model's memory capacity requirements and enables training a large model or a model with larger B on a single device. It, however, adds considerable recomputation overheads. We executed BERT Large training with activation checkpointing, which checkpoints activations at four ($\sqrt{N}$) different points and recomputes activations after backprop of every six Transformer layers. This increases kernel count by ~33% and runtime by ~27%. However, the breakdown within Transformer layers remains similar. Furthermore, since

LAMB remains unaffected, its proportion drops.

4.5 Effects of Multi-device Training

Although studying BERT training on a single device is important and reveals interesting computational behaviors, BERT is usually trained in a multi-device environment using data parallelism (DP), a form of model parallelism called tensor slicing (TS), or both (Section 2.3). Thus, in this section we first describe our profiling strategy for multi-device training. Next, we characterize training BERT Large on 128 GPUs using DP and TS (Megatron-LM [256] with 2-way and 8-way TS) approaches.

4.5.1 Modeling Multi-device Training

We construct per-device execution profiles in a distributed setting by building an analytical model from a single GPU's data. We use an analytical model because the publicly available BERT implementations are not optimized for multiple devices. For example, they do not overlap gradient computation and communication in DP training, without which network communication becomes a bottleneck. Thus, to avoid drawing incorrect conclusions we instead model the behavior analytically to take optimizations like these into account. This model also allows us to study different multi-device configurations and can project performance for hypothetical GPU/network improvements. We briefly describe how we model DP and TS training below:

Modeling Data Parallelism: Since DP training replicates the model on every device (described in Section 2.3.2), the per-device computation matches single-device training. Additionally, an all-reduce operation gathers each device's gradients (during backprop). To estimate all-reduce's communication costs, we use the gradient sizes and ring all-reduce algorithm [80]. To estimate communication time, we divide the gradient sizes by the communication bandwidth assuming PCIe™ 4.0. Finally, since

the communication and computations of different layer's gradients are independent, they can be overlapped (e.g., layer L's gradients are communicated while the device calculates gradients for layer L−1). We model this overlap by taking the maximum of the computation and communication times for every pair of consecutive layers.

Modeling Tensor Slicing: Megatron-LM splits most of the layer's operations across all devices as described in Section 2.3.3. Some involve splitting of weight matrices horizontally, while others are split vertically. The remaining layers (e.g., LN) are replicated across devices to reduce communication overheads. Figure 2.4(b) and (c) illustrate this change for the FC layer operations sliced across two devices. To model TS computations, we execute BERT operations with the expected input dimensions after splitting and replicating the layers. Since each device only updates a fraction of the weight matrices, the LAMB operations are also split equally amongst the GPUs. Finally, there are four all-reduce operations executed per forward and backward pass of a Transformer layer. We estimate this communication time using the approach described above. However, unlike in DP, these all-reduce operations cannot be overlapped with computations due to data dependencies.

4.5.2 Multi-GPU Training Profile

Figure 4.8 compares the execution breakdown within a single GPU participating in different distributed training mechanisms.

Data Parallel: The per-GPU execution profiles of BERT's DP approach with overlap, D2 (DP, B=16 w/ overlap) in Figure 4.8, is similar to a single GPU training, S1 (w/ B=16). This is unsurprising as each GPU has a copy of the model and independently computes the entire forward, backprop, and update phases. Although DP requires additional inter-GPU communication of local gradients, this cost (except for the first layer) can be hidden by overlapping computations and using a fast channel such

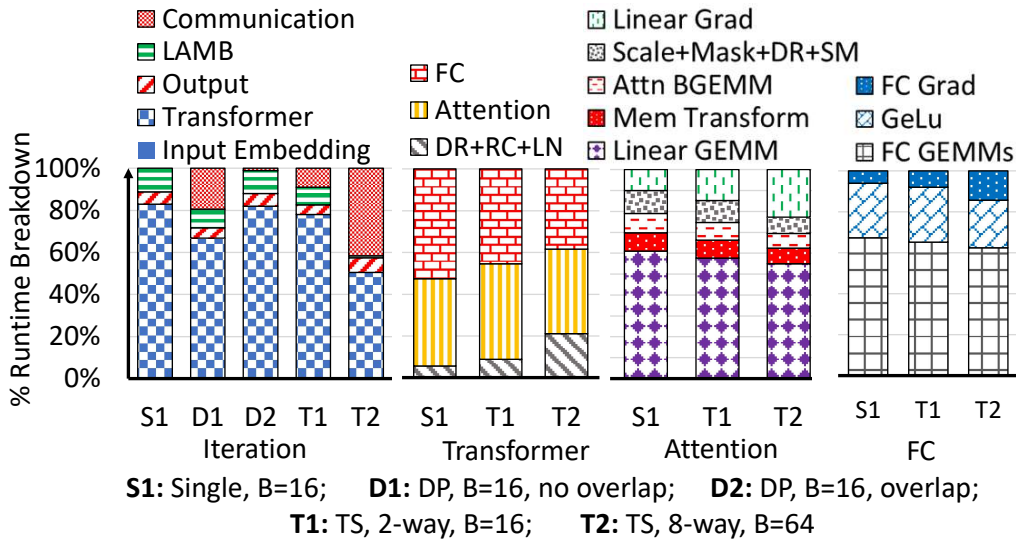


Figure 4.8: BERT iteration breakdown in a multi-GPU setup (SM=Softmax in this figure).

as PCIe 4.0™. D1 (DP, B=16, w/o overlap) uses the same data parallel approach as D2 but communicates gradients after the entire backprop, highlighting this cost. Consequently, a significant portion of D1's runtime (19%) is spent communicating gradients. Recent work has also shown that these communication overheads and redundant updates could potentially be reduced by making each device gather a reduced copy of a subset of gradients and only update the corresponding subset of parameters [229]. However, certain optimizers such as LAMB require normalization of all the layers' gradients at the beginning of the algorithm, thus requiring at least a single device to have a copy of all gradients.

Tensor Slicing: Figure 4.8 shows the per-GPU runtime breakdown for TS implementations: T1 (TS, 2-way, B=16) and T2 (TS, 8-way, B=64). The reduction in parameters going from 2-way to 8-way TS enables the increase in B. The high-level iteration breakdown of T1 is similar to S1, single-GPU training with the same B (16). However, there are two differences. First, T1 spends considerable time (9%) communicating activations and gradients.

Second, LAMB's proportion scales by half as each device is responsible for half of the model's parameters. These changes are more prominent in T2 which uses eight devices. The communication costs increase to about 42% due to the larger volume of data communicated (due to its larger B). Moreover, the proportion of LAMB is negligible in T2 with 8-way partitioning of parameters and is unaffected by an increase in B. As device count continues to increase, this trend continues since the total data traffic increases with device count while the per-device computations scale down proportional to device count and any scaling of B to improve per-device computations would also scale the communication volume. Finally, T2 also highlights that the proportion of replicated layers (DR+RC+LN), which are memory-intensive, increases with device count.

Key observation 4.5: The compute and memory-bound operation breakdown in a data-parallel, multi-GPU setting is similar to single-GPU training due to data parallel's ability to overlap most communication with computation.

Takeaway 4.12: Proportion of memory-bound LAMB updates drops for model-parallel, multi-GPU training as parameter count per device scales inversely with device count.

Takeaway 4.13: The communication volume (and runtime) increases with tensor-sliced devices due to a larger B.

To validate our analytical model, we compared our observations to prior work and found the takeaways to be similar [236, 256]. Megatron-LM observed near-linear scaling as they increase the number of devices in the DP training of BERT, implying little impact from synchronization and communication. Similarly, ASTRA-SIM show that using a DP approach for some ML algorithms can provide a near-perfect overlap of communication and computation, although this can change if compute speeds up much faster than communication. These observations are in line with our observation 4.5. Furthermore, Megatron-LM also shows that BERT

training's scaling efficiency drops with more TS devices due to increased communication overheads. This is also in line with our Takeaway 4.13.

Although we assume a homogeneous topology and network bandwidth, our takeaways also hold for non-homogeneous networks. Communication costs will not change for DP since communication and compute are overlapped. Although TS is more sensitive to communication, algorithms are often optimized for the underlying substrate (e.g., TS is usually employed within a node for higher bandwidth). Furthermore, while non–homogeneous networks within a node can change absolute communication cost (bottlenecked by the slowest connection), increasing cost with additional TS devices would still hold.

4.6 Discussion

4.6.1 Other Accelerators

While our analysis largely focused on a GPU, by focusing on platform-independent analysis, our takeaways can also guide BERT or other Transformer analysis on other devices or accelerators. Most of the observations (4.1, 4.4, 4.5) and takeaways (4.1, 4.2, 4.5-4.7, 4.11-4.13) are architecture-agnostic, only depending on BERT's architecture and the manifestation, size, computational complexity, and arithmetic intensity of its training operations. Thus, they hold regardless of the profiled accelerator. For example, analysis of BERT inference on CPUs shows that observation 4.1, which is purely based on model architecture, is also applicable to CPUs (differences between BERT training and inference are discussed in Section 4.6.2) [63]. Although some takeaways (e.g., 4.8) about operation runtime distribution might differ across accelerators, one can *approximately* extrapolate these proportions to another device by comparing the device's compute and memory bandwidth ratios. For example, the measured pro-

portion of memory-bound and GEMM operations on an AMD Instinct™ MI100 GPU are similar to other commercial GPUs with similar compute and bandwidth ratios [102]. While differences in GPU architectures can also impact this distribution, we believe they would be small enough to not alter the application's compute- or memory-boundedness. This demonstrates the value of architecture-independent takeaways and using any particular device only secondarily. Finally, since compute generally improves faster than memory, takeaways (4.7, 4.8, 4.9) involving the memory boundedness of BERT operations will either hold or be amplified in current and future accelerators.

4.6.2 BERT Fine-tuning & Inference

Although we focus on BERT's pre-training phase, our takeaways also hold for fine-tuning since the latter uses the same training techniques and model with changes only to the output layer (which is often simpler and thus negligible). For example, the output layer of SQUAD (Q&A) [231] is simpler than tasks BERT is pre-trained for, requiring fewer GEMMs and thus making it a negligible component of SQUAD fine-tuning. Importantly, just like pre-training, the Transformer layers still dominate the runtime. BERT's inference differs from pre-training since the former does not require backpropagation and parameter updates. Since backpropagation has approximately 2× more operations as a forward pass with similar properties, the breakdown of the Transformer layer's execution during inference would remain similar to pre-training. However, the high-level breakdown of an inference iteration would not include LAMB updates.

4.6.3 Other NLP Models

Although several Transformer-based models have been proposed after BERT, we focus on BERT as it embodies several of the essential trends that

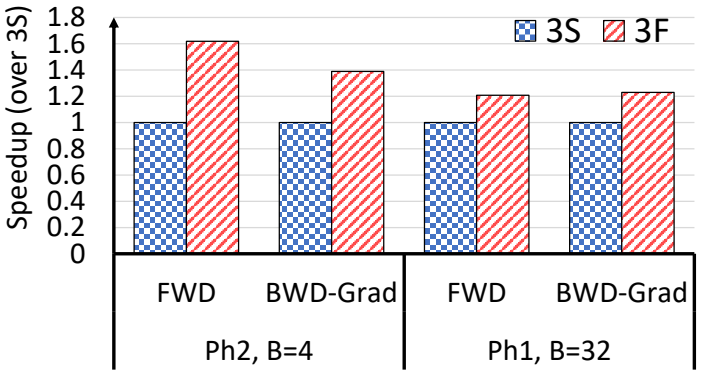


Figure 4.9: Impact of fusing 3 Linear GEMMs (3F vs. non-fused serial, 3S, execution).

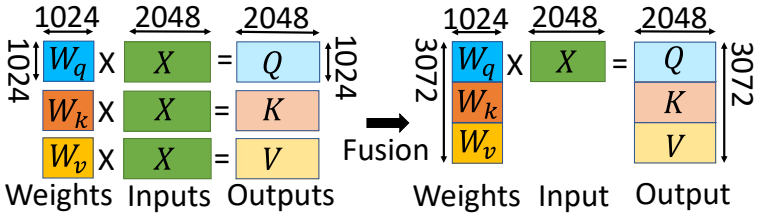


Figure 4.10: Fusion of Attention linear GEMMs in BERT.

are important when optimizing accelerators for these networks (discussed in Section 2.2.2). Furthermore, our analysis on the impact of larger and deeper models as well as of different input sizes (Section 4.3) capture future Transformer trends.

4.6.4 Optimizations for BERT

- **GEMM Fusion:** Fusing multiple smaller, independent GEMMs with a common input matrix into a single, large GEMM is another common optimization (Section 2.4). Figure 4.10 shows how the independent linear transform GEMMs of the attention layers can be fused. Since these GEMMs operate on the same input matrix and their respective weight matrices, W_q, W_k, and W_v (Figure 4.10, left), they

can be fused such that the weight matrices are concatenated, and the input matrix is read once (Figure 4.10, right). The output of this GEMM is simply the output of the three individual GEMMs concatenated, which can be split for subsequent use. Figure 4.9 examines the impact of this fusion on both FWD and BWD Grad. GEMMs of the linear layer. Fusion improves performance by up to 62% by enabling reuse of the common input matrix and increasing parallelism by using a larger matrix dimension. Its impact is higher when the input matrices are small (smaller token count or hidden dimension). Along with data layout improvements, such fusion can speed up BERT training considerably [102].

- **GEMM Accelerators:** A dynamically configurable accelerator would be well positioned to process the diverse sets of GEMMs. As shown in Section 4.2.2, GEMMs corresponding to BERT'S different sub-layers have different characteristics. While some GEMMs, such as those in the FC layer, are larger and compute-bound, others, such as those of the attention-heads, are small and memory-bound. Therefore, a GEMM's requirements can vary as its size, shape, or memory layout change, making a single implementation and/or policy for a GEMM accelerator sub-optimal.

- **In-Network Processing:** In-network processing involves acceleration of computations by adding compute capabilities to network switches. Such accelerators reduce the need for CPUs or GPUs to perform network-related operations, and eliminate the interference between computation and communication operations. In particular these accelerators have shown considerable gains for collective operations such as All-Reduce [130], used heavily in both model and data-parallel distributed training (discussed in Section 4.5).

4.7 Related Work

Characterizing DNNs: Prior work characterize ML workloads, especially CNNs, RNNs and recommendation models. Although most focus on inference [237, 274, 289], some also characterize training [82, 164, 295]. However, Transformer-based models, an important optimization target for modern systems, have received less attention; especially the expensive pre-training phase we focus on. Works that include Transformer characterization either do not provide detailed runtime breakdown amongst operations, only focus on its FC layers, or focus on inference rather than training [84, 274, 279, 289]. Instead, we focus on detailed end-to-end breakdown which helps in identifying LAMB or optimizer updates as one of the important candidates for Transformer acceleration. We also show how Transformer operations scale with varying hyperparameters and when employing different training techniques, which can be useful to project bottlenecks when training future Transformer models. While some works [63, 191] examine the impact of sweeping input size on throughput, they either do not include in-depth characterization that explains the behavior or are focused on inference in CPUs.

Optimizing Transformers: Recent work has also designed accelerators for Transformer-based networks. However, the relative lack of comprehensive characterization of Transformers has led these works to overlook important characteristics of self-attention. For example, recent works design both efficient matrix-vector [84, 89] and matrix-matrix engines [51] to accelerate BERT even though BERT does not execute matrix-vector operation the majority of the time, as our work shows. Unlike in RNNs where tokens are processed one at a time, Transformer layers process all the tokens of the input sequence in parallel. This leads to matrix, rather than vector, operations in Transformer layers even if mini-batch is one (e.g., during inference) as illustrated in Figure 4.3. Although some prior work acknowledges this property when comparing their accelerator against

GPUs [84], it did not influence the accelerator's design. This confusion about matrix-vector operations in BERT underscores the necessity of our work – understanding DNNs at an algorithmic level – before building efficient accelerators for them. Very few works optimize for non-GEMM operations or data-intensive phases [102], which we show have a significant runtime contribution that increases with reducing precision and increasing layer size. Amongst non-GEMMs, complex optimizers (e.g., LAMB), used in modern NLPs, have received little attention; we highlight LAMB's bandwidth-intensive characteristics and demonstrate how near-memory computing can help accelerate it. Finally, other works optimize Transformer inference [53, 63, 68, 276] or memory management [239].

Near-Memory Computing for Optimizers [125]: GradPIM [125] evaluated NMC for optimizers. However, they only evaluate simple momentum-based optimizers and focus on CNNs, which have an order of magnitude fewer parameters to update compared to NLP models.

4.8 Chapter Summary

BERT has been a groundbreaking innovation in NLP. Its accuracy stems from its Transformer architecture, millions of parameters, and its ability to train on enormous, unlabeled datasets. Its success has also inspired several popular models that are larger but have a similar structure to BERT. However, training these models is expensive due to their large compute and memory requirements. They pose challenges to system designers that must be met through deeper understanding of algorithmic behaviors as the waning of Moore's Law changes the virtuous synergy that has helped propel prior transformative improvements of ML and NLP. Thus, we focus on BERT's most expensive component, pre-training, analyze its execution, and provide a detailed characterization that acts as an exemplar for optimizing Transformer networks. Moreover, we further analyze how these characteristics change with evolving hyperparameters,

and training techniques, including mixed precision and in a distributed setting. Our analysis identified inefficiencies (memory-bound updates, small/underutilizing GEMMs) in their single-GPU execution which we address in Chapters 6 and 7. It also motivated our extended analysis of multi-GPU communication costs in Chapter 5, which we subsequently optimized for in Chapter 8. Overall, this characterization in this chapter identified several acceleration opportunities for Transformer networks.

5 TALE OF TWO CS: COMPUTATION VS. COMMUNICATION SCALING FOR FUTURE TRANSFORMERS ON FUTURE HARDWARE

Chapter 4 provided a detailed characterization of Transformers on a single GPU. It also briefly explored the impact of distributed, multi-GPU, training on BERT's execution and showed that, while communication from some setups can be overlapped and hidden, other setups have serialized communication costs. Thus, and given the rapid scaling in size of these Transformer models (e.g., 1000× increase in parameters) without the necessary growth in GPU memory (only 5×) [230], it is important to understand *how compute and communication in distributed training will scale relative to one another as both DNNs scale and hardware evolves*.

To guide the design of future systems to more efficiently train future large models, in this chapter, we perform this analysis, which we term **Comp-vs.-Comm**. We perform a comprehensive multi-axial (algorithmic, empirical, hardware evolution) Comp-vs.-Comm analysis for future Transformer models on future hardware. Figure 5.1 summarizes our approach. We first perform an algorithmic analysis. This analysis provides a system-agnostic view of Comp-vs.-Comm scaling, which is important given the wide variety of system/infrastructure capabilities, ranging from standalone accelerators to accelerator clusters with state-of-art interconnects. Our algorithmic analysis shows that the complexity of compute operations is often higher than communication volume (data size). We call this ***compute's edge*** over communication. A compute-dominated execution profile is often a positive edge because (a) traditionally, and especially for accelerators, compute (FLOPS) scaling has received considerably more attention than communication (bandwidth) scaling, and (b) often algorithmic/system optimizations are employed to overlap com-

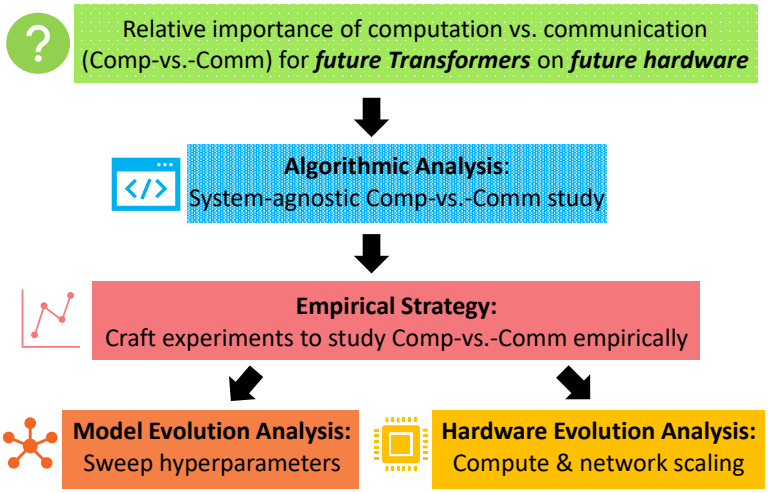


Figure 5.1: Overview of Comp-vs.-Comm analysis.

munication with useful compute. Thus, compute's edge also helps hide communication costs. Compute enjoys this edge for both serialized and overlapped compute and communication scenarios – both of which occur in distributed training. However, model scaling and memory capacity trends are stressing this edge.

To understand how compute's edge may be impacted by future models and future hardware, we empirically study Comp-vs.-Comm scaling. This approach has several challenges, including requiring studying many model/hardware evolution scenarios, each of which incurs significant profiling costs. Our empirical strategy addresses these challenges by (a) designing controlled experiments (guided by our algorithmic analysis), (b) executing only certain regions-of-interest (ROIs) and (c) using operator-level models which we show accurately capture runtime trends of operations for varying hyperparameters. These enable us to study hundreds of future models/hardware scenarios at 2100× lower profiling costs.

Our empirical strategy-driven experiments back up the conclusions from our algorithmic analysis. Specifically, we find that the compute's edge over communication is stressed: up to 50% of a future Transformer's

training time will be spent communicating data. Furthermore, communication that is overlapped or hidden today can exceed the compute time in future models, further increasing communication's proportion. Moreover, if past hardware evolution trends on the scaling of compute capability vis-a-vis communication bandwidth continue, communication will become an even bigger bottleneck ($>$ 75% of training execution) on future systems. Overall, our work highlights communication's increasingly large role as Transformer models scale and motivates our proposal, T3 (Chapter 8). This chapter is based on the paper, *Tale of Two Cs: Computation vs. Communication Scaling for Future Transformers on Future Hardware*, published in IISWC 2023 [213].

The relevant background for this chapter is provided in Chapter 2. The rest of this chapter is organized as follows. In Section 5.1, we motivate the study and introduce the terminology we use to describe different communication flavors. Next, in Section 5.2 we provide an algorithmic analysis of communication vs computations. In Section 5.3, we describe our strategy to empirically study communication vs computations and discuss the corresponding results. In Sections 5.5 and 5.6 we discuss the applicability/extensions of this work and related work, respectively. Finally, we summarize this chapter and our key takeaways from all the characterization work in this dissertation in Section 5.7

5.1 Motivation

5.1.1 Distributed Training Techniques and Associated Communication

All distributed training techniques have associated *communication* between devices as described in Section 2.3.1. These communication patterns are handled by *collectives* as described in Section 2.4.3. In this chapter, we

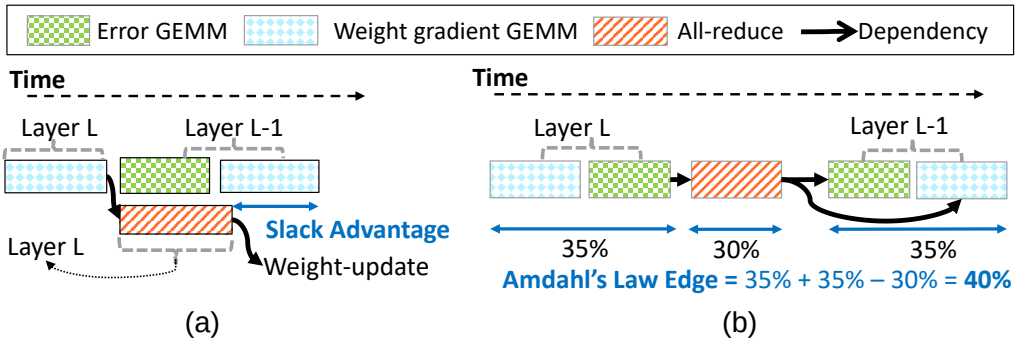


Figure 5.2: All-reduce in (a) data & (b) tensor parallelism.

focus on *all-reduce* (detailed in Section 2.4.3.1), the collective used in two of the most effective and widely adopted distributed techniques used by Transformer models: *data* and *tensor parallelism* (described in Sections 2.3.2 and 2.3.3). We discuss the communication impact of other techniques in Section 5.5, *Beyond DP & TP*.

5.1.1.1 All-reduce Communication Flavors

As described in Section 2.4.3.1, the all-reduce (AR) collective reduces copies of data generated by all participating devices such that each device contains a completely reduced version of the data. AR has different implementations optimized for different system topologies. While the AR collective remains the same, involving both communication and compute (e.g., element-wise summation), in both data parallelism (DP) and tensor parallelism (TP) setups its usage and thus its impact on the overall training behavior differs.

5.1.1.2 Data Parallelism (DP) & Slack Advantage

As described in Section 2.3.2, devices in DP all-reduce their weight gradients during the backward training pass to keep the model copies in sync. This all-reduce of gradients from one layer can happen asynchronously

with the gradient calculation of another layer (Figure 5.2(a)). Thus, the associated communication can potentially be overlapped and hidden by computations. However, complete overlap is only possible if the execution of computations equals or exceeds that of communication. We call this difference in overlapped compute and communication executions of each layer compute's *slack advantage* (Figure 5.2(a)).

5.1.1.3 Tensor/Horizontal Parallel (TP) & Amdahl's Law Edge

TP, as described in Section 2.3.3, splits model layers across devices and requires all-reduce to generate the final layer activations and errors (illustrated in Figure 5.3). This implies that a layer's forward and backward executions are dependent on another layer's all-reduce of activations and errors. Thus, unlike DP, in TP compute and communication are not asynchronous and communication is on the critical path of model execution (Figure 5.2(b)). We call the difference in compute and serialized communication executions compute's *Amdahl's Law edge* (Figure 5.2(b)).

5.1.2 Why Study Evolution of Compute vs. Communication Scaling

Although communication is necessary for distributed training, it may limit throughput scaling with increasing device count and cause compute resources to be idle when communication is on the critical path. Further, unlike a system's compute throughput, which accelerator designers have heavily focused on, network bandwidth has not scaled as much (e.g., 12× compute improvement versus 2× network bandwidth improvement [130]). If compute continues to scale more rapidly, when coupled with increasing communication volume, training future large-scale models on future systems will be inefficient. Thus, it is important to understand how Comp-vs.-Comm scale relative to one another as models

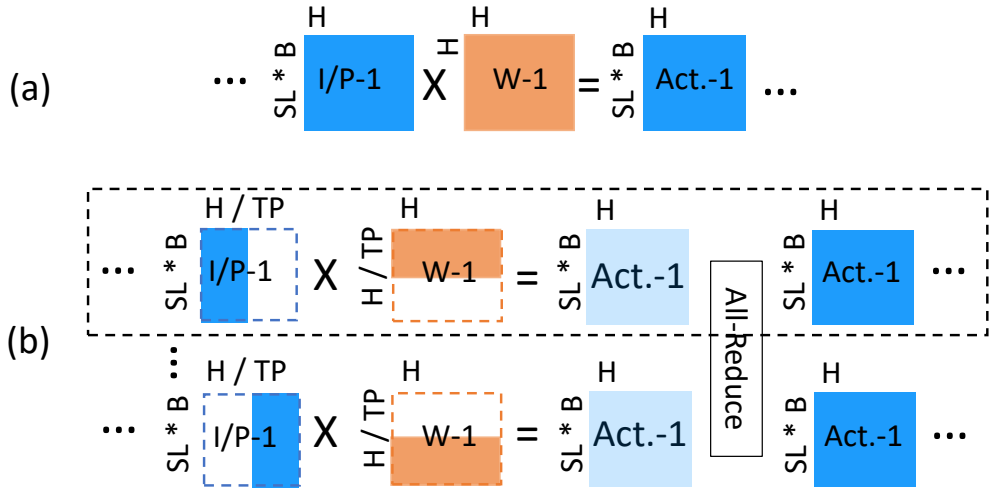


Figure 5.3: Layer operations (a) original, or w/ DP (b) w/ TP.

scale and hardware evolves. To address this we perform a multi-axial (algorithmic, empirical, hardware evolution) analysis of Comp-vs.-Comm scaling which will both inform and guide future system design to better support large-scale training of future models.

5.2 Comp-vs.-Comm: Algorithmic Analysis

We analyze compute and communication algorithmically as it provides a strong foundation to draw meaningful conclusions about future models. Moreover, as we demonstrate (Section 5.3), it also helps to create an empirical strategy to study model scaling on future hardware using existing hardware. Additionally, and equally importantly, it provides a system and infrastructure agnostic view of Comp-vs.-Comm scaling – ensuring that the takeaways are relevant regardless of studied system.

5.2.1 Distributed Techniques Studied

Although there are other distributed techniques and technique combinations, we study Transformers in the most commonly used data parallel (DP) and tensor parallel (TP) setups [55, 256]. DP and TP are imperative to divide and conquer large datasets and models, respectively. Furthermore, DP and TP are heavily supported in popular DNN frameworks such as TensorFlow and PyTorch.

5.2.2 Important Hyperparameters

The size, and thus cost, of model components is dictated by a model's hyperparameters [216]. As shown in Figure 5.3(a), the key hyperparameters that impact the size of weights, and activations in Transformers are: layer width or hidden dimension (H), input batch size (B), and input sequence length (SL). Although other hyperparameters are tuned during training (e.g., layer count, learning rate), they do not directly impact the size of operations.

Distributed setups can also impact the size of operations. In DP, since the model is replicated, operation size is unaffected. Conversely, as shown in Figure 5.3(b)'s dotted box, TP slices the operations. Hence, the number of devices a model is split across, the TP degree,[1] also impacts operation size. Thus, we use H, B, SL and TP to analyze Comp-vs.-Comm in an algorithmic and hardware-agnostic manner (Section 5.2.3).

5.2.3 Amdahl's Law Edge for Compute

As described in Sections 5.1.1.2 and 5.1.1.3, a distributed setup with DP and TP introduces communication in the form of all-reduce. Here we consider the TP-related communication that all-reduces partial activations. As shown in Figure 5.4(b), this communication, hereafter referred to as

[1] We use TP to refer to both tensor parallelism and its degree.

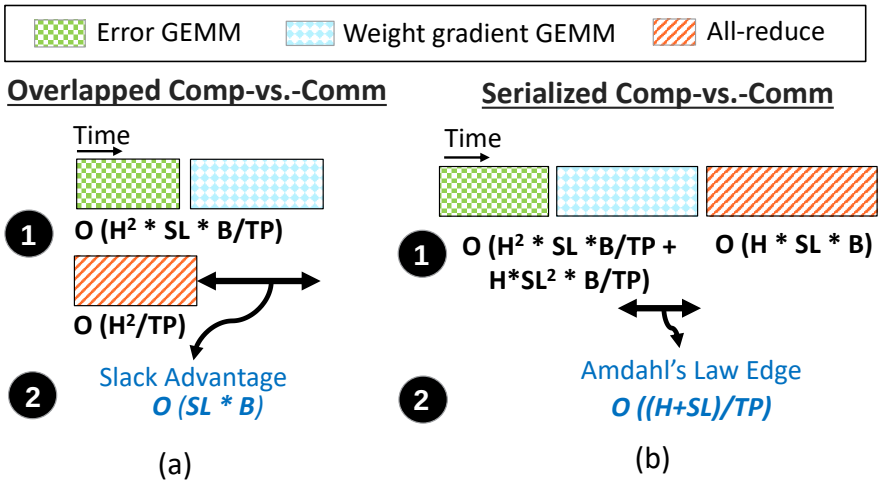


Figure 5.4: Comp's (a) slack over overlapped Comm. (b) edge over serialized Comm.

serialized communication, is on the critical path of model execution. To assess compute's relative edge or Amdahl's Law edge (Section 5.1.1.3), we find the relative contribution of all compute and serialized communication operations in a Transformer block. Since a model consists of multiple Transformer blocks of the same size, studying compute and communication within a single block is sufficient to characterize this for an entire model. For compute, this includes matrix multiplications (GEMMs) which represent Transformer's key layers (attention and FC, Section 2.2.2) and other element-wise and reduction operations that constitute the remaining activation functions and normalization layers. However, modern Transformer implementations usually fuse [66, 72, 264, 275] non-GEMM operations with the preceding GEMM to increase on-chip data reuse.

Thus, our algorithmic analysis only considers the dominant GEMMs for compute. Since GEMMs are usually compute-bound, we evaluate their algorithmic cost as the number of operations (multiplies and adds) they perform: $2 \cdot M \cdot N \cdot K$ (where M, N, and K are matrix dimensions and are derived from model hyperparameters, as shown in Figure 5.3). In a Transformer block, there are three key sets of GEMMs [216], with

computational complexities (with TP) shown in Equations 5.1- 5.4 (and in ① in Figure 5.4(b)).

$$\begin{aligned} \text{FC GEMM Ops.} &= 2 \cdot (4 \cdot H \cdot H/TP \cdot SL \cdot B) \\ &= \mathcal{O}(H^2 \cdot SL \cdot B/TP) \end{aligned} \tag{5.1}$$

$$\begin{aligned} \text{Attention GEMM Ops.} &= 2 \cdot (H/TP \cdot SL \cdot SL \cdot B) \\ &= \mathcal{O}(H \cdot SL^2 \cdot B/TP) \end{aligned} \tag{5.2}$$

$$\begin{aligned} \text{Linear GEMM Ops.} &= 3 \cdot 2 \cdot (H/TP \cdot H \cdot SL \cdot B) \\ &= \mathcal{O}(H^2 \cdot SL \cdot B/TP) \end{aligned} \tag{5.3}$$

$$\text{Total Comp. Ops.} = \mathcal{O}(H \cdot SL \cdot B/TP \cdot (H + SL)) \tag{5.4}$$

For serialized communication, we consider the total bytes of data that are all-reduced. In TP, layers' output activations and errors are all-reduced. Their sizes are a multiple (depending on precision) of the GEMMs' output matrices sizes (i.e., $M \cdot N$), and can also be represented in terms of the hyperparameters (see Figure 5.3). In a Transformer block, there are four serialized all-reduce operations, all with complexity shown in Equation 5.5 (and in ① in Figure 5.4(b)).

$$\begin{aligned} \text{Total Comm. Bytes} &= (\texttt{precision}/8) \cdot (H \cdot SL \cdot B) \\ &= \mathcal{O}(H \cdot SL \cdot B) \end{aligned} \tag{5.5}$$

Using Equations 5.4 and 5.5, we find the ratio between the number of compute operations and communicated bytes in Equation 5.6. This represents the complexity of **Amdahl's Law edge** that compute has over communication (② in Figure 5.4(b)).

$$\begin{aligned} \textit{Amdahl's law edge} &= \mathcal{O}((H^2 \cdot SL \cdot B/TP) + \\ &\quad (H \cdot SL^2 \cdot B/TP))/\mathcal{O}(H \cdot SL \cdot B) \\ &= \boldsymbol{\mathcal{O}((H + SL)/TP)} \end{aligned} \tag{5.6}$$

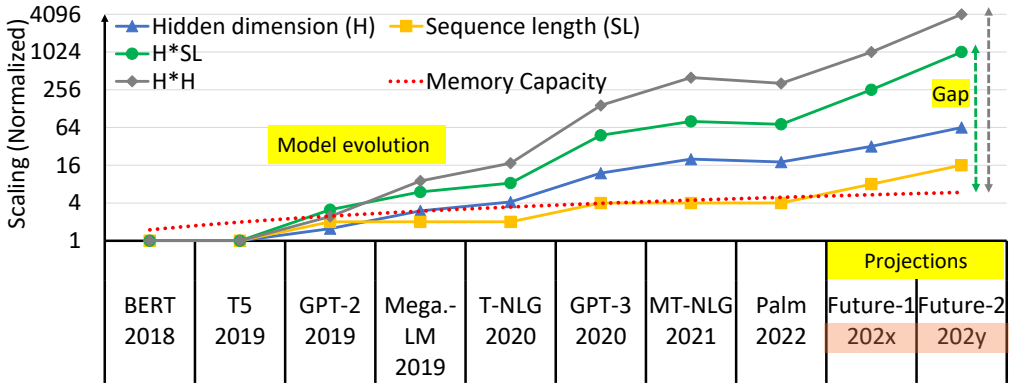


Figure 5.5: Model and device memory capacity trends.

This complexity has two implications. First, given the values of these hyperparameters in state-of-the-art Transformers (Section 5.2.5), with $(H+SL)$ being always greater than TP, compute (ops) enjoys an algorithmic edge over communication (bytes). Second, this edge can decrease if the required TP degree increases more than the increase in $(H + SL)$ between Transformer models, resulting in an overall increase in communication proportion.

5.2.4 Slack Advantage for Compute

Similar to our serialized communication analysis, we also study DP's *overlapped communication* that all-reduces partial gradients in backprop, as illustrated in Figure 5.4(a). We algorithmically analyze the relative cost of compute and overlapped communication and assess compute's ability to hide communication (Section 5.1.1.2). Unlike serialized communication, analyzing overlapped communication per Transformer layer is sufficient. Unlike TP, gradient communication occurs only in backpropagation, is done for every layer, and is usually overlapped with gradient/error calculating GEMMs. Thus, we can study the overlap efficacy by analyzing the compute and communication at every layer during backprop. For compute, we consider GEMMs which calculate backprop weight gradient

and error (or input gradient); for communication we consider the weight gradient size that is all-reduced. Absolute values of compute and communication can differ across layers: for example, with $4\times$ layer widths, the compute and communication costs in FC layers are $4\times$ those of attention. However, their complexities with respect to hyperparameters are the same. Thus, while Equations 5.7 and 5.8 derive these complexities for the FC layer (① in Figure 5.4(b)) they also hold and summarize algorithmic behavior for all Transformer layers.

$$\begin{aligned}\text{FC weight gradient + Error GEMM Ops.} &= 4 \cdot (4 \cdot H \cdot H/TP \cdot SL \cdot B) \\ &= \mathcal{O}(H^2 \cdot SL \cdot B/TP)\end{aligned} \tag{5.7}$$

$$\begin{aligned}\text{Comm. bytes} &= (\mathtt{precision}/8) \cdot (4 \cdot H \cdot H/TP) \\ &= \mathcal{O}(H^2/TP)\end{aligned} \tag{5.8}$$

$$\begin{aligned}\textit{Slack advantage} &= \mathcal{O}(H^2 \cdot SL \cdot B/TP)/\mathcal{O}(H^2/TP) \\ &= \mathbf{\mathcal{O}(SL \cdot B)}\end{aligned} \tag{5.9}$$

Equation 5.9 (② in Figure 5.4(b)) uses Equations 5.7 and 5.8 to find the ratio between the number of GEMM operations and total bytes communicated or all-reduced, and the complexity of compute's **slack** (i.e., ability to hide communication). This $SL \cdot B$ factor provides compute operations additional slack to hide the cost of bytes communicated. However, decreasing the input size ($SL \cdot B$) can decrease the slack and potentially expose communication costs.

5.2.5 Model Scaling Stresses Compute Edge/Slack

Our analysis in Sections 5.2.4 and 5.2.3 show that, algorithmically, compute has both an Amdahl's Law edge over serialized communication and slack

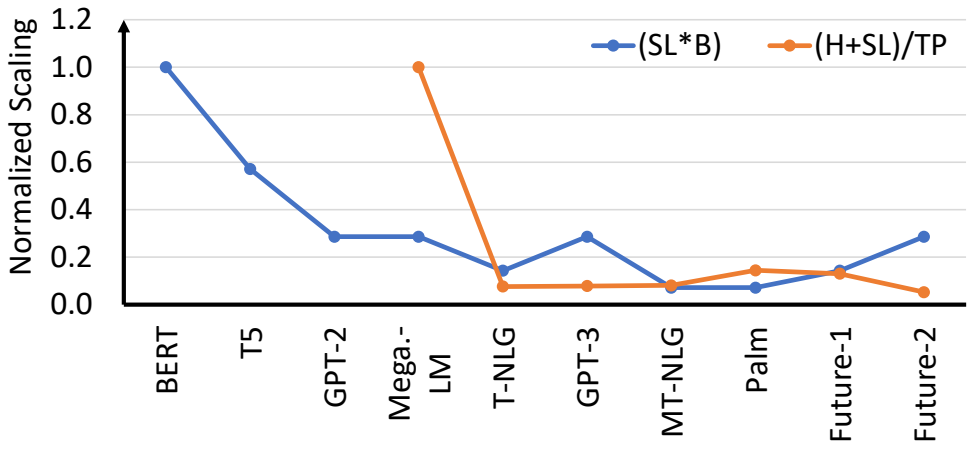


Figure 5.6: Algorithmic scaling of slack and edge.

to hide the overlapped communication. However, the extent of this edge and slack varies depending on the hyperparameters: the edge grows as H or SL increases but decreases if TP increases. Similarly, slack grows with increasing B or SL. In recent years H and SL have increased considerably across Transformer models [41, 55, 62, 167, 226, 227, 256, 263].[2] As shown in Figure 5.5, these trends are expected to continue since larger H and SL are strongly correlated with improved model quality [227]. However, model parameters scale quadratically with H and activations scale linearly with both H and SL – thus increasing Transformers' memory requirements. Figure 5.5 uses H·H and H·SL values to show memory requirement scaling for parameters and activations. These results show that if the trend of linear scaling of device memory capacity continues, the gap between available device memory capacity and models' future memory demand will increase. Consequently, using smaller B's to reduce activation sizes, and larger TP slicing to distribute parameters have become imperative to

[2]Although recent work has improved accuracy by increasing training token count instead of model size [95], scaling H is still the most widely used technique to improve model quality. Further, our empirical analysis shows that compute throughput scale faster than network bandwidth – thus increasing communication even for a fixed H.

limit memory pressure (detailed in Section 5.3.3.2 and Figure 5.8(b)). If this trend in B and TP exceeds the corresponding increase in H and SL, the resulting algorithmic edge ratios (i.e., $(H + SL)/TP$) and slack (i.e., $SL \cdot B$) can decrease, exposing additional communication on the critical path. We show this scaling in Figure 5.6, which plots compute's algorithmic slack and edge over communication for all studied Transformers, normalized to that of BERT's. Due to a considerable decrease in B (=1), compute's slack has reduced by ~75%. Similarly, if TP is scaled to fit these models (details in Section 5.3.3.2) compute's edge can decrease by ~80%.

The $SL \cdot B$ values for futuristic models increase as we project larger SLs for them while B remains at the minimum of one. In reality, SL scaling is also often limited by memory capacity and will result in the slack being constant. Consequently, model scaling and memory capacity trends are stressing compute's algorithmic edge over communication.

This algorithmic analysis also does not account for the cost of executing an operation or communicating a byte. Thus, an individual Transformer's compute ops to communication bytes ratio does not directly translate to execution time ratio, and compute may actually have no/smaller edge and slack (explored further in Section 5.3). Nevertheless, our algorithmic analysis provides insights on how evolving Transformers affect edge and slack.

5.3 Comp-vs.-Comm: Empirical Analysis

Thus far our analysis has shown that compute enjoys an algorithmic edge over communication, but this edge is being stressed as models evolve (Section 5.2). We next use empirical analysis to quantify this edge. Since an exhaustive empirical study can be expensive, we propose a strategy based on our algorithmic analysis that uses existing hardware to project Comp-vs.-Comm for any future model on future hardware.

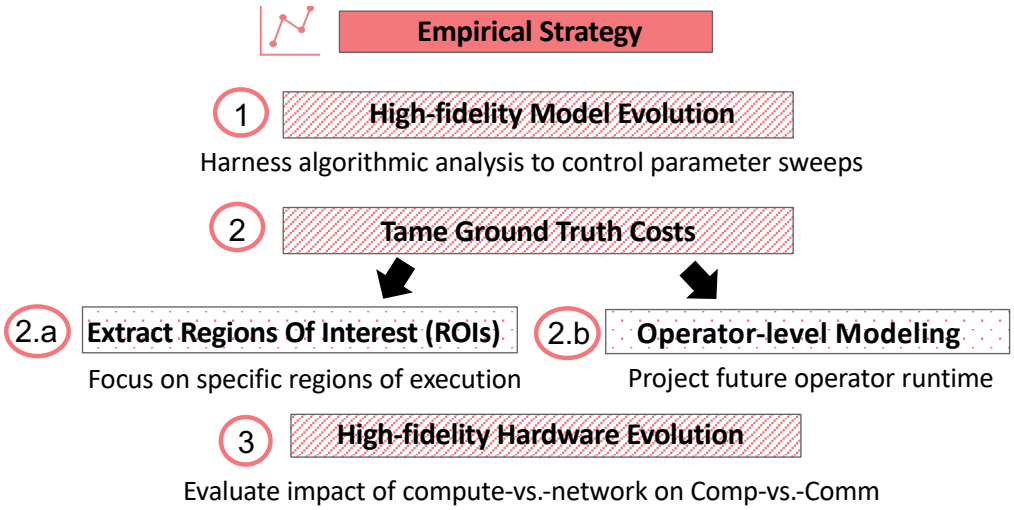


Figure 5.7: Components of proposed empirical strategy.

5.3.1 Empirical Analysis Challenges

An empirical analysis must be designed carefully because model evolution can cause an explosion of scenarios (hyperparameters) to consider and, consequently, experiments to run. This is further exacerbated when considering hardware evolution due to many hardware parameters. Thus, it is important to carefully identify variables of interest when designing experiments to study both model and hardware evolution. Moreover, even with a disciplined exploration of the hyperparameters and hardware parameters, profiling costs can still be very high, especially in scenarios requiring entire training iterations to be profiled. Thus, careful examination of the variable space, close attention to controlling profiling overheads, and high-fidelity model and hardware evolution designs are paramount to empirically study Comp-vs.-Comm scaling for future models and hardware.

5.3.2 Proposed Empirical Strategy

Next, we discuss the components (Figure 5.7) of our empirical strategy to overcome the aforementioned challenges.

5.3.2.1 Step 1: High-fidelity Model Evolution

To effectively study Comp-vs.-Comm scaling for future models, we must carefully consider model evolution. Section 5.2.5 demonstrated that historically models have scaled both the hidden dimension (H) and sequence length (SL) to improve accuracy. Other hyperparameters (batch-size B and degree of tensor-parallelism, TP) depend on a system's resources (e.g., compute and memory capacity). A naive but exhaustive exploration of such a hyperparameter space will help faithfully study model evolution. However, even after excluding unrealistic configurations (e.g., large H and large B with a small TP), it would require running an impractically large number of experiments.

We overcome this challenge by anchoring on our algorithmic analysis. Specifically, we use the Comp-vs.-Comm scaling ratios identified in Section 5.2 to design controlled experiments. For scenarios where communication is overlapped with computation (e.g., DP), since algorithmically the Comp-vs.-Comm ratio is $\mathcal{O}(SL \cdot B)$, we focus on sweeping $SL \cdot B$ for different H values to study how compute's slack advantage scales for future models. However, this still requires several different (from H, and $SL \cdot B$) iterations to be profiled. Furthermore, for serialized communication (e.g., TP), since the ratio of Comp-vs.-Comm is $\mathcal{O}((H + SL)/TP)$, we can only factor out B. We identify additional strategies to tame this exploration (Section 5.3.2.2).

5.3.2.2 Step 2: Taming Ground-truth Cost

Although algorithmic analysis helps prune the search space, further solutions are needed to reduce profiling costs. Accordingly, similar to prior work [217] we use application understanding to identify and study only specific fractions of executions where possible. When entire iteration times are required, we rely on high-fidelity operator-level models to project the

Parameter / Setup	Value
H	1K, 2K, 4K, 8K, 16K, 32K, 64K
{B}, {SL}	{1,4}, {1K, 2K, 4K, 8K}
{TP degree}, {DP degree}	{4, 8, 16, 32, 64, 128, 256}, {Any}

Table 5.1: Parameters and setup of models studied.

runtime of different Transformer configurations without actually running them. We further explain these strategies below.

Step 2.a: Region of Interest (ROI) Extraction: As discussed in Section 5.2.4, to study overlapped Comp-vs.-Comm (e.g., in DP) it suffices to extract the specific compute (e.g., GEMMs) and communication fractions (e.g., all-reduce) which will manifest for future models and profile the execution of only these regions in hardware. These controlled experiments help us study how compute's slack, to hide communication, will evolve as models scale and hardware evolves, and avoids the cost of running the entire training iteration for all configurations of interest.

Step 2.b: Operator-level Models: For serialized Comp-vs.-Comm (e.g., in TP), executing ROI regions is insufficient. To quantify how much Amdahl's Law edge compute enjoys over communication, it is necessary to study entire training iterations. However, we observe that building high-fidelity operator-level models and combining their results can help us project entire network behavior. Specifically, for every operator in the Transformer layer's execution that repeats during a training iteration (e.g., GEMMs and layer-normalization), we use algorithmic analysis to identify hyperparameters that affect its execution time. Given the operator's execution time for a hyperparameter configuration, we can project the execution time for any different set of hyperparameter values. Thus, these operator-level models project Transformer behavior for a wide variety of hyperparameter values without significant profiling costs. Moreover, our evaluation (Section 5.3.3.8) shows that these operator-level models are reliable and accurately capture the behavior of operations with changing hyperparameters.

5.3.2.3 Step 3: High-fidelity Hardware Evolution

Similar to model evolution, a disciplined hardware parameter search space is equally important. Accordingly, we identified the key drivers important to Comp-vs.-Comm scaling: compute throughput (FLOPS), network bandwidth, and memory bandwidth. Of these, we focus on the first two factors. Although communication performance is impacted by all three factors, efficient communication primitive (e.g., all-reduce) implementations are pipelined. Thus, they can overlap memory accesses with network transfers – and since network transfers usually dominate, memory bandwidth has a relatively smaller impact. Moreover, while compute operations depend on both compute FLOPS and memory bandwidth, key Transformer operations (e.g., GEMMs) are often compute-bound (e.g., Gshard reports >85% peak FLOPS utilization [146]) and have low memory bandwidth utilization [216]. Thus, we focus on compute FLOPS and network bandwidth, specifically on their *relative scaling ratios* based on historical data for GPUs from different vendors (discussed in Section 5.3.3.6).

5.3.2.4 Benefits Compared to Exhaustive Profiling

Our empirical strategy reduces execution and profiling costs (Section 5.3.3.8) of Comp-vs.-Comm analysis for future models on future hardware. First, our algorithmic analysis helps identify a subset of hyperparameters to sweep, limiting the combinations to consider ($SL \cdot B$ rather than individually sweeping SL and B). Second, the operator-level models enable projection of iteration times for many (196) different configurations using the execution and profiling of a single iteration. Finally, focusing on specific ROIs avoids executing end-to-end iterations for overlapped communication.

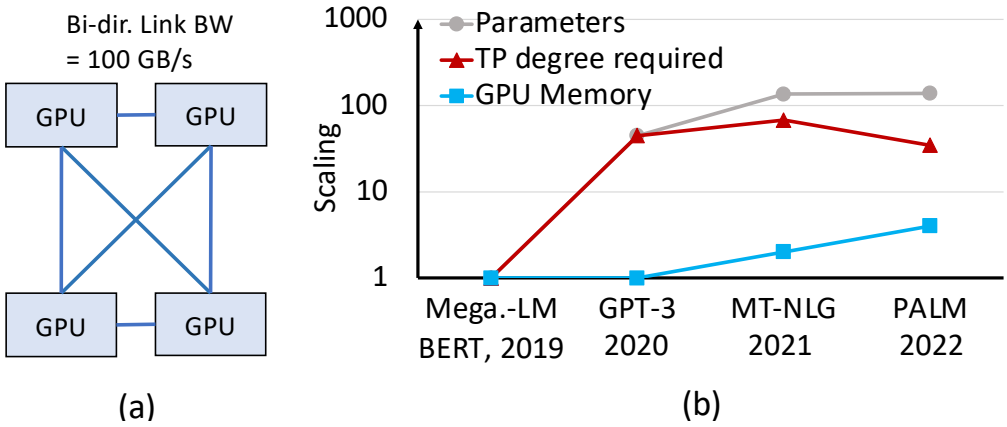


Figure 5.8: System: (a) 4-GPU node (b) TP scaling with model size.

5.3.3 Observations from Experimental Analysis

5.3.3.1 System Setup

We run experiments based on our empirical strategy using a system with an AMD Ryzen™ Threadripper™ CPU and four AMD Instinct™ MI210 accelerators (GPUs) [25] (Figure 5.8(a)), each with 64GB HBM[3]. The GPUs are fully connected using AMD Infinity Fabric™ links with bidirectional link bandwidth of 100GB/s. These links form multiple rings, providing a peak ring all-reduce bandwidth of 150GB/s. We also calibrated our system's performance and found it was similar to prior work using other commercial systems [107]. Finally, our software stack uses AMD's open source ROCm™ version 5.2 [18] with PyTorch v1.7, the rocBLAS [14] BLAS library, and the RCCL [10] communication collectives library.

5.3.3.2 Models & Cluster Setup

To study a range of Transformers (Figure 5.5) we evaluate the hyperparameter combinations and distributed setups listed in Table 5.1.

[3]Given the fast-evolving GPU space with improved ML-specific optimizations built into each generation, we use a setup with the latest available GPU for ML at the time. Thus, the GPU used for studies in this chapter differs from those in Chapters 3 & 4.

Model Setup (H, B, SL): Scaling Transformers typically involves scaling H and SL [227]. Thus, for H we examine power-of-two values up to 16K and for SL up to 2K, as they represent a wide spectrum of modern Transformers. Additionally, to project future model behavior we scale H to 32K (Future-1 with 1 trillion parameters) and 64K (Future-2 with ten trillion parameters) with SLs of 4K and 8K, respectively. For B, we consider small values of one and four. Smaller Bs (and larger TPs discussed in Section 5.2.5) are required to bridge the large gap between required and available memory capacity (Section 5.2.5). In fact, most modern larger models (e.g., MT-NLG [263] and PALM [55]) already use the smallest B of one.

Training Cluster Setup (**TP degree, DP degree**): We study a range of TP and DP degrees:

TP degree: We determine the appropriate TP range based on modern Transformer setups. We start with the 3.9B parameters Megatron-LM model (Mega.-LM_BERT), the first publicly known Transformer to use tensor-parallelism with TP of eight. To estimate the TP for a future Transformer, we consider device memory capacity and model size. Assuming a capacity of eight devices ($=base_{TP}$) is required for Mega.-LM_BERT, we estimate a larger model's TP by scaling up $base_{TP}$ by the ratio (p) of its model size compared to that of Mega.-LM_BERT. To account for potential device memory capacity increases in the same time period, we divide the projected TP by the memory capacity scaling ratio (s) in that time period. Thus, the required TP degree is $base_{TP} * (p/s)$. Figure 5.8(b) shows the scaling of Transformers, and device memory capacities, as well as the resulting scaling of TP (p/s) required to fit the Transformers, all normalized to Mega.-LM_BERT. Since memory scaling (16GB [9] to 64GB [25]) has not been proportional to Transformer model scaling (8.3B [256] to 540B [55]), TP needs to be scaled by 40-60×. With a $base_{TP}$ of eight, this implies that the required TP degree can be ~250-550. Although TP has increased over the last few years as models scale, considerable innova-

tions in interconnect technology will be necessary to realize such large TPs. Furthermore, pipeline parallelism can also be relied on to limit TP. Consequently, we study a range of TP values up until 256.

DP degree: Our data-parallel empirical analysis is largely agnostic to DP degree. Unlike TP, compute FLOPS and overall communication size are not dictated by DP degree. Furthermore, while we use a four-GPU ($N = 4$) setup, it also provides us with a reasonable, albeit conservative, estimate of communication time on larger setups because (ring) all-reduce traffic scaling is small at large device counts ($(N-1)/N$ is ~ 1 for large N). However, increasing device count also increases synchronization cost between devices, causing the actual communication time to be slightly higher. Furthermore, DP training is usually setup on large-scale multi-node, often heterogeneous, systems with slower inter-node links. Since we did not have access to any of these machines, we instead optimistically estimate the communication times using intra-node links and discuss the implications of this in Section 5.3.3.7.

5.3.3.3 Profiling Setup

For the overlapped communication analysis, as discussed in Section 5.3.2.2 we extract relevant regions from training iteration (compute and communication operations) and execute only these relevant regions for all possible hyperparameter combinations under consideration (Table 5.1). Although in reality they execute concurrently, we execute and study them in isolation to avoid interference slowdowns from shared resources and to understand their optimal characteristics in isolation. For serialized communication analysis, we first profile BERT [62] training iterations on a single GPU as a baseline. Next, we employ our operator-level models (Section 5.3.2.2) to project training runtime for hundreds of Transformer configurations. Finally, we use rocProf [12] to measure GPU kernel execution times.

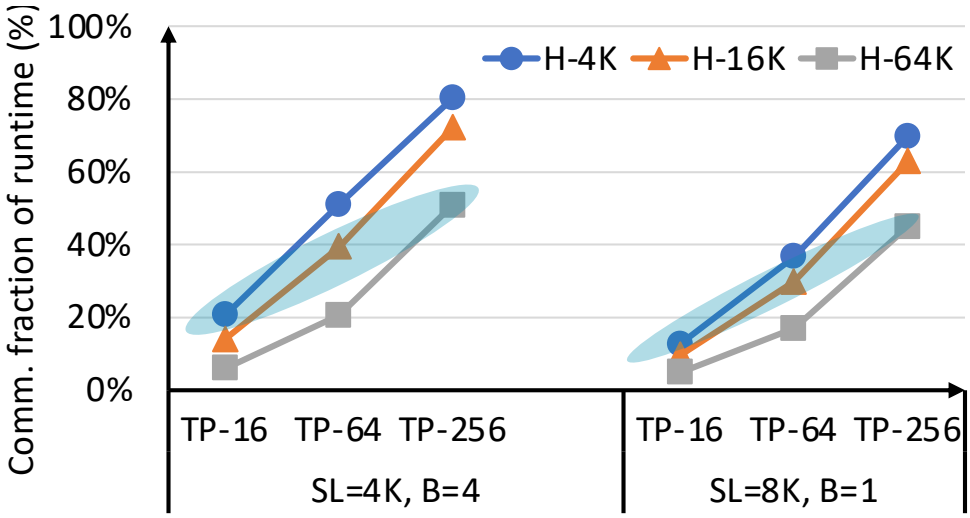


Figure 5.9: Fraction of serialized comm. time.

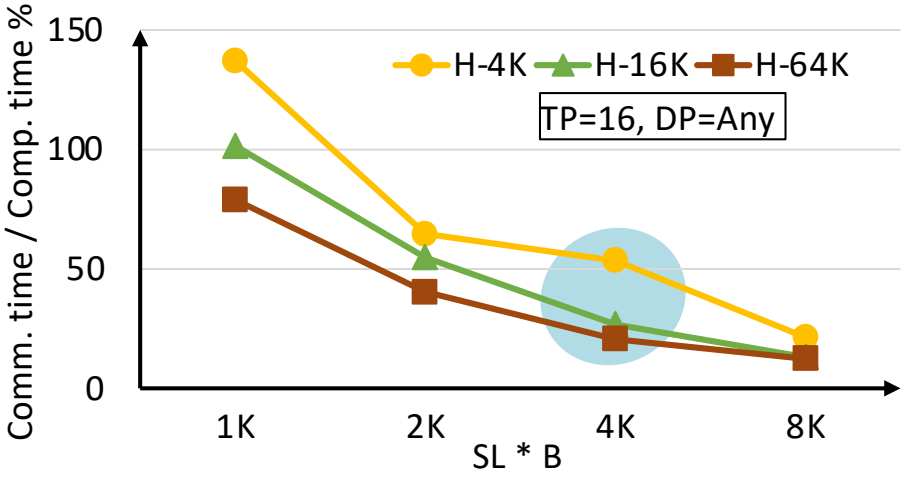


Figure 5.10: Overlapped comm. as a percentage of comp. time.

5.3.3.4 Amdahl's Law Edge Analysis

Using our empirical strategy (Section 5.3.2), hyperparameter trends (Section 5.2.5), and estimated TP values (Section 5.3.3.2), we project the pro-

portion of serialized communication as compared to compute. Figure 5.9 shows the fraction of Transformer training time spent on communication for a subset of varying H, SL, and TP values. It includes a medium Transformer (~T-NLG [167]), one of today's largest publicly-known Transformer models (~PALM [55]), and a futuristic Transformer (Future-2). For a fixed H and $SL \cdot B$ (a line), the communication proportion increases with increasing TP. Conversely, with fixed TP it decreases with either an increasing H or SL. These trends mirror our algorithmic takeaways (Section 5.2.3). Furthermore, the communication fraction is considerable and increases as models scale. Models of different sizes require different TP values to train (discussed in Section 5.3.3.2). While a TP degree of 16 can potentially suffice for a model with $H = 4K$ (e.g., T-NLG), it has to be scaled for larger models (e.g., TP of 64 for H of 16K). These parameter combinations are highlighted in blue in Figure 5.9 and show that communication increases to a considerable 50% of the execution time for a model with $H = 64K$ (Future-2). This trend also correlates with our algorithmic takeaways (Section 5.2.3): with SL constant, and similar scaling of H and TP, the denominator of $(H + SL)/TP$ scales much more, causing compute's Amdahl's Law edge to decrease.

5.3.3.5 Slack Advantage Analysis

We estimate the fraction of time that communication is overlapped with compute using our empirical strategy (Section 5.3.2) and hyperparameter trends (Section 5.2.5). This helps estimate both how well compute's slack advantage can hide communication costs and how this slack scales. Moreover, these estimates hold irrespective of the degree of DP. Figure 5.10 shows that the overlapped time decreases as the product of SL and B ($SL \cdot B$ in Figure 5.10) increases, similar to our algorithmic takeaway in Section 5.2.4. Additionally, the overlap percentage is higher at smaller H, causing smaller remaining slack. Our algorithmic analysis did not

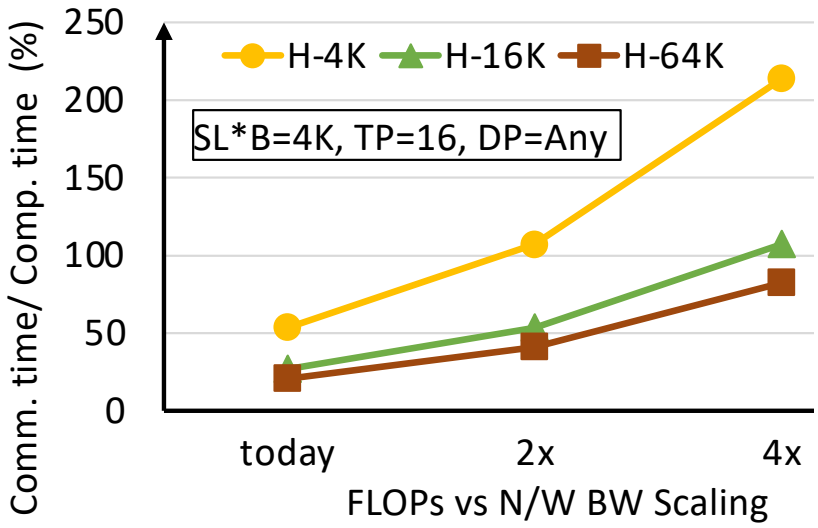


Figure 5.11: Hardware evolution impact on overlapped comm. as a percentage of comp. time.

account for this since it is an artifact of hardware execution. Additionally smaller H, and thus smaller communication sizes do not fully use the network bandwidth capacity of devices that larger sizes can. This results in a sub-linear increase in communication costs until the network bandwidth saturates. Conversely, compute operations are large enough to saturate compute FLOPS. Thus, the slower communication at smaller sizes creates a larger overlap and leaves less compute slack.

Furthermore, the communication overlap percentages are very high, ranging from 17% to 140% for the range of H, SL, and B values, with a fixed TP degree of 16 and irrespective of the DP degree. In particular, the highlighted blue region shows that for the common $SL \cdot B$ value of 4K, across a range of models, communication forms 20-55% of compute time, leaving smaller compute slack. Moreover, this percentage is likely to be much higher if this communication occurs in large multi-node setups with slower network links than the high-bandwidth intra-node links we study.

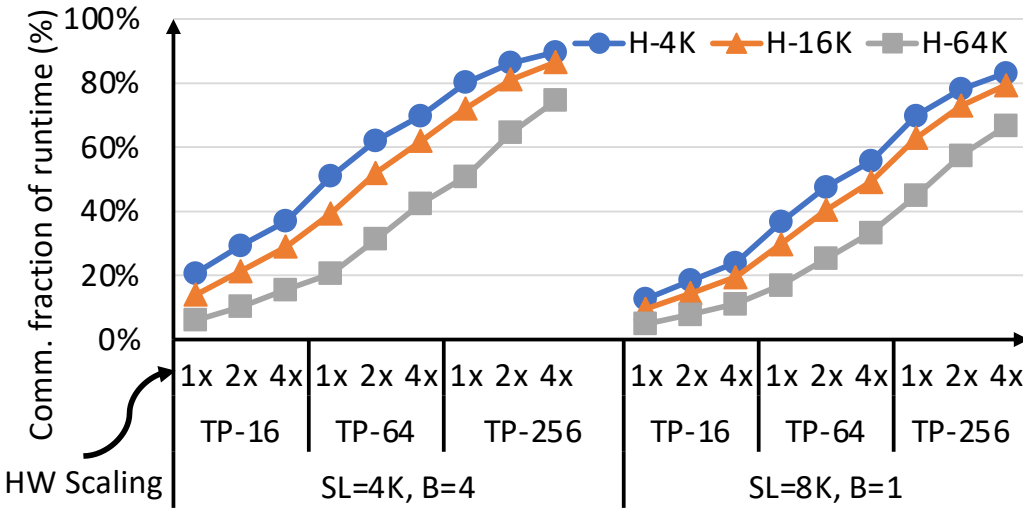


Figure 5.12: Impact of hardware evolution on fraction of serialized communication time.

5.3.3.6 Future Hardware Analysis

Thus far we have estimated the Comp-vs.-Comm costs while training Transformers on current systems. However, evolving hardware can change these estimates and shift application bottlenecks. Thus, we next estimate the Comp-vs.-Comm costs for future systems, using past hardware trends to help inform future system design. First we estimate the relative scaling of compute FLOPS versus network bandwidth, which we call *flop-vs.-bw*. This value varies across GPU generations as well as vendors. Between 2018 and 2020, compute FLOPS scaled by ~5× [188, 194] and ~7× [9, 21], while corresponding network bandwidth scaled only by ~2× [188, 194] and ~1.7× [9, 21], respectively. This implies that compute FLOPS have scaled relatively more than network bandwidth: by ~2-4×. We use these relative *flop-vs.-bw* ratios to scale the compute time estimated in Sections 5.3.3.4 and 5.3.3.5 and project its resulting slack advantage and Amdahl's Law edge over communication. Reducing precision can further disproportionately scale compute FLOPS more than network, causing this ratio to be much higher (discussed further in Section 5.5, *Number-formats*).

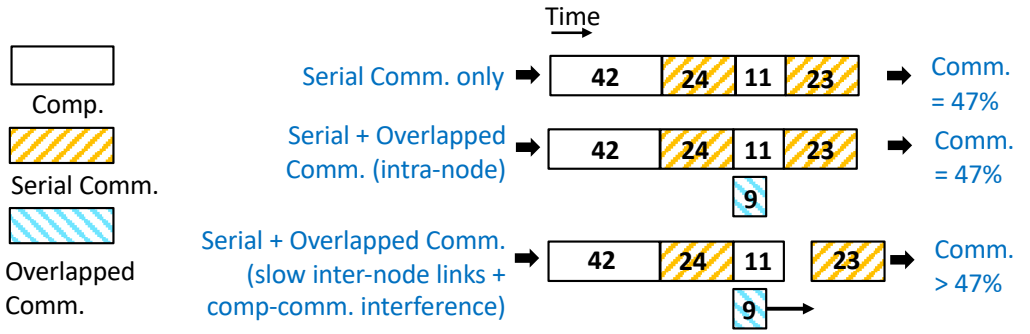


Figure 5.13: Overall Comp-vs.-Comm Case Study. Setup: H=64K, B=1, SL=4K, TP degree=128, flop-vs.-bw scale=4x.

Figure 5.12 shows that, with 2× and 4× flop-vs.-bw scaling, serialized communication starts to dominate training execution. The range increases from 20-50% to 30-65% and 40-75%, respectively, for the configurations in Section 5.3.3.4. Similarly, compute acceleration also reduces, or even eliminates, compute's slack to overlap communication. Figure 5.11 shows that the overlapped communication is 50-100% and 80-210% of compute time with 2× and 4× flop-vs.-bw scaling, and communication is exposed (i.e., on the critical path) in many cases (when >= 100%). Furthermore, these communication percentages will increase in inter-node setups (discussed in Section 5.3.3.7). Thus, if similar trends in hardware evolution continue, communication will become a critical bottleneck in training Transformers.

5.3.3.7 End-to-end Comp-vs.-Comm Case Study: Combining Serialized & Overlapped Communication

Figure 5.13 shows the combined impact of both TP and DP for a large futuristic Transformer model. 47% of time is spent on serialized communication while 9% is spent on overlapped communication. Since the latter is completely hidden by independent (backprop GEMM) computations, 47% of the overall communication is on the critical path.

Lower inter-node communication bandwidth and interference slow-

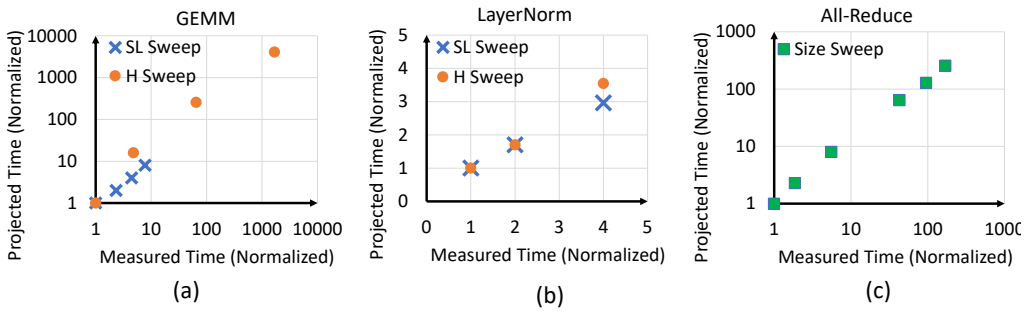


Figure 5.14: Effectiveness of Operator-level modeling.

down also affect overlapped communication (~8× [235]). The former is pertinent since portions of DP's overlapped communication may be sent over inter-node links. The latter is pertinent since communication can potentially slow down due to interference during its concurrent execution with compute [235]. The third scenario in Figure 5.13 shows their impact – DP-directed communication is no longer completely hidden. Thus, with TP-directed communication serialized and DP-directed communication only partially overlapped, total communication will become a larger bottleneck for future Transformer training.

5.3.3.8 Evaluating Operator-level Model

Hardware execution of all models and system setups can provide a more accurate Comp-vs.-Comm analysis. Although ROI extraction makes this possible for the overlapped communication study in DP, measuring the end-to-end model breakdown in order to evaluate serialized communication for all configurations is impractical. Thus, to study the end-to-end breakdown of a future model's training execution we project the runtime of all its individual components using our operator model (Section 5.3.2.2). Here we evaluate the approach's effectiveness and benefits:

Accuracy: To evaluate the operator-level model's effectiveness, we compare the projected runtimes of operations and communication against those measured on hardware while sweeping hyperparameters and data size, respectively. Figure 5.14 shows this comparison for three operators which

cover a spectrum of Transformer hardware characteristics: compute-bound GEMMs, memory-bandwidth bound LayerNorm, and the all-reduce communication collective. For GEMM and LayerNorm, we show the projected and measured execution times for different hyperparameter (H, SL) values. For all-reduce, we compare them for a range of array sizes. These results are normalized by the measured execution time of the operation (or communication) using the base hyperparameter (or array size) used for projections.

Figure 5.14(a) shows how linearly scaling Transformer GEMMs' runtimes with SL accurately captures their hardware behavior as SL varies. Similarly, scaling GEMM's runtime quadratically with the H (layer width) captures hardware trends with increasing H. However, individual errors in projecting GEMM runtimes are not negligible as operator efficiency changes with input size. Generally, GEMM efficiency improves as input sizes increase, causing runtime scaling ratios to be smaller than the ratios of input sizes. This pattern continues until GEMMs achieve near-peak efficiency, after which runtime scaling becomes analogous to input size scaling. Thus, errors with projecting future GEMM runtimes using a small operation size as the baseline can be large. Although we use BERT as the baseline model, its GEMMs do not achieve peak efficiency. This results in the projected GEMM runtimes for future models to be higher, as demonstrated in Figure 5.14(a). Although the smaller-than-projected GEMM runtimes suggest that the proportion of TP-related communication for larger models is slightly higher, it does not alter our main insights. For example, while the error may improve by using a larger baseline model, and thus operation sizes, Figure 5.14's trends and our key takeaways will still hold. Overall, across all studied GEMMs, the model projects runtime with an error of ~15%. Similarly, Figure 5.14(b) shows we accurately model LayerNorm's runtime, which is linear with both SL and H: ~7% geomean error. Finally, Figure 5.14(c) shows our model accurately models

all-reduce trends as data size varies: ~11% geomean error. Although efficiency also impacts the projections of these operators (e.g., due to better memory and network bandwidth utilization), its impact is small since they are usually close to saturation in the baseline models.

Profiling Speedups: Finally, exhaustively studying hundreds of configurations (parameter combinations from Table 5.1) without actually executing them saves considerable profiling time and effort. Specifically, our strategy avoids executing ~198 different Transformers (some very expensive), reducing profiling costs by three orders of magnitude (2100×) compared to serialized Comp-vs.-Comm for the 198 configurations. We also avoid executing end-to-end iterations, specifically the forward propagation, to estimate the overlapped Comp-vs.-Comm costs. This speeds up profiling by 1.5×.

5.4 ML/System Evolution Recommendations

Our Comp-vs.-Comm analysis demonstrates that communication is starting to become a considerable bottleneck for distributed training. Here we discuss some promising techniques that stand to tackle this challenge and also discuss how our analysis influences their potential improvements.

5.4.1 System-aware ML Evolution

Design of novel DNNs is often influenced by constraints of the underlying hardware and vice-versa (e.g., number formats in ML).

Comp-vs.-Comm influence: Our analysis shows that stressing certain hyperparameters more than others (e.g., scale SL more than H) stands to strengthen compute vis-a-vis communication. This is so, as first, scaling SL improves both the edge and the slack compute has over communication (Section 5.2). While scaling H helps increase compute's edge, it stresses memory capacity (for parameters) quadratically. As such, model evolution which scales

SL more than H is likely to have a lower communication fraction than vice-a-versa. Furthermore, existing works have also shown that scaling SL can improve model accuracy / results across various applications [95, 242, 290]. Thus both accuracy and efficiency stand to benefit from scaling SL.

5.4.2 Communication Offloads/Fusion

Some techniques offload communication from an accelerator (e.g., GPU) to a co-processor (e.g., ASIC, FPGA, DPU) [27, 235] which are specialized to accelerate communication. They can address communication which can be overlapped with computation (e.g., data parallelism). To tackle communication on critical path, techniques that break communication abstractions and optimize for pipelining/overlap of data generation and communication can be employed [70, 107, 278].
Comp-vs.-Comm influence: Our analysis indicates that both serialized and overlapped communication are important. Consequently, a judicious combination of both offload and fusion will be necessary for future Transformers. We show this in Chapter 8 with T3.

5.4.3 Processing-in-memory (PIM)

Several commercial realizations of Processing-in-memory (PIM), which push compute units closer to memory, have recently emerged that can accelerate and reduce memory traffic resulting from memory-heavy operations [250, 261].
Comp-vs.-Comm influence: Lowering communication-induced memory traffic can help improve efficiency. This can be enabled by efficient support for in-memory atomics with PIM which can lower memory traffic required for the reduction computation in an all-reduce primitive. This also stands to lower interference in memory between communication and computation executing on the accelerator. We implement this in Chapter 8 as part of T3.

5.4.4 Processing-in-network (PIN)

Processing data during traversal is also promising [241, 251]. Specifically, techniques that enhance existing network switches to execute collectives [81, 130, 153] halve the network's transmitted bytes compared to a bandwidth-optimal ring all-reduce [35]. This is because devices only send their copies of data to the switches once and receive the reduced version from the switches. Unlike in software-based ring/direct all-reduce approaches where devices send and receive arrays twice; for reduce-scatter and all-gather. However, PIN-based techniques are limited to topologies with switches.

Comp-vs.-Comm influence: As switch-based collectives are limited in their bandwidth benefit (~2x), our analysis shows that judiciously combining them with fusion will be necessary for future Transformers given the fraction of execution time bottlenecked by communication.

5.5 Discussion

Beyond DP & TP: While we focus on DP and TP, communication from other distributed techniques can be folded into our analysis. *Mixture-of-experts* (*MoE*) sparsely activate parts of a network to reduce computational costs [69, 228]. Besides DP and TP, MoEs also deploy expert parallelism with additional serialized all-to-all communication – which can be incorporated into our serialized communication analysis. Overall, due to this additional communication and reduced computation, MoEs potentially increase the fraction of communication even further. *Pipeline Parallelism* (*PP*) partitions a model to assign a subset of layers to each device such that devices execute their layers in a pipelined manner [99]. We do not focus on PP as it adds pipeline bubbles which either degrades efficiency or requires a large number of micro-batches, which add memory pressure and degrade model quality. Nevertheless, our overlapped communication

methodology/analysis can be extended to include the peer-to-peer communication of activations between PP devices. Finally, the Fully Shared Data Parallel (FSDP) technique combines DP's and TP's benefits by distributing weights amongst DP devices and asynchronously gathering them just before layer computation. This adds additional overlapped communication in both forward and backward execution passes, which we could extend our slack analysis to examine.

Large/Other System Setups: Large Language Model (LLM) training usually uses both DP and TP: TP is employed within nodes and DP across nodes. Thus, DP-related communication occurs over slower inter-node links (e.g., Ethernet). While our empirical analysis focuses on intra-node setups with faster network links, it can be extended to encompass other network types. Nonetheless, our algorithmic analysis for identifying slack remains applicable and can also be utilized to reduce the time and effort needed for empirically estimating slack/overlap in large clusters.

Finally, while our empirical analysis uses a single GPU/hardware type, it can be extended to consider other hardware types (e.g., accelerators or GPU systems from other vendors). Recent work has shown that Transformers operations runtime can be calculated using their sizes and hardware specifications such as FLOPS, memory bandwidth, and intra-node bandwidth (albeit with efficiency considerations) [169]. Thus, our operator-level model could also be enhanced to project runtimes for another device using the ratios of the devices' specifications.

Large System Memory: Techniques to place the model state in system memory (CPU-attached DDR, NVMe memory) can help reduce accelerator memory pressure [198, 230, 240]. While this limits the required model-parallel (TP or PP) degrees and inter-accelerator communication, it can increase training time due to the limited compute capacity of fewer devices. Nevertheless, our methodology can be used to model communication for the resulting TP and be extended to include additional overlapped

communication between CPU/NVMe and accelerator memory.
Number-formats: Number formats with lower number of bits [166, 247] have both computational and communication benefits during training. Our analysis and methodology, although for state-of-art mixed-precision training, are largely agnostic to the formats. Further, compute time can potentially decrease more than communication at smaller number formats. As such, the key takeaways of our analysis will likely carry over to these alternate formats.
Fine-tuning & Inference: Our takeaways hold for fine-tuning since it uses the same techniques and model as pre-training. Conversely, inference has much smaller memory requirements [79, 93] and thus can avoid distributed setups and communication. If deployed in distributed setups [210, 228], our methodology and takeaways will continue to hold.
Other DNNs: While we focus on Transformers due to their generality, our proposed methodology can be translated as is and/or extended to other DNNs. Specifically, our insights on ROI extraction and operator-level projections can be easily translated to other models. Similarly, our algorithmic analysis-based empirical strategy can be extended for other DNNs.

5.6 Related Work

DNN Characterization DNNs, especially Transformers, are an important application domain that are driving system optimizations. Consequently, there have been several works on benchmarking and characterizing them [4, 82, 164, 237, 274, 289, 293, 295, 295]. However, unlike our work, these focus on compute bottlenecks in single-device DNN executions and thus do not characterize the communication costs that arise in multi-device, distributed setups. Instead, we focus on characterizing the relative cost of communication compared to compute operations and

show that more communication-focused innovations will be needed in the future.

Studying & Accelerating Communication: Other works study and/or optimize for communication in distributed setups [57, 130, 216, 235, 278]. However, unlike our work they do not examine how communication costs, and thus benefits of their optimizations, evolve across different Transformers, hardware capability, and different distributed techniques. While some works [63, 191] examine the throughput impact of sweeping a subset of hyperparameters, they either do not include in-depth characterization that examines the behavior or are on a single device.

5.7 Chapter Summary

Scaling of Transformers models and their datasets has necessitated very large-scale distributed setups, which raises the key question: *how will compute vs. communication* (***Comp-vs.-Comm***) *scale as models scale and hardware evolves?* In this chapter, we conducted a multi-axial (algorithmic, experimental, hardware evolution) analysis of Comp-vs.-Comm scaling for Transformer models. Our system-agnostic, algorithmic analysis highlighted that while compute has enjoyed an edge over communication, future model and hardware trends are likely to make communication dominant soon. We also empirically studied Comp-vs.-Comm for future Transformer models as hardware evolves. By extracting specific regions of interest and modeling future operator runtimes, we enabled the study of hundreds of future Transformers/hardware scenarios with 2100× less profiling costs. These experiments validated that communication will play an increasingly large role (40-75%) in a distributed training setup as models scale. Finally, we discuss how our analysis influences some promising techniques and technologies. This chapter's findings serve as the basis for our proposal in Chapter 8.

5.7.1 Key takeaways from SQNN characterization

Here we summarize some of the key takeaways from characterizing sequence-based models (from this and previous Chapters 3, 4) that subsequent chapters (Chapters 6, 7, and 8) address.

5.7.1.1 Memory-bound operations

Chapter 4 highlights the importance of non-GEMM (optimizer update, other element-wise) operations (Section 4.2.3); about 30-40% of BERT training time is spent on memory-intensive operations. Furthermore, it showed how reducing batch size (common for data-parallel training), reducing precision, and increasing layer size (Section 4.3.2) can make optimizing for non-GEMMs even more important. Moreover, as GEMMs speed up, the remaining memory-intensive operations will become the bottleneck. We address this in Chapter 6.

5.7.1.2 Ineffective Operational Parallelism

GEMM operations which constitute the major proportion of runtime in both RNN and attention-based SQNNs can often be small with low GPU utilization as shown in Sections 3.5 and 4.2.2. While this can be alleviated by leveraging the abundant parallelism in networks and training techniques, we show in Chapter 7 that executing such operations concurrently on the GPU is not always beneficial. We study issues with exploiting and improving concurrency amongst GEMMs in detail in Chapter 7.

5.7.1.3 Exposed inter-device communication

As shown in this chapter, multi-device training can add considerable communication overheads. While in data parallelism, the cost of this communication and reduction can often be hidden with the execution of model backpropagation, as models and hardware evolve, these costs can get exposed. Communication in tensor slicing is already in the critical path (between layers) of model execution as shown in Figure 2.3.3. As models

scale and require larger distributed setups, these communication costs start to dominate model execution times. These exposed communication overheads leave GPUs idle and cause sub-linear scaling of application throughput with multiple devices. We address this with our proposal T3 in Chapter 8.

6 NEAR-MEMORY COMPUTING FOR OPTIMIZER UPDATES

Chapter 4 highlighted the importance of non-GEMM operations (Section 4.2.3): about 30-46% of BERT training time is spent on memory-bound operations with low operation-to-byte ratios. These include the optimizer update algorithm responsible for updating all model parameters at the end of each training iteration (Section 2.4.2) as well as other interleaved element-wise operations required through the network. While the runtime of element-wise operations scales linearly with model size, those of optimizer updates scale quadratically due to the quadratic increases in model parameters in larger models (Takeaway 4-11). Furthermore, optimizer updates are always performed at a higher precision despite the reduced precision of forward and backward propagation through layers. Therefore, the runtime contribution of optimizer updates can increase with the current trend of scaling model parameters and reducing precision and thus, is an important bottleneck in training Transformer models.

To address this bottleneck, in this chapter we first explore a key software optimization, *kernel fusion* (Section 2.4), which can reduce expensive data movement by re-using more data in on-chip memories. However, we find kernel fusion only helps optimizer updates to a certain extent such as when there is a series of element-wise operations with a producer-consumer relationship. Optimizer updates access multiple large arrays that require DRAM accesses and cannot be avoided by fusion. To overcome this challenge, we propose to leverage emerging near-memory computing (NMC) devices and offload these memory-intensive update operations to ALUs near memory. We show that the sequence of highly parallelizable operations in optimizer algorithms lends well to the NMC architecture with ALUs associated with each DRAM bank and requires only a few cross-bank synchronizations. Our evaluation shows that the optimizer algorithm, LAMB (used by BERT) can be sped up by at least 3.8× which

results in a 9% speedup of end-to-end BERT training. The benefits further increase with enhancements to NMC ALUs as well as by considering a more realistic GPU baseline in our evaluations. This chapter details an optimization briefly discussed in the paper, *Demystifying BERT: System Design Implications*, published in IISWC 2022 [216].

The relevant background for this chapter is provided in Chapter 2. The rest of this chapter is organized as follows: Section 6.1 discusses and evaluates a common software optimization, kernel fusion. Section 6.2 provides an overview of a system with near-memory computing (NMC). Sections 6.3 and 6.4 demonstrate and evaluate LAMB's execution using NMC. Finally, Section 6.5 discusses related work and we conclude with a summary in Section 6.6.

6.1 Kernel Fusion

Fusion combines two or more consecutive GPU kernels, potentially with a producer-consumer relationship, into a single one. It improves performance by preventing data from being flushed into global memory between kernel calls [8, 91, 132, 258, 259], thus reducing duplicate memory accesses [31, 71, 72, 148, 187, 260, 264, 275]. Thus, data reuse across operations is often directly correlated with improved performance from their fusion. Data-intensive phases in BERT (e.g., GeLu, DR+RC+LN, and Scale+Mask+DR+Soft in Section 4.2.3) in which the output of one operation feeds into the next are perfect scenarios for applying kernel fusion.

The LAMB algorithm described in background Section 2.4.2 and Figure 2.7 is implemented as two stages in [184]. LAMBStage1 determines per-layer update values and learning rate multiplier using momentum (m) and velocity (v) states from past iterations, and gradients of the current iteration (all of the same size as the model parameters being updated).

This stage performs multiple element-wise add, multiply, divide, scale, and square-root operations on these parameters. LAMB then performs the L2 Norm (reduction) of the per-layer update values which is used by the second stage (LAMBStage2 in Figure 2.7) to update model weights using multiple element-wise operations. LAMB operations in each stage are already fused in PyTorch [184]. These two stages are executed for each layer and access the corresponding layer's data (weights, gradients, and optimizer parameters).

Further fusion of the algorithm across stages (but for a single layer) can be complex. This is because it requires fusing the L2 norm operations with preceding element-wise operations which can be non-trivial and expensive as it would involve data reduction across multiple GPU work-groups. Furthermore, there is little benefit from fusing LAMB operations of different layers. This is evident in Figure 6.1 which shows the impact of fusion on kernel counts, runtime and memory accesses. Fusion is very effective for LayerNorm; runtime and memory traffic scale similar to kernel count (by $6-8\times$) implying high data-reuse opportunities across the unfused kernels. However, for Adam,[1] the reduction in memory accesses and runtime ($6-8\times$) is not proportionate to that of kernel count ($\approx 250\times$). This is because most of these kernels access independent data - data corresponding to the different model layers without any producer-consumer relationship or reuse.

6.2 Near-Memory Computing

Near-memory computing (NMC) can help accelerate BERT's memory-intensive phases. It performs operations using specialized ALUs that are part of the main memory thus avoiding additional latency and energy to read and write data to and from memory [6, 125]. It further provides very

[1] Adam is an alternate to LAMB; we chose Adam for this study because its unfused and fused versions were publicly available.

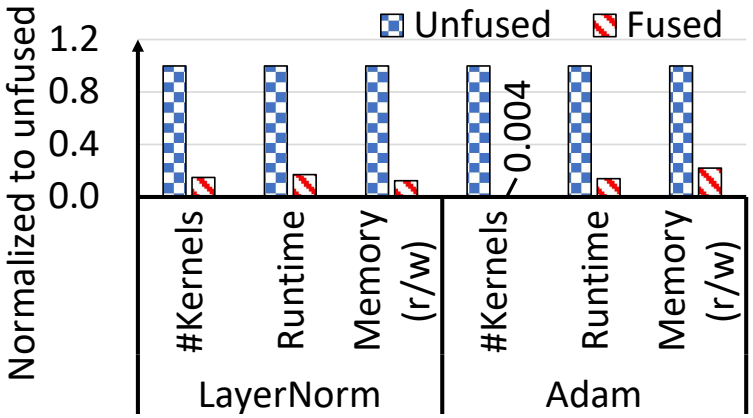


Figure 6.1: Impact of fusing kernels vs. non-fused serial execution

high bandwidth accesses to data from the ALU units. Furthermore, it also overcomes the capacity issue that on-chip memory faces, as computations can span all the banks in memory, thereby providing the illusion of an extremely large on-chip cache. In this section we describe and evaluate one way of leveraging NMC for BERT's memory-intensive phases.

6.2.1 Enhancing GPU with NMC

We consider a system where the compute-intensive phases such as GEMMs are executed on the GPU, as is done today, and only the memory-intensive operations are offloaded to units close to memory. While there are several memory-intensive operations in Transformer networks, we mainly focus on the optimizer algorithms, i.e., LAMB which is used for BERT training. We focus on these algorithms because they consist of a sequence of element-wise operations (detailed in Section 4.2.3) and are usually invoked towards the end of a training iteration after the GPU compute units have made all their memory updates. Thus, executing them on NMC units may not require frequent synchronization between the NMC and GPU compute units, which can be expensive. Moreover, as described in Section 6.1, these algorithms access large amounts of disjoint data with no producer-

consumer relationship and thus provide no data reuse opportunities. Thus any additional kernel fusion cannot reduce data accesses to the memory (Section 6.1).

6.2.2 NMC System Details

Placement: We consider a near-bank NMC architecture which, as discussed in background Section 2.5.2, has a good balance between bandwidth advantage and costs. Placing ALUs at each bank, enabling parallel access to all (or many) banks, leads to fewer ALUs and, thus, reduced cost and increased memory capacity accessible by each ALU. Further reductions in ALU count can be achieved by sharing ALUs among multiple banks. However, reduced ALU count limits performance as fewer operations can occur in parallel. A more thorough discussion of design tradeoffs can be found in prior works focused on NMC [6, 89, 125, 145]. Here, we consider a balanced design point with ALUs at each bank. This is similar to recent proposals by major memory vendors [125, 143, 145].

Orchestration: The NMC compute units operate on commands sent from the host. To parallelize operations across all banks, the command is broadcasted across all memory channels of DRAM. Since all banks sharing a channel operate on the same command, bits in the command used to identify a particular bank are freed up to encode ALU instructions.

ALU design: We adopt an ALU design similar to those explored by other works. It supports data width equivalent to the bank output width (e.g., 256b) and supports operations on multiple smaller data words in a SIMD fashion (e.g., 16b, 32b). It has temporary storage or registers to store intermediate values. The source of data to an ALU can thus be either memory, an immediate (constant) value from GPU, or these registers. The ALU supports the basic operations necessary for ML operations: *add, mul, shift and scale* in all floating point and integer formats.

Data mapping: Having compute units on each bank requires all data

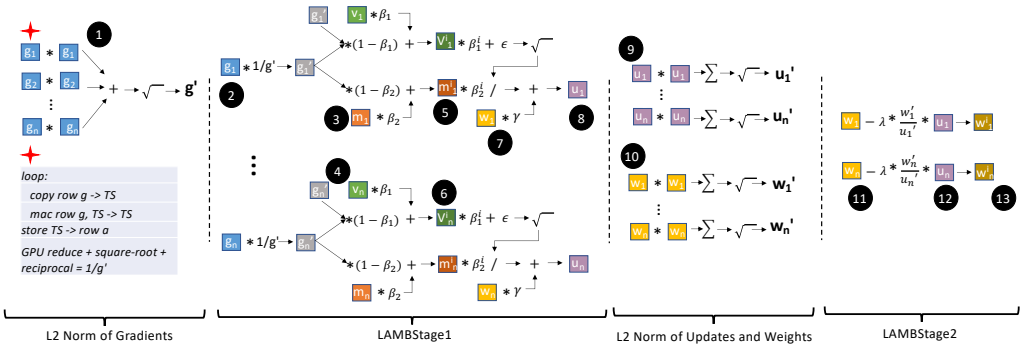


Figure 6.2: Operations in LAMB algorithm with embedded NMC commands for L2 Normalization operations (TS= Temporary Storage).

corresponding to operations available locally and are aligned similarly in a bank. Modern memory systems typically use lower physical address bits to select a channel and a bank and higher address bits to select a row. Thus this can be accomplished by allocating data structures at physical page granularity that's large enough to encompass the channel and bank bit positions within the page offset. This ensures that data structures operated upon are aligned similarly across banks.

6.3 Accelerating LAMB using NMC

Figure 6.2 shows a sequence of operations in the LAMB optimizer that are executed independently for every parameter matrix/vector in the model. As shown, the algorithm reads and writes 13 parameters (① to ⑬) with few interspersed element-wise operations. While most of the operations are simple element-wise multiplies and adds, it also uses additional L2 normalization operations on all gradients, and per-layer parameter and update values. L2 normalization of an array involves squaring all elements, summing them all up and calculating its square-root. While these operations can be fused with kernels which generate the array, they are executed as a separate kernel on the GPUs. These kernels are optimized

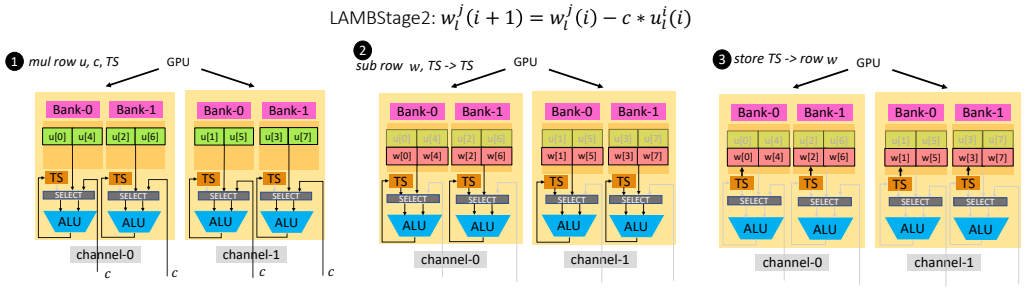


Figure 6.3: Orchestration of NMC instructions to compute the final stage of LAMB, LAMBStage2 (TS= Temporary Storage).

for fewer inter-thread and inter-block communication that are required to reduce the array. However, this causes additional memory accesses. When using NMC, while data located within a bank can be reduced efficiently, since the data arrays are spread across banks, further reduction would either require inter-bank communication or a reduction by the host GPU. To avoid complexity, we take the latter approach for the L2 Normalization of all gradients (Figure 6.2).

Figure 6.3 demonstrates the execution of a few instructions using NMC. Specifically, it shows the execution of the final stage (LAMBStage2 as described in background Section 2.4.2 and detailed in Section 4.2.3) and shown in Figure 6.2) of the LAMB optimizer in which the update matrix is scaled and subtracted from the parameter matrix. The first instruction (1) involves reading the update values from memory, scaling them and storing them in the temporary storage, the second instruction (2) involves reading the parameter values from memory, subtracting from it the values stored in the TS, and storing the updated parameter in the TS. Finally, instruction (3) involves storing the updated parameter value back in memory. Note that some of the operation in LAMB, i.e., the L2 normalizations, require reduction of data both within and across banks. To accomplish the latter, the GPU reads in the partially reduced values from each bank to do a global reduction and returns the generated value using subsequent NMC

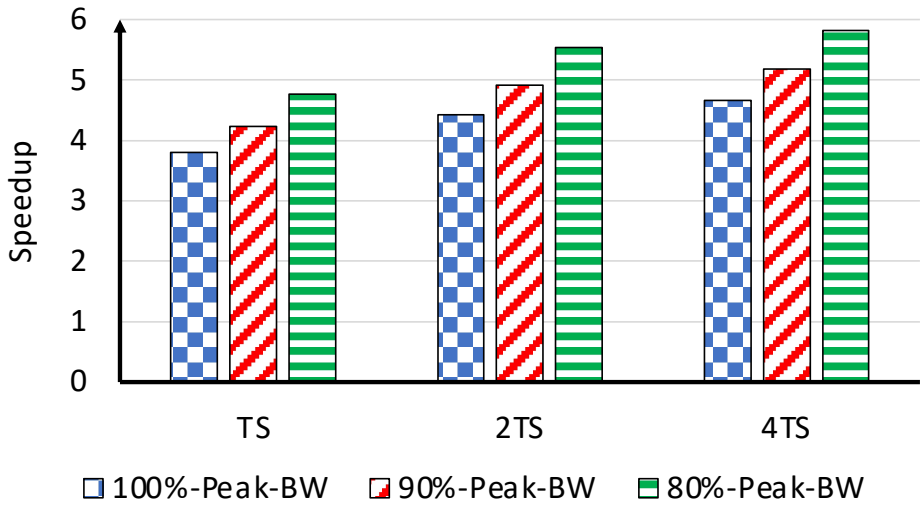


Figure 6.4: Speedup of LAMB using NMC compared to GPU (TS=Temporary Storage).

instructions.

6.4 Evaluating LAMB execution on NMC

We evaluate the execution of the entire LAMB algorithm using NMC and compare it against its execution on the GPU. We model the execution of LAMB on NMC units using common DRAM timing parameters from prior works [6, 125, 145]. Since consecutive element-wise operations such as shown in Figure 6.3 rely heavily on temporary storage, we vary the size of this temporary storage to study its impact. Finally, for comparison, we model an optimistic GPU execution comprising of only (minimal) data reads and writes to memory, not considering any temporary values that can be cached in on-chip memory. We use the peak HBM bandwidth to calculate the time taken for the memory accesses, but also consider more realistic cases when the bandwidth utilization is 80 and 90% of the peak.

Figure 6.4 summarizes the speedup LAMB can achieve via NMC. LAMB speeds up by about 3.8×, using NMC compared to an ideal GPU

execution which reads/writes at peak memory bandwidth. The speedup increases to about 4.7× and 4.6× if the temporary storage size on the NMC compute units is increased by 4×. Larger temporary storage enables larger reads/writes from DRAM rows on each activation, thus reducing the total DRAM row activations required. Furthermore, comparing against more realistic GPU memory bandwidth (80% peak) for baseline, NMC speedups are up to 5.8× and 5.7×. Given LAMB contributes to 10-20% of BERT's total iteration time (shown in Chapter 4), executing LAMB on NMC can accelerate BERT execution by up to 9-20%.

6.5 Related Work

Many works leverage NMC's data movement and performance benefits to accelerate specific ML primitives [6, 143, 145], but this is the first work that showcases NMC's benefit in accelerating weight updates algorithms that are used across all DNNs. GradPIM [125] also evaluated NMC for optimizers. However, they only evaluated simple momentum-based optimizers and focused on CNNs, which have an order of magnitude fewer parameters to update compared to NLP models.

6.6 Chapter Summary

Our characterization in Chapter 4 revealed how memory-bound gradient descent updates of billions of Transformer parameters can under-utilize modern accelerators like GPU. To overcome this, in this chapter, we offloaded these updates to near-memory compute units. Mapping a sequence of operations to memory required few expensive synchronizations with GPU compute units, and provided increased data access bandwidth along with concurrency of multiple DRAM banks. This accelerated weight updates by 3.8× for the Transformer, BERT. Finally, it considerably reduced

(~13×) expensive data movement between DRAM and GPU compute units.

7 GOLDYLOC: GLOBAL OPTIMIZATIONS & LIGHTWEIGHT DYNAMIC LOGIC FOR CONCURRENCY

Although DNN compute requirements have scaled [78, 115, 173, 256], our characterization of sequence-based networks in Chapters 3 and 4 showed that their individual operations often do not have high GPU utilization. GEMMs, which make up 30-65% of the runtime in RNNs and Transformer networks [273], only utilize 40-50% of a GPU [102, 108, 216, 256, 289]. This is further worsened as floating point operations per second (FLOPS) has significantly scaled across GPU generations (e.g., ~7× in two years [9, 21]).

A useful technique to improve this compute utilization is to concurrently execute independent operations. Programmers often expose independent operations via streams [11, 157, 185] within applications and use multi-instance deployments [22, 202]. Systems also greedily maximize the number of concurrent operations. However, naively executing all independent operations concurrently can be sub-optimal and may degrade performance. There are two key factors that impact this. First, operations must be aware of and optimized for shared resources during concurrent execution (*Operator Optimization Environment*). Second, operators whose performance degrades considerably from sharing resources must avoid concurrent executions (*Concurrency Control Logic*). We use these factors as axes (Table 7.1) to describe current GPUs and prior works that leverage concurrency.

Current GPUs optimize operator implementations for *isolated* environments. GPU libraries exhaustively optimize implementations for performance/efficiency of key operators like GEMMs. However, this tuning assumes the availability of all GPU resources, and does not consider the *global* resource environment during execution from potential intra- and inter-process concurrency. Thus, while these operators are fast and efficient when executed in isolation on devices, they can suffer significant

		Operator Optimization Environment	
		Isolated	**Global**
Concurrency Control Logic	**Static**	Current GPUs	RAMMER[161], Elastic Kernels[206]
	Dynamic	Queue/WF schedulers	***GOLDYLOC***

Table 7.1: Mechanisms to exploit concurrency on GPUs, including operators optimized in isolation vs. for global resources and static/dynamic concurrency management.

slowdowns when executed concurrently with other operators due to resource sharing and contention.

Furthermore, current GPUs *statically* manage concurrency within an application (e.g., using streams), while the hardware concurrently schedules as many operations (kernels) as possible. However, the concurrency benefits and/or opportunities available within a device can change *dynamically* with varying application inputs [217] and multiple simultaneous processes. Thus, the number of concurrent GPU kernels can be higher or lower than desired, exacerbating contention and hurting performance.

Unfortunately, both globally optimized kernel implementations and dynamically controlling concurrency are challenging to realize. GEMMs based on their input can be bottlenecked by different resources (e.g., memory, compute) during concurrency. Furthermore, and similar to baseline BLAS libraries, each GEMM of a given size requires unique kernel implementations to optimize for the bottlenecked resource. Manually identifying such implementations can be challenging. Moreover, we find (Section 7.2) that a combination of multiple factors including tensor sizes, input sizes, shapes, memory layouts, and kernel implementations dictate whether and how much concurrency is beneficial. Thus, concurrency benefits cannot be determined at runtime using simple heuristics.

Accordingly, we propose GOLDYLOC. GOLDYLOC augments kernel

tuning to identify, for each input, efficient kernels for both isolation and *global* resource environments resulting from varying degrees of concurrent execution. To find the latter GOLDYLOC tunes kernels offline with *resource constraints* which emulates various shared resource environments. Similar to the baseline, isolated-tuned BLAS libraries, where kernels have unique properties per GEMM input, tuning for concurrency also makes unique trade-offs per input to efficiently share resources while also limiting a GEMM's performance degradation. To select the appropriate kernels at runtime based on the global resource environment and concurrency, GOLDYLOC extends the kernel scheduling data structure to include pointers to globally optimized kernels. This allows the GPU's command processor (CP), the interface between software and hardware responsible for scheduling work on the GPU, to select the appropriate kernel at runtime. Moreover, we also augment the GPU's CP to *dynamically* control the executed concurrency using a predictor (trained offline) to select the appropriate concurrency to exploit – i.e., which type and degree of concurrent GEMMs to select given the available independent GEMMs and their inputs. This includes detecting when sequential execution is preferred when concurrency hurts performance. To our knowledge, GOLDYLOC is the first to combine *dynamic* concurrency control and *globally* optimized GPU kernels.

We evaluate GOLDYLOC on a real GPU using AMD's open-source BLAS infrastructure [14, 23]. Overall, across 410 GEMMs from modern DNNs, GOLDYLOC improves performance by up to 2.5× (43% geomean per app) over sequential execution and 2× (18% geomean per app) over naively exploiting all parallelism, without requiring hardware changes. GOLDYLOC also improves performance in GPUs with explicit resource partitions [202], and GOLDYLOC's benefits increase with reduced precision and as FLOPS scale, underscoring its importance given hardware scaling trends. This chapter is based on the paper, *GOLDYLOC: Global Op-*

timizations & Lightweight DYnamic LOgic for Concurrency, which is currently under submission.

The relevant background for this chapter is provided in Chapter 2. The rest of this chapter is organized as follows. Section 7.1 motivates the need to optimize for concurrent GEMMs on GPUs. In Section 7.2, we describe challenges with improving GEMM libraries and GPU runtimes for efficient concurrent GEMMs. Section 7.3 provides the details of our proposal GOLDYLOC. We describe the methodology used to evaluate the efficacy of GOLDYLOC in Section 7.4 and show results in Section 7.5. In Sections 7.6 and 7.7 we discuss GOLDYLOC's applicability/extensions and related work, respectively. Finally, we summarize and conclude in Section 7.8.

7.1 Motivation

7.1.1 Scaling GPUs and low utilizing GEMMs

GPUs compute cores (e.g., AMD CUs, NVIDIA SMs) and, therefore, their peak achievable FLOPS, have scaled considerably; for example, between 2018 and 2020, FLOPS scaled by ~7× [9, 21]. However, application utilization of these GPUs, especially for NLP-based DNNs, has often remained low. GEMMs GPU utilization can be low when the input/output matrix sizes (Figure 2.5(a)) are small. This is common in DNNs (Section 2.2) due to their training/inference setup and/or algorithmic properties, including lower input batch sizes, short Transformer input sequences, and sequential RNN input token processing. Reducing input batch sizes helps memory capacity requirements, improves convergence during training, and helps meet application deadlines during inference [104]. However, smaller input batch sizes also limit matrix sizes, hurting utilization and throughput (e.g., only up to 23% of TPU peak throughput [116]). Short Transformer input sequences (e.g., length 512 BERT attention GEMMs

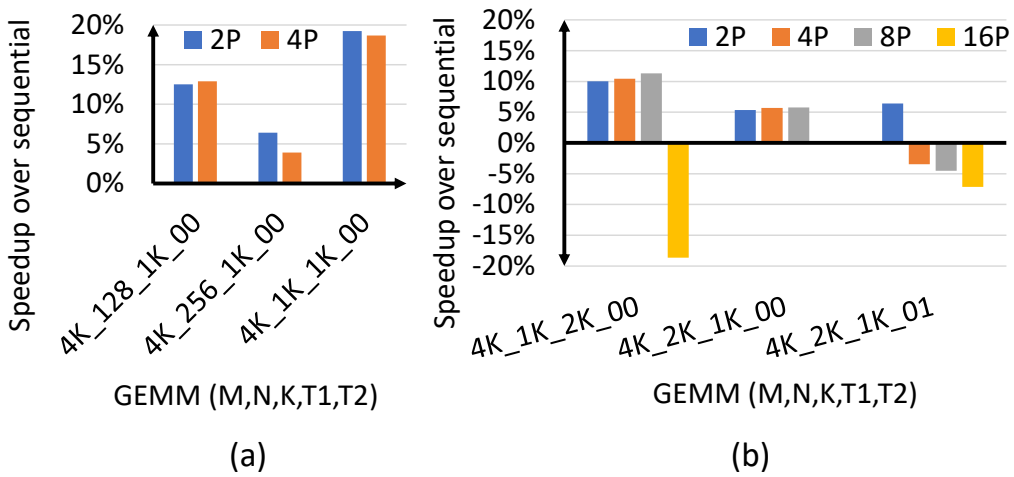


Figure 7.1: (a) GEMM sizes with fewer FLOPs benefit less from concurrency (b) GEMM sizes with the same FLOPs can have different concurrency behavior. GEMM FLOPs= 2*M*N*K.

only achieve 25% of peak throughput across vendors [102, 216]), and sequential RNN input token processing also limit matrix sizes (e.g., 2-30% utilization [96, 151, 161, 291]). Figure 2.5(a)'s weight matrix can also be small: BERT GEMMs only achieve 40-50% of peak FLOPs across GPU vendors [102, 216, 256, 289]. Larger models may use tensor parallelism to slice matrices [186], which reduces per-device memory capacity pressure but decreases their GEMM utilization. These utilization trends will also worsen with continued GPU FLOP scaling.

7.1.2 Sub-optimal GEMM concurrency in GPUs

While there are abundant opportunities to concurrently execute low utilization GEMMs as shown in Figure 2.6 and Table 2.1, they often provide small performance improvements on GPUs. Figure 7.1 illustrates this with a few examples. First, Figure 7.1(a) shows the speedup of concurrently executing two and four independent GEMMs (IG=2, 4), with the size of GEMMs (particularly the dimension N) increasing from left to right.

While the largest GEMMs achieve ≈ 19% speedup over their sequential execution, the smaller ones (with fewer FLOPs), achieve much smaller speedups.

Figure 7.1(b) shows speedups for the three sets of IGs with the same FLOPs, but different input tensor shapes (the first two) or transposes (the last two). In the first two cases speedups over sequential execution are similar or slightly increase as concurrency degree increases from 2 to 8 IGs. However, for 16 IGs performance degrades in the first case. For the last case, with a transposed tensor, performance degrades for any number of IGs beyond two. Thus, naively executing all IGs concurrently can be sub-optimal. Furthermore, GEMMs with similar compute requirements can have very different concurrency behavior.

7.2 Challenges with GEMM Concurrency

Next, we further investigate how Section 7.1.2's examples reinforce Table 7.1's two key challenges with leveraging concurrency on current GPUs.

7.2.1 Isolation-tuned kernel implementations

Figure 7.1 shows that GEMMs with the largest FLOPs benefit more from concurrency. Besides size, these GEMMs have different kernel implementations (e.g., the largest GEMM has the largest tile size, among other differences). A GEMM's kernel implementation involves tens of features that are tuned to improve its *isolated* GPU execution (Section 2.4.1.3). As a GEMM's hardware requirements differ based on its input (size, shape, transpose), they also prefer unique kernel features for maximum performance: the 410 GEMM sizes we study (Section 7.4) chose 291 unique kernel implementations.

Kernel implementations also have a significant impact on concurrent performance: a larger tile size reduces the number of WGs a GEMM

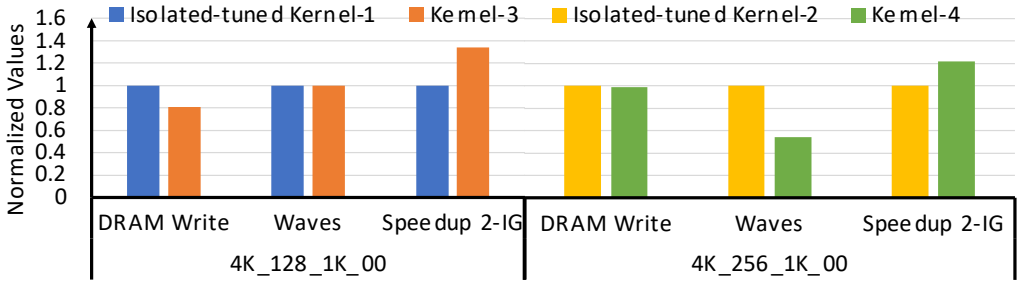


Figure 7.2: GEMM behavior with different kernel implementations. Kernels-1 and -2 are the GEMMs' isolated tuned kernels; Kernels-3 and -4 are alternate implementations with smaller memory traffic and fewer WG waves, respectively.

executes but increases the extent of data reuse at the LDS. Features such as coalescing limit global memory traffic while also increasing register/LDS requirements and decreasing per CU occupancy. The WG count and occupancy impact how concurrent GEMMs share CU resources, while data reuse and total memory traffic impact how they share the cache/memory bandwidth. Similarly, every other feature has a unique trade-off.

Figure 7.2 compares the two smaller FLOPs GEMMs from Figure 7.1 with alternate, more concurrency-amenable kernels. *Isolation-tuned* Kernel-1 and Kernel-2 are tuned for the GEMM's performance in isolation, as done by BLAS libraries. Kernel-3 improves both LDS reuse (via larger tile size) and the kernels' accesses to the LDS (via padding and prefetching) compared to Kernel-1, which reduces the (4k_128_1K_00) GEMM's global memory accesses and improves its concurrent performance with two independent GEMMs by 1.34×. Conversely, Kernel-4 slightly increases the number of WGs (smaller tile size) and reduces LDS requirements (via less coalescing) compared to Kernel-2, which improves the GEMM's CU occupancy by 2× for 4k_256_1K_00 and reduces the number of waves (set of WGs a kernel simultaneously executes on a GPU). This improves the GEMMs' overlap and increases speedup by 1.22× for two concurrent GEMMs.

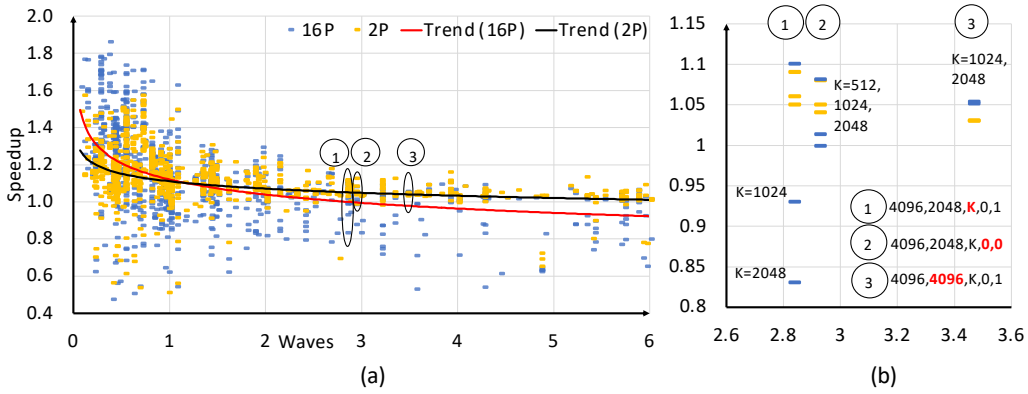


Figure 7.3: (a) Speedups over sequential execution for 2 & 16 concurrent GEMMs (2P & 16P) versus the #waves in their isolated execution. (b) Speedups of GEMMs with fixed #waves but with varying K, input shape, or transpose.

Overall, these exemplar results show that considering the *global* resource environments for kernel implementations, based on the operations executing concurrently can improve performance. However, there are two challenges in realizing them: (a) GEMMs have different (e.g., memory, compute) bottlenecks depending on the input properties and must optimize for different resources as shown in Figure 7.2 and (b) there are several features, each with a unique trade-off, that can be tweaked to optimize for the bottlenecked resource – and similar to the baseline BLAS libraries, these will differ for each GEMM input. Thus, manually identifying such alternative implementations can be challenging. Therefore, we need a method to identify *globally* optimized kernels across many GEMMs.

7.2.2 Static concurrency control

Figure 7.3(a) examines how the 410 studied GEMMs (Section 7.4) perform when two and 16 independent GEMMs are run concurrently. The x-axis shows the number of waves used by the GEMM kernels. While the general trend shows that smaller wave GEMMs see better concurrency behavior

(higher 2-IG speedups and benefits with higher concurrency degrees, 16-IG), and matches our earlier observation (Section 7.2.1) that smaller/fewer waves enable better overlap/sharing of CUs, the behavior varies quite a bit for GEMMs with similar waves. We highlight this using examples ①, ②, and ③, zoomed in on the right in Figure 7.3(b). ① shows that benefits of concurrently executing GEMMs with the same M, N, T1, and T2, as well as waves but differing K dimensions can vary considerably; performance degrades at K of 1024 and 2048. The summation dimension (K) determines the amount of work performed and data read per thread and per WG. Our profiling of isolated GEMM execution[1] shows that increasing K also increases the memory reads-to-input matrix size ratio, implying larger K GEMMs are more prone to Last-Level Cache (LLC) and memory bandwidth contention.

Similarly the transpose combination (T1,T2) determines the GEMM input tensors' layout in memory and thus its memory access pattern. Certain transpose combinations have better data locality and improve cache/bandwidth sharing during concurrency. ② in Figure 7.3(b) which has the same GEMM dimensions and similar waves as ①, but a different $(0, 0)$ transpose, does not see performance degradation as in ①. Finally, the shape of tensors also dictates behavior. Generally, similar-sized inputs ($M \sim N$) indicate that input rows and columns have similar cache reuse. Therefore, ③, which has similarly-sized inputs but larger GEMMs with more waves, also does not see ①'s performance degradation.

There are many such varied behaviors amongst the 410 GEMMs in Figure 7.3(a). Whether GEMM concurrency is beneficial is dictated by a combination of input sizes, tensor shapes, layout, and kernel implementations. These concurrency benefits cannot be determined via simple heuristics and require profiling. Although offline profiling could potentially identify the right amount of concurrency to exploit in every intra-application case,

[1]AMD and NVIDIA GPUs currently do not support performance counter monitoring with concurrent kernels.

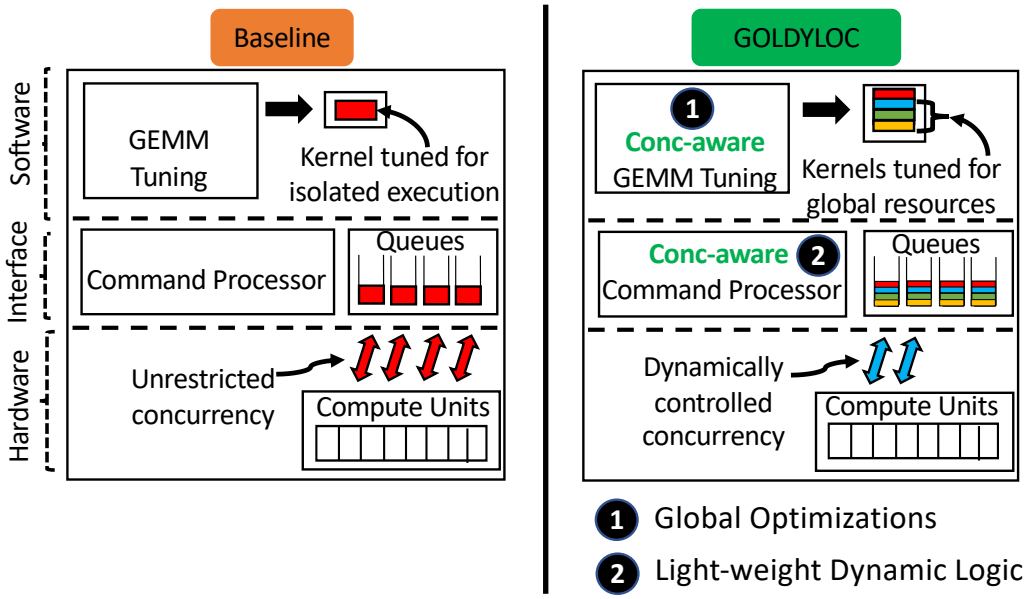


Figure 7.4: GOLDYLOC overview and baseline comparison.

since inputs and the number of parallel operations change dynamically (via multi-instance deployments) profiling for all possible combinations is onerous. Conversely, profiling at runtime can add significant overheads, and diminish concurrency benefits. Thus, GPUs need a lightweight, *dynamic* logic to manage concurrency.

7.3 GOLDYLOC Design

7.3.1 Overview

Figure 7.4 depicts the baseline system (left) and GOLDYLOC (right). We only show system components that GOLDYLOC affects. In the baseline, there is a one-time GEMM library tuning for a given GPU such that, for a given GEMM size, at runtime the library returns a kernel optimized for its *isolated* execution (Section 7.2.1). At runtime the CP, an embedded programmable microprocessor within the GPU, acts as the interface

Acronym	Definition	Acronym	Definition
CD	Concurrency Degree	CP	Command Processor
RC	Resource Constraint	CU	Compute Unit
nP	n Parallel GEMMs	KO	Kernel Object
GO	Globally Optimized	LLC	Last Level Cache

Table 7.2: GOLDYLOC Acronyms

between the software and hardware [140, 142]. It schedules as many independent GPU kernels as resources permit [205, 222], either exposed by programmers via streams/queues *statically* [157, 185] and/or from multiple processes. In Figure 7.4, the CP may schedule all four available GEMMs concurrently, each using an isolation-tuned kernel.

GOLDYLOC (Figure 7.4, right) redesigns GPU libraries and runtime to add *concurrency awareness* to the system. Similar to the baseline GOLDYLOC requires a one-time tuning of the GEMM library for a given GPU. However, GOLDYLOC enhances the tuning methodology: for a given GEMM size, at runtime the library returns both a kernel optimized for isolated execution and also kernels which are ***globally optimized*** for multiple concurrency degrees (CDs, i.e., number of concurrent GEMMs, Section 7.3.2). GOLDYLOC further programs the CP with a ***lightweight dynamic logic*** to control the amount of concurrency on the GPU (Section 7.3.3). At runtime, given a set of independent GEMMs and their globally optimized kernels, the CP predicts a performant CD and schedules that many GEMMs with appropriate kernels. In Figure 7.4, the CP dynamically predicts and schedules two out of the four GEMMs with kernels globally optimized for a CD of two. Thus, GOLDYLOC only *dynamically* executes concurrent GEMMs which can improve overall performance, with kernels optimized for a *global* shared resource environment.

7.3.2 Globally optimized (GO) GEMM kernels

Concurrently executing GEMMs with kernels tuned for isolated execution, as in the baseline, is suboptimal (Section 7.2.1) and may also hurt performance (Figure 7.3). The baseline's rigorous benchmarking minimizes a kernel's latency assuming all GPU resources are available for a single GEMM. This leads to kernels that may end up hoarding resources that must be shared during concurrent executions (e.g., isolated tuned Kernel-1 is cache/memory bandwidth-heavy, Kernel-2 is CU-heavy). Therefore, GPUs must use kernels that are globally optimized (GO) for the available (shared) resources (e.g., Kernel-3 and Kernel-4 which limit the respective GEMMs' bandwidth and CU usage, respectively). This requires identifying, for each given GEMM, which resources must be optimized for, and which kernel feature(s) to focus on to achieve that. GOLDYLOC identifies such kernel implementations by augmenting the tuning process to include **resource constraints** (RCs). Executing GEMMs with RCs emulates a concurrent environment where resources are shared, and thus limited. Thus, tuning the kernel for each GEMM in such RC environments (Section 7.3.2.1) can automatically help identify the features optimized for the bottlenecked resource.

7.3.2.1 Resource-constrained (RC) tuning

When incorporating RC into tuning, we must consider: which resources to focus on and how to augment tuning?

The most pertinent GPU resources are: compute units (CUs), cache, registers, LDS, and memory bandwidth. Although GPU configurations can be modified to limit a kernel's on-chip resources (e.g., CUs, cache, LDS) [202, 205], limiting memory bandwidth is more difficult. Sophisticated data placement (e.g., over a subset of memory channels) adds significant software complexity. Moreover, while tweaking memory frequency is possible, it may lead to lower access latency that may not be represen-

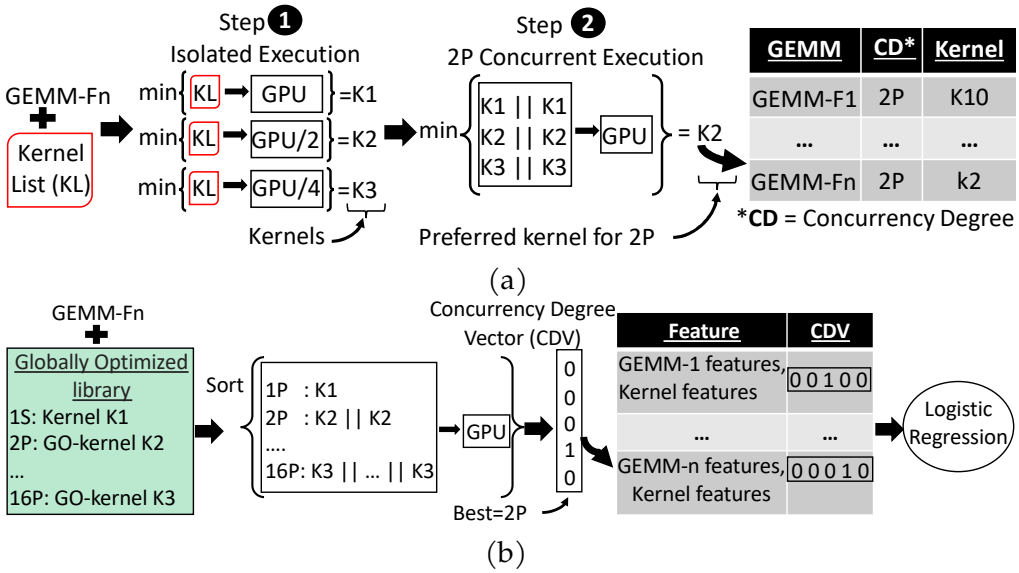


Figure 7.5: (a) GOLDYLOC's tuning methodology for a single GEMM for concurrency degree = 2P. (b) Identifying optimal concurrency degree for a single GEMM feature, and taming its overhead using a logistic regression-based model.

tative of access latency during concurrent execution. Thus, we focus on constraining CU count and LLC size. We create two RC configurations in addition to baseline GPU configuration (GPU): GPU/2 (halves #CUs and LLC size) and GPU/4 (quarters #CUs and LLC size). We selected these based on available parallelism (or concurrency degree, CD) and empirical results which show little benefit from stricter RCs (Section 7.6).

Figure 7.5a shows how GOLDYLOC tunes for a given GEMM (GEMM-Fn). The baseline tuning process rigorously benchmarks the available GPU kernels (Kernel List (KL)) on a resource-unconstrained GPU configuration. Our tuning process also examines GPU/2 and GPU/4 (Step ①). Next, using the set of most efficient kernels from Step ①, we benchmark concurrent execution for each CD of interest (e.g., 2P, 4P) (Step ②). We benchmark kernels from all three RC configurations for all CDs. For example, for CD=2P we benchmark kernels most efficient for GPU, GPU/2,

and GPU/4. The smallest runtime kernel is preferred for the given GEMM and CD (K2 in Figure 7.5a). It is possible that a kernel tuned for isolated execution (RC=GPU) is preferred for concurrency as well. This happens if the GEMM is bound by a resource during its isolated execution and already selects the appropriate kernel to use that resource, requiring no further RC-tuning. For example, very large compute-bound GEMMs will often use kernels that limit the total WG and wave count. This is also possible for small GEMMs at low CD which already have sufficient overlap and few waves. (e.g., GEMMs with 0.5 waves will not benefit further from 0.25 waves if CD=2P). To reduce the benchmarking cost in Step ②, we also propose using similarity analysis amongst GEMMs to determine the RC config for every CD (discussed in Section 7.6).

7.3.2.2 Globally optimized GEMM library

The baseline *GEMM library* has GEMM inputs and associated GPU kernels optimized for isolated execution. GOLDYLOC augments this library (Figure 7.6): during runtime for each GEMM it also returns pointers to globally optimized (GO) kernels efficient for the global resource environment per CD (①).

7.3.3 Dynamic logic for concurrency control

Baseline GPUs statically control concurrency within applications, without knowledge about dynamic input sizes or number of processes. This can degrade performance because, as observed in Section 7.2.2, not all concurrency is beneficial, even when using GO kernels. Moreover, while dynamic control is important, determining the appropriate amount of concurrency at runtime is challenging. It depends on a combination of factors (GEMMs' tensor size, shape, and layout as well as kernel implementation (Figure 7.3(b)) and requires profiling which can add significant overheads

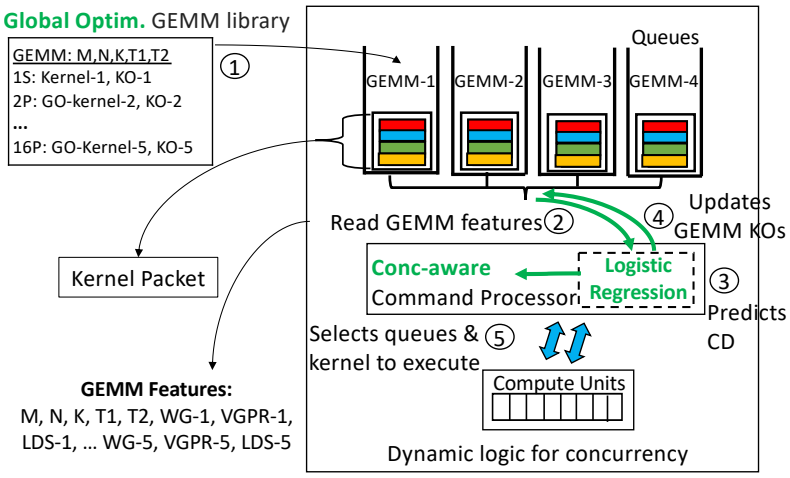


Figure 7.6: GOLDYLOC GEMM library and dynamic logic.

at runtime. To overcome this, GOLDYLOC uses one-time offline profiling of a subset of GEMMs and trains a lightweight predictor to determine the appropriate CD to execute at runtime.

Offline profiling & predictor dataset: Figure 7.5b depicts offline profiling which identifies the appropriate CD for a GEMM and creates the dataset used to train the predictor. For a given GEMM GOLDYLOC benchmarks the kernels identified by the GO GEMM library with their associated CD (e.g., 2P uses GO K2). Amongst all possible CDs, it associates this GEMM with the CD that delivers the most speedup over its corresponding serial execution. Increasing the number of concurrent GEMMs up to this CD improves performance but further increases either provide no further improvement or degrade performance. Thus, the final executed CD should be the minimum of this preferred CD and the available GEMMs.

Based on our observations in Section 7.2.2 GOLDYLOC uses GEMM dimensions and its per-CD kernels' (#WGs, occupancy, and #waves) as the predictor's input features as they capture all input, implementation, and underlying GPU's hardware properties. #WGs is a function of output size (M×N) and determines total parallelism within the GEMM. Occupancy

$C \rightarrow \#total\ possible\ CDs$
$D \rightarrow \#available\ GEMMs$
$X \rightarrow input\ vector\ of\ size\ N$ // M, N, K, per-CD features (WGs, occupancy, waves)
$W \rightarrow Weight\ matrix\ of\ size\ N \times C$
$P = \frac{e^{X \times W}}{\sum_{i=0}^{C} e^{X \times W_i}}$ // trained multi-class regression predictor with weight W
// P is the vector of probabilities for all possible CDs
$M = rowwise\ argmax(P)$ // CD with max probability
$CD = \min(M, D)$ // actual executed CD

Figure 7.7: GOLDYLOC's dynamic logic.

accounts for each WG's resource requirements, hardware resources per CU, and potential L1 cache contention. Wave count incorporates total CU count in hardware, kernel tile size, and potential for overlap. Finally, size (specifically, K) and shape (M, N) provide information on memory contention. We also considered other individual kernel features (e.g., grid size, LDS/register size) and performance data, but found minimal accuracy improvements.

Logistic regression model details: To compare different CD's relative benefits GOLDYLOC trains a multi-class (one class per CD) logistic regression model [46, 97, 262]. Logistic regression is appropriate as GEMMs have multiple input features with near-linear relationships with concurrency benefits (e.g., speedup drops with increasing K) and because it generates a multi-class output (either no concurrency or CD of 2, 4, 8, 16). The predictor calculates the probability of preferring one CD over the rest (one-vs-rest, OvR) and predicts the appropriate CD, including no concurrency. Training it fits (learns the weights of) Equation 7.1:

$$P = \frac{e^{X \times W}}{\sum_{i}^{C} e^{X \times W_i}} \tag{7.1}$$

where P is the probability vector to select one CD over the rest, X $(x_1, x_2, .., x_n)$ are input features, W is the weight matrix and C is the possible CD count.

The predictor is trained on the dataset created via offline profiling. In

the training dataset all GEMMs' features are mapped to their preferred CD (Figure 7.5b's table). To create a more exhaustive dataset GOLDYLOC includes additional GEMMs beyond the evaluated applications, for a total of 1072 GEMMs. We apply min-max normalization to normalize the dataset feature values. GOLDYLOC trains the model offline once per GPU (accuracy discussed in Section 7.5.6) using 90% and 10% samples for training and testing, respectively. After training it predicts the appropriate CD (1S, 2P, 4P, 8P, or 16P, Figure 7.7). Given the queued GEMMs' feature vector, X, and learned weights, *W*, it calculates the probability to choose each possible CDs (total C) with Equation 7.1 and selects the one with maximum probability. The final chosen CD is the minimum of the predicted CD and available GEMMs. Figure 7.6 shows how GOLDYLOC incorporates this predictor into the GPU CP (discussed further in Section 7.3.4).

7.3.4 Integrating GOLDYLOC into GPU's CP

Kernel-packet Extensions: To schedule a GEMM on a GPU, CPUs enqueue a *kernel packet* [24] in the CP's queues on that GPU. This packet includes a pointer to the kernel object (KO) that is invoked to execute the GEMM, along with its associated metadata such as the kernel's input arguments and features (e.g., WG size). The packet also includes additional header, setup, and reserved bytes. Since identifying the appropriate GO kernel, and thus the appropriate KO, for a given GEMM requires dynamic information about available parallelism and input sizes, a kernel packet cannot be pre-mapped to a single KO. Instead, GOLDYLOC extends kernel packets to include a map of KO pointers and metadata for each GO kernel (max three per GEMM from the three RC configurations) from the GO library (Section 7.3.2.2). These extensions add a little overhead, but since KOs are relatively small and only in CP memory until dispatch completes, the packets still fit in the CP's memory.

Command Processor Extensions: At runtime, existing GPU CPs inspect

all available software queues (streams) and their kernels to schedule as many independent kernels from separate queues as resources permit [205, 222]. Thus, the CP is well suited to dynamically control the amount of concurrency. GOLDYLOC programs the CP to inspect the kernel packets at the head of all active queues (② in Figure 7.6) for available independent GEMMs that could execute concurrently. This includes (a) checking if the kernels are GEMMs or non-GEMMs, (b) if there are multiple GEMMs, reading the necessary features from queued packets, and (c) calculating the remaining features (occupancy and waves) needed for prediction. The CP performs these operations each time a queue's head changes – when a kernel finishes dispatching its WGs or when new work is enqueued. CP functionality is unchanged if it detects a single GEMM and/or non-GEMMs. For multiple GEMMs, given the number and features of the GEMMs, the CP predicts the appropriate CD (③ in Figure 7.6). both the right (set of) GEMM(s) and how many GEMMs to execute concurrently. Finally, the CP updates the packet contents of each GEMM in the queue heads to point to the KO corresponding to the GO kernel for that CD (④ in Figure 7.6) which are then executed on the GPU ((⑤ in Figure 7.6)).

7.4 Methodology

7.4.1 System Setup

We evaluate GOLDYLOC with AMD's ROCm™ because it has an open-source BLAS tuning framework that performed similarly to other BLAS libraries. We use an AMD Ryzen™ Threadripper™ CPU [13] and an AMD Instinct™ MI100 GPU [21][2] with 32GB of HBM2 [110]. We calibrated our

[2]Given the fast-evolving GPU space with improved ML-specific optimizations built into each generation, we use a setup with the latest available GPU for ML at the time. Thus, the GPU used in this chapter is same as that in Chapter 4 but differs from those in Chapters 3, 5 & 8.

system's baseline performance and found it was similar to other commercial systems and prior work [102]: they had similar FLOPS relative to the peaks, 90% of all studied GEMMs had differences within -12% to +10%.

Our software extends AMD's ROCm™ 4.1 [18] by using Tensile [23] for tuning and rocBLAS [14] to build the custom BLAS libraries. Both the tuner and the library utilize Matrix Core Engines [20].

7.4.2 Applications and GEMMs Studied

To evaluate GOLDYLOC we use 410 GEMMs (Table 7.3) from forward and backward passes of state-of-the-art RNNs and Transformers while varying their batch and token sizes ("Input Params" in Table 7.3). We evaluate independent GEMMs both within and across networks for multi-instance inference deployments (Section 2.4.1.5): 2, 4, 8, and 16 instances (there were diminishing returns beyond 16). To create a more representative dataset we include additional GEMMs (1072 total). The GEMM's ranges are: 32K-168M for output size (M*N, dictates parallelism), and 64-20K for K dimension (dictates data per thread/WG). They represent a wide variety of memory and compute-bound behavior; ops/byte (dictates memory-boundedness) ranges from 28-1400. We also study concurrent strided batched-GEMM (B-GEMMs) from Transformer Attention layers (with over 40 combination of different SLs). Finally, we examine full and half precision GEMMs.

7.4.3 Measurement

For GO kernel tuning and profiling for dynamic predictor datasets (Section 7.3), we execute GEMMs with different resource constraints (RCs), concurrency degrees (CDs, via GPU streams), and kernels. We execute GEMMs back-to-back on the same stream multiple times to average

Network	Hyperparameters	Input Params
GNMT [280]	H=512;1024	B=64;128;256;512
DeepSpeech2 (DS2) [29]	H=800	B=64;128;256
RNN-T [90]	H=2048	B=64;128;256;512
Transformer [273]	H=512;1024	Tokens=512;1024;2048;4096; 3072;8192
BERT [62]	H=768;1024	Tokens=2048;3072;4096;8192
GPT-2 [226]	H=1280;1600	Tokens=2048;3072;4096;8192
GPT-3 [41]	H=4096;5140	Tokens=2048;3072;4096;8192
Megatron-LM_BERT [256]	H=1024;2048;2560	Tokens=2048;3072;4096;8192
Megatron-LM_GPT [256]	H=1920;3072	Tokens=2048;3072;4096;8192
Turing-NLG [167]	H=4256	Tokens=2048;3072;4096;8192

Table 7.3: Benchmarks with hyperparameters and inputs.

out queuing delays in concurrent setups. We measure runtimes using rocProf [12].

7.4.4 GOLDYLOC Performance Measurement

7.4.4.1 Globally Optimized (GO)-Kernels

We modify Tensile [23] tuning infrastructure to create a custom globally optimized library (Section 7.3.2). Sequential GEMM applications are linked to the baseline library. We create two binaries of the concurrent GEMM application, each linked to the baseline or GO library. To evaluate GO-Kernels, for each GEMM size, we find the speedup of both the concurrent binaries (with different CDs) over the sequential run of the GEMM.

7.4.4.2 GOLDYLOC

Although the dynamic control logic can be implemented in existing GPUs by reprogramming the CP, GPU vendors have not disclosed an API [140, 142, 234, 285]. We also implemented our changes in gem5's CP [42, 83, 156, 243] but its performance trends did not match real hardware [106,

232, 233]. Thus, we evaluate GOLDYLOC by measuring the runtime of concurrent GEMMs with CD predicted by the dynamic logic (using the custom GO library) on real hardware and add our CP change overheads.

We model the CP's dynamic detection, prediction, and selection (Section 7.3.4) latency. This includes the CP's kernel packet reads and writes from queues and logistic regression model execution. We model the CP's latency assuming the CP runs at 1.5 GHz [181] and the CP's memory access latency is 31 cycles [131]. Given 32 software streams maximum, the CP takes ≈0.32 µs to read or write the necessary queues. Finally, we estimate the predictor overhead by executing it on a CPU with similar specifications to the CP. Collectively, the total time for the CP to inspect, predict, and write queues is 8 µs (implications discussed in Section 7.5.5). Overall, this setup closely mimics executions on real GPUs, since we add GOLDYLOC's overheads to runtimes from a real GPU for each given GEMM.

7.4.5 Configurations

Since our experiments use a real GPU (Section 7.4.1), we can only perform apples-to-apples quantitative comparisons against other strategies that run on real GPUs (we qualitatively compare against other schemes in Section 7.7). We evaluate the following configurations:

- **Sequential** uses the isolated tuning and executes all GEMMs sequentially.
- **Default** uses isolated tuning and baseline GPU to execute all available GEMM (via streams) concurrently given GPU resource.
- **Globally optimized-Kernels** (**GO-Kernels**) uses global resource-aware tuning and baseline GPU.

- **GOLDYLOC** uses **GO-Kernels** and **dynamic logic** at CP to predict the appropriate CD.
- **Oracle** uses GO-Kernels and always chooses the right CD, including sequential execution if no CD provides $\geqslant 5\%$ benefit.

Additionally, we also evaluate GOLDYLOC on GPU configurations with explicit resource partitions:

- **CU-Partition** uses CU masking [204] to statically partition CUs across streams.
- **Resource-Partition** statically partitions CUs, LLC, and memory bandwidth across streams [22, 202]. Since our GPU only supports partitioning CUs, we simulate nP concurrent GEMMs for **Resource-Partition** by executing a single GEMM with $1/n$ CUs, $1/n$ LLC (by reducing cache size), and $1/n$ memory bandwidth (by varying memory clock frequency (MCLK)). This model is optimistic, since partitions' combined usually have have fewer resources than the overall GPU [202]. Furthermore, since our setup can only halve MCLK, we only include 2P results for *Resource-Partition* and provide optimistic projections for higher CDs (Section 7.5.9).

We also evaluated Rammer [161] and ElasticKernels [206]. However, ElasticKernels does not support kernels that use LDS, which all of our GEMMs do, and our baseline outperformed Rammer by 88%, which only supports the older ROCm 3.5. Thus, we do not show results for Rammer.

7.5 Results

Figures 7.8, 7.9, 7.10, and 7.11 show GOLDYLOC's benefits over sequential execution for with 2, 4, 8, and 16 independent GEMMs, respectively. These are run on the default GPU configuration (non-resource partitioned).

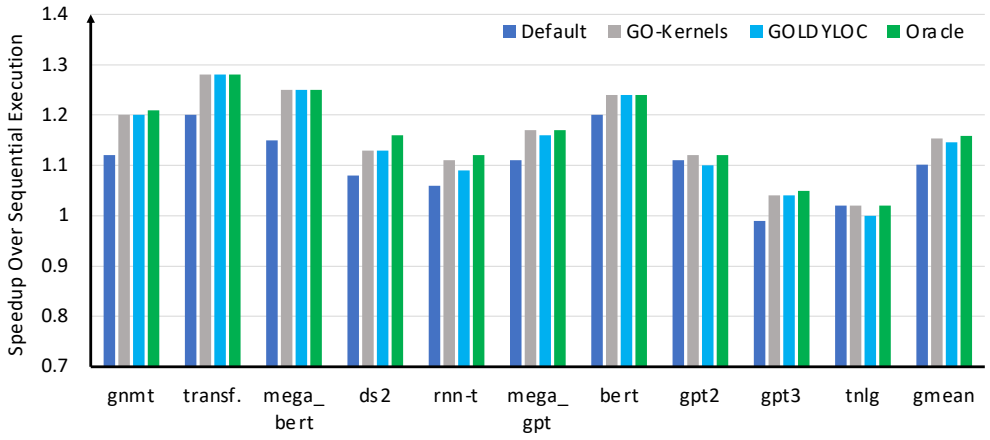


Figure 7.8: Per-app GEMMs geomean speedups with 2 independent GEMMs

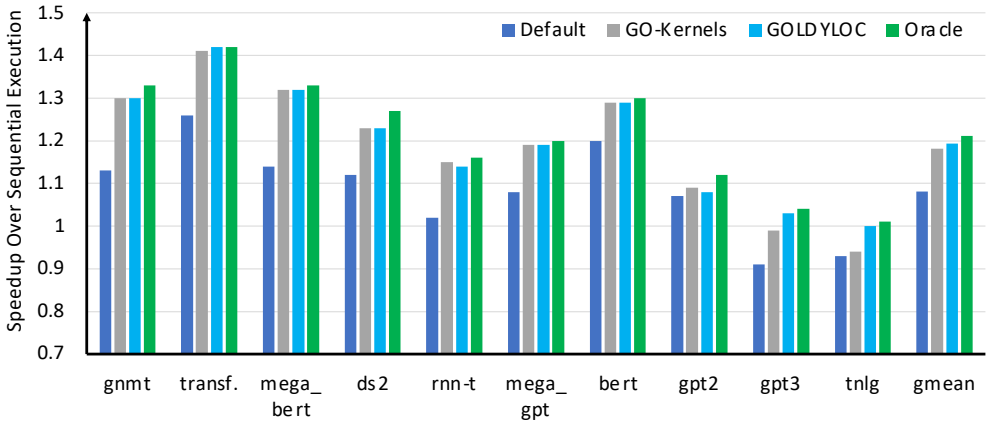


Figure 7.9: Per-app GEMMs geomean speedups with 4 independent GEMMs

Overall, GOLDYLOC's geomean benefits increase with more independent GEMMs. However, the speedups vary considerably for GEMMs across applications.

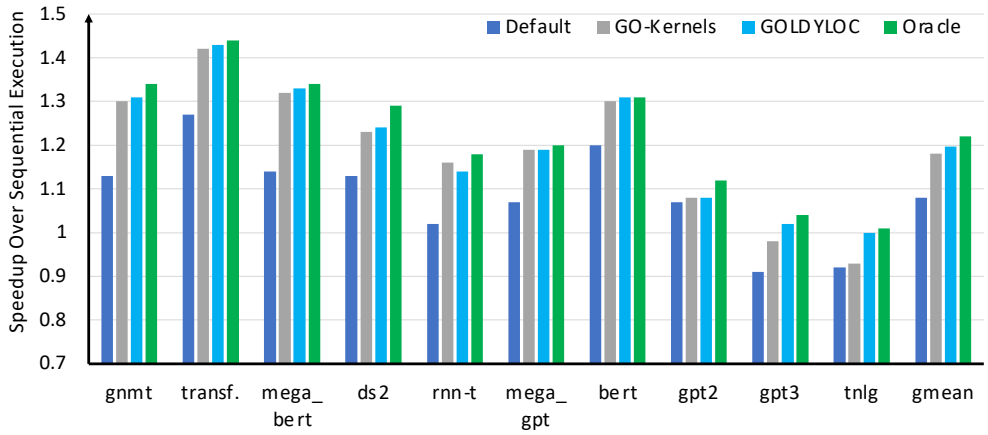


Figure 7.10: Per-app GEMMs geomean speedups with 8 independent GEMMs

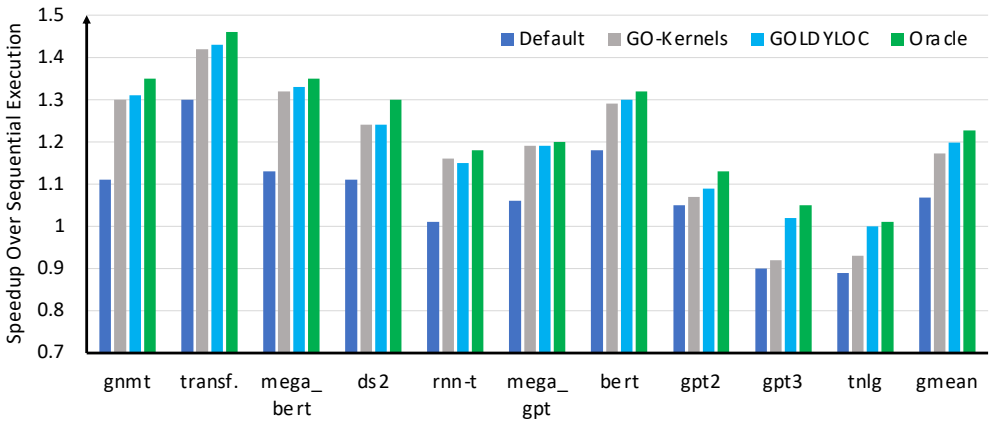


Figure 7.11: Per-app GEMMs geomean speedups with 16 independent GEMMs

7.5.1 Exploiting Concurrency (default)

With two independent GEMMs, *default* provides 10% geomean speedup over executing them sequentially. However, for almost all GEMMs, further increase in independent GEMMs do not always improve throughput and cause severe slowdowns for GEMMs from large hyperparameter applications (e.g., gpt2, tnlg). Thus, naively executing all available GEMMs concurrently without tuning for concurrency leads to low speedups. More-

over, *default*'s geomean speedup across all GEMMs drops (10% to 7%) as concurrency increases to 16 independent GEMMs.

Result-7.1: *Naively executing all available GEMMs concurrently without tuning for concurrency leads to small speedups. Moreover, the benefits drop further as concurrency increases.*

7.5.2 Globally Optimized (GO)-Kernels

GO-Kernels which are optimized for global resources considerably improve performance (Figures 7.8 to 7.11). With improved benefits, they enable higher CDs that *default* cannot.

GO-Kernel Properties: Each GEMM, given its input properties, makes unique trade-offs under resource constraints to pick a uniquely different kernel than its isolated counterpart. However, there are two key trends: fewer/partial waves and reduced global memory requests. In many cases, GO-Kernels have a larger tile size than their isolated counterpart. Larger tiles improve LDS reuse, reducing LLC/memory requests and thus contention. While larger tile size also decreases the total #WGs, it can increase per-WG resource requirements (e.g., LDS). Thus, GO-Kernels also change other kernel features to balance performance and per-WG requirements and limit the drop in per-CU occupancy. This combination reduces #waves and improves overlap. GO-kernels can also have a relatively smaller tile size, but also a higher occupancy which also reduces the kernel's #waves.

Figure 7.12 plots the ratio of #waves and per-wave LLC accesses/misses in GO-Kernels vs. isolated kernels. The ratios are largely < 1, indicating that GO-Kernels have fewer waves and LLC accesses/misses than their isolation-tuned counterparts, making them better for globally sharing resources (Section 7.2.1). Occasionally (right side of graph), #waves decrease and LLC activity significantly increases but the latter's absolute values are very small. Thus, the resource-constrained tuning employed by GOLDYLOC properly emulates concurrent execution environments.

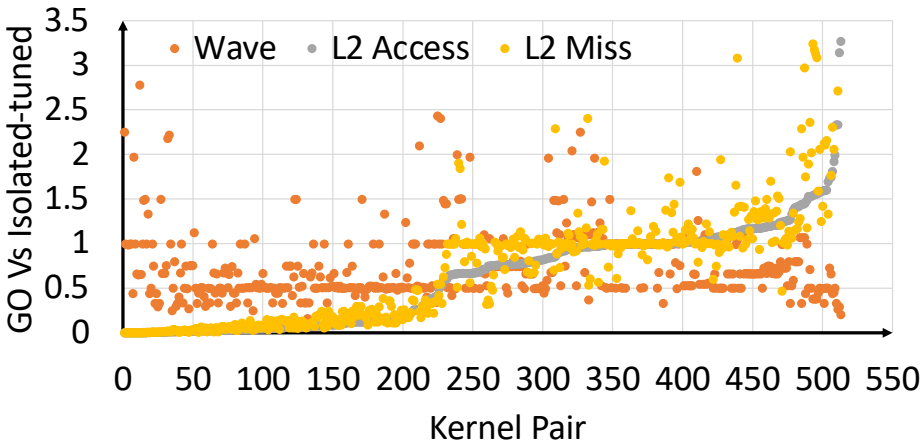


Figure 7.12: Globally optimized (GO)-Kernel properties.

Result-7.2*: Global resource-aware, GO-Kernels uniquely differ from their isolated counterparts.*

Result-7.3*: GO-Kernels have fewer resource requirements, execute in fewer #waves, and have lower global memory traffic compared to their isolated counterparts.*

GO-Kernels Benefits: In CD=2P, GO-Kernels have a maximum speedup of 52% over *default* and achieve more than 20% and 10% speedup for 11%, and 24% of the 410 GEMM sizes, respectively. GEMM sizes that did not benefit from GO-kernels with 2P do benefit at higher CDs; 53% of GEMMs in 16P (vs. 34% in 2P) benefit from GO-kernels. Furthermore, the benefits of GO-kernels increase at 16P: 2× maximum speedup, 25% of all GEMMs obtain $>$ 20% speedup, and 43% of all GEMMs obtain $>$ 10% speedup, all over *default*.

Since not all GEMMs benefit from GO-Kernels (Section 7.3.2.2), per-application benefits depend on how many of an application's GEMMs use GO-Kernels and the extent of their benefits (Figures 7.8 to 7.11). Most gnmt, transformer, and mega-bert GEMMs prefer GO-Kernels and achieve higher speedups over *default* and *sequential* execution: 7-9% and 20-28% geomean speedup in 2P and 11-20% and 30-42% in 16P. For applications

with few GEMMs that prefer RC-tuning, benefits over *default* are only up to 5% for CD=2P but 9-17% geomean for CD=16P. Finally, large-dimension GEMMs from large networks (e.g., gpt2, tnlg) are often compute-bound and do not benefit from GO-Kernels. Across all GEMMs in Figures 7.8 to 7.11, GO-Kernels achieve 5% and 10% geomean speedups over *default* for CDs of 2P and 16P, respectively. For 4P and 8P CDs, GO-Kernels achieve up to 1.7× and 2× speedups, respectively, with 9% geomean speedups. Overall, *GO-Kernels*'s benefits are large for small- and medium-sized workloads and increase at higher CDs. Thus choosing globally optimized kernels is important.

Result-7.4*: GO-Kernels's benefits are high for small- and medium-sized workloads. Its benefits increase at higher CDs.*

7.5.3 GOLDYLOC

Two GEMMs with GO-Kernels often execute concurrently without heavy contention. Thus, *GOLDYLOC*, which dynamically controls concurrency (Section 7.3.3) often provides no additional benefits for two independent GEMMs. However, its benefits increase as available independent GEMMs increase. Large compute-bound GEMMs in gpt2, gpt3, and tnlg suffer at CDs > 2 because their large per-WG data increase LLC thrashing for more than two concurrent GEMMs. *GOLDYLOC* accurately predicts this, improving overall performance by 10% over *GO-Kernels*. Moreover, GOLDYLOC mispredictions only hurt 7% of GEMMs (Section 7.5.6). Overall, GOLDYLOC improves performance by up to 35% (3% geomean) over *GO-Kernels* and by 5%, 10%, 11%, and 12% geomean for 2P, 4P, 8P and 16P, respectively, over *default*.

Result-7.5*: GOLDYLOC predicts performant CDs and improves geomean GEMM performance by up to 12% over default.*

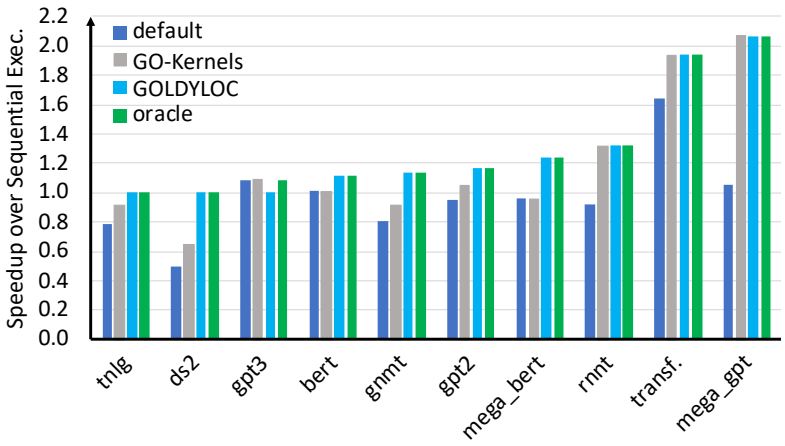


Figure 7.13: Select GEMMs, GOLDYLOC (CD=16).

7.5.4 Range and Distribution of Benefits

To demonstrate the range of *GOLDYLOC*'s benefits, Figure 7.13 plots their speedups for 16 independent GEMMs for few GEMM sizes. In the best cases (rnnt, transf., mega_gpt GEMMs), *GO-Kernels* improves performance (up to 2×). In others (tnlg, ds2, bert, gnmt, gpt2, mega_bert), *GO-Kernels* provides little benefit, but *GOLDYLOC* selects a more performant CD. In the worst case (gpt3), *GO-Kernels* do not help, and *GOLDYLOC* mispredicts, hurting performance. Compared to *default*, across 410 GEMMs *GOLDYLOC* improves 64% of cases, has no impact on 29%, and degrades performance only in 7% of cases.

7.5.5 CP Overheads

To avoid increasing the critical path, CP attempts to perform the prediction, packet setup, and queue prioritization (Section 7.4) in parallel with prior executing kernels. Thus, the 8 µs overhead is incurred only for the initial kernel and if prior kernels are short ($< 8\mu s$). We study kernel runtime distributions (including non-GEMMs) of several DNNs and all but two kernels have runtimes greater than 8 µs as shown in Figure 7.14. Thus, the

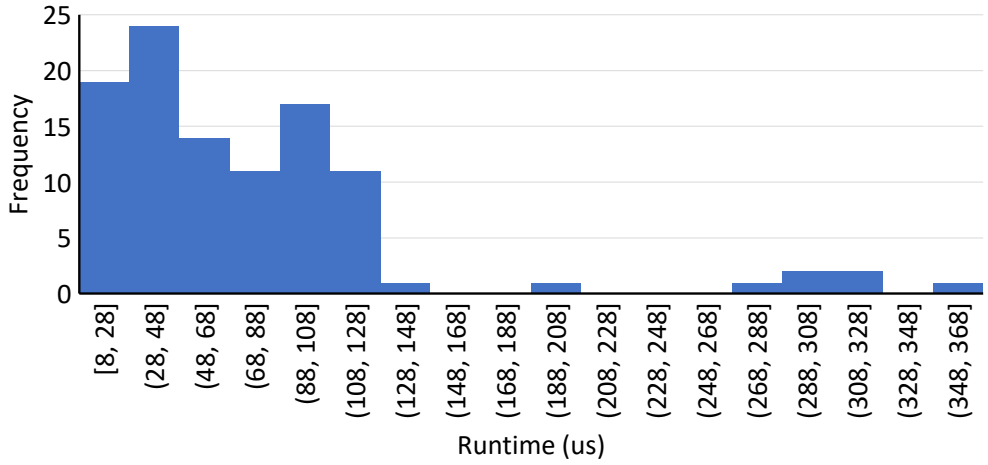


Figure 7.14: Distribution of kernel runtimes (2 samples with < 8 µs are excluded).

latency can be hidden without impacting end-to-end time.
***Result-7.6**: GOLDYLOC's overheads are small and can be hidden.*

7.5.6 Logistic Regression Model Accuracy

The *accuracy* of the Logistic regression-based model for 2, 4, 8, and 16 available GEMM scenarios is 82%, 70%, 62% and 47%, respectively. *GOLDYLOC*'s accuracy decreases with higher number of available GEMMs, which have more output classes. However, when *GOLDYLOC* is incorrect, often multiple CDs provide similar (better than *default*) performance. Thus, it still selects a high-performance CD and provides most of *Oracle*'s benefits. Training with a more exhaustive set of GEMM sizes could further improve accuracy and reduce the gap between GOLDYLOC and *Oracle*.

7.5.7 Heterogeneous GEMMs & Batched-GEMMs

GOLDYLOC also improves the performance of heterogeneous concurrent GEMMs, where concurrent GEMMs have unique input sizes. For brevity, we only consider two unique GEMMs. *GO-Kernels*, which are

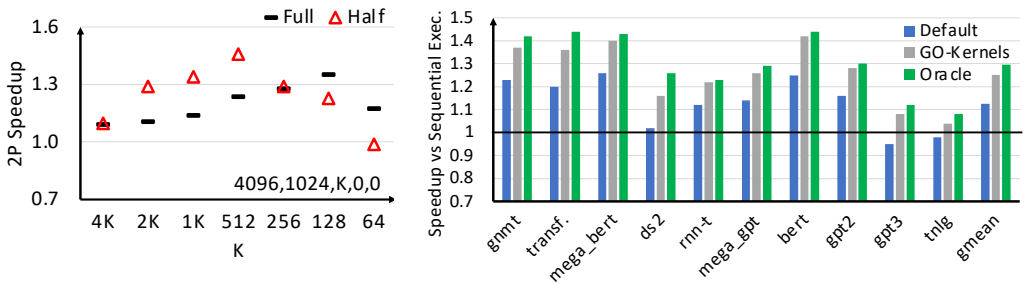


Figure 7.15: FP16 (a) vs. FP32 2P concurrency with varying GEMM sizes (b) 16P benefits with GOLDYLOC-Kernels.

heterogeneity-agnostic provide 3-10% geomean speedup over *default* for CD=2-16P. Further, *GOLDYLOC*'s prediction logic is extrapolated for heterogeneity and provides up to 5% additional speedup for CD=16. For 16 independent GEMMs, the CP executes all concurrently only if both unique GEMMs prefer 16P. If not, CP makes decisions assuming two sets of 8 independent homogeneous GEMMs. Overall this provides 15% geomean speedup over default in 16P.

GOLDYLOC also helps with heterogeneous concurrent batched-GEMMs (B-GEMMs) [183]. B-GEMMs execute many small, independent, and same-sized GEMMs in one kernel [2, 111]. Transformers execute independent B-GEMMs to process variable-length inputs. Applying *GO-Kernels* to 2P heterogeneous B-GEMMs provides up to 1.94× and 1.5× speedups, and geomeans speedups of 5% and 8%, respectively.

Result-7.7: *GOLDYLOC accelerates heterogeneous concurrent GEMMs by 15% geomean over default in 16P.*

Result-7.8: *GOLDYLOC accelerates heterogeneous concurrent strided batched-GEMMs by 8% geomean over default in 4P.*

7.5.8 Reduced Precision

Figure 7.15 examines FP16 precisions [20, 73, 165, 190, 203, 223, 277] impact on GOLDYLOC. Since FP16 throughput on the same device is higher than

FP32's, its peak concurrency speedup also increases (Figure 7.15(a)). The curve also shifts left with FP16, implying more potential benefits with larger sizes. While concurrency benefits with larger (e.g., tnlg) GEMMs could be higher in FP16 than FP32, it is not observed due to concurrency-unawareness. Thus, *CG-Tuning* speeds up 16P GEMMs with gpt2, gpt3, and tnlg sizes by 10%, 14%, and 6% geomean, respectively (Figure 7.15(b)). Similarly, GOLDYLOC's FP16 benefits will also increase at higher CDs. Finally, although *CG-SP* would be retrained to consider precision, it will still provide benefits by controlling the CD.
Result-7.9: *GOLDYLOC benefits increase for large GEMMs at FP16.*

7.5.9 GOLDYLOC with Resource Partitioning

Figure 7.16 evaluates resource partitioned configurations with *default*, and *GO-Kernels*'s impact on them for CD=2P. *CU-Partition* has little or no benefit over sequential execution and is often worse than *default* due to shared memory resource contention and leaving resources of a partial wave (tail, Section 7.2) within a partition underutilized. Conversely, the optimistic *Resource-Partition*'s dedicated memory resources help it outperform *default* (similar to prior work [61, 269]). Nevertheless, partitioning resources defines constraints, making RC-tuning important for improved performance. For CD=2P simply reusing *default* tuned concurrency-aware kernels (Section 7.5.2) provides up to 1.4× and 1.6× (3% and 4% geomean) benefits over *CU-Partition* and *Resource-Partition*, respectively. Further tuning GOLDYLOC for these configurations increases them to 6% and 9% geomean (Figure 7.16). Benefits further increase for higher CDs: 5-22% for 4-8P over *CU-Partition* and 7% over an optimistic 4P *Resource-Partition*.

Result-7.10: *While partitioning resources improves performance over default, concurrency-aware tuning further accelerates it.*

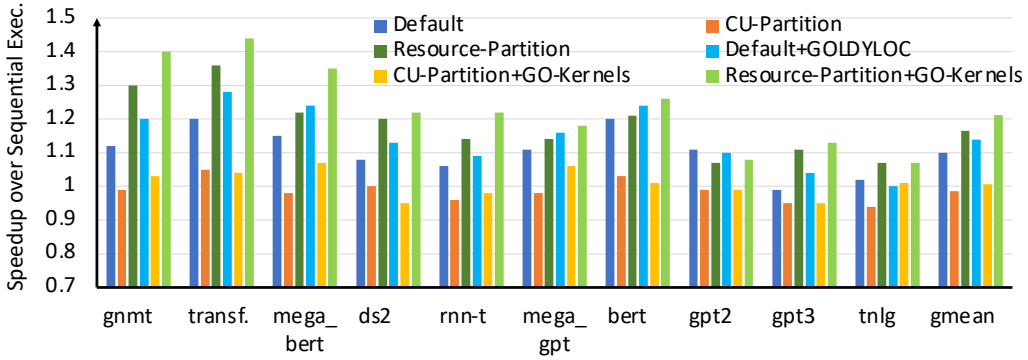


Figure 7.16: *GOLDYLOC* (CD=2P) with *default* & CU/Resource partition.

7.5.10 End-to-end Speedups

RNNs and Transformers have significant intra-network parallelism. RNN-based GNMT model (H=1024) can execute up to eight (layer) GEMMs in parallel. Thus, *GOLDYLOC* speeds up its iterations by ~14% and ~13% (for batch size 128 and 256, respectively) over *default*. GOLDYLOC also speeds up parallel Attention B-GEMMs and gradient GEMMs in Transformers: GOLDYLOC speeds up BERT's iteration times by 5-12% over *default*.

7.5.11 GEMM Fusion

While GEMM fusion [71, 148, 187] improves throughput by concatenating independent GEMMs' matrices into larger ones, it is applicable only if GEMMs share an input matrix or if the application sums all of the GEMM outputs. Its benefits also saturate as matrix sizes grow. In RNNs, the extent of fusion also determines the amount of available parallelism amongst other operations. For example, for the GNMT model evaluated in Section 7.5.10, although fully fusing all possible GEMMs improves performance by 19% over sequential execution, it causes other, smaller GEMMs (Section 2.4.1.5) to be serialized, while providing little benefit beyond fusing eight GEMMs. Leveraging concurrency with *GOLDYLOC* instead outperforms it by 10% (14% over *default*).

Result-7.11: *GOLDYLOC speeds up GNMT by 14% over default, and by 10% over maximum GEMM fusion.*

7.6 Discussion

7.6.1 Reducing Tuning Overhead

GO-Kernels's one-time cost can be reduced by predicting a GEMM's preferred RC configurations (PRCs) for a given CD. We exploit K-Nearest Neighbor (KNN)-based classification to predict a new GEMM's PRC based on the PRC of K closest GEMMs by Euclidean distance. We exhaustively tune for 20% of GEMMs (Section 7.3.2 and predict the PRC for the remaining 80%, using size ($M * N$) and *default* kernels' tile size to determine closeness. Along with dynamic control, it still improves performance over *default* by 2-9% overall (for CD=2-16P).

7.6.2 Non-GEMM Kernels

DNNs also have interspersed non-GEMM kernels, including element-wise adds, multiplies, reductions, and activation functions. Most of these kernels are bottlenecked by memory accesses. To overcome this, software frequently uses optimizations such as kernel fusion to fuse series of such operations into a single kernel, and often with preceding GEMMs, to avoid redundant global memory accesses. This significantly reduce the runtime of non-GEMM kernels. Thus, as mentioned in Section 7.4, we focus on GEMMs because they constitute the majority of runtime in DNNs. Furthermore, unlike non-GEMM kernels, libraries are rigorously tuned for different GEMM input sizes for performance, leaving much room for improvement in case of concurrent execution.

7.6.3 Additional Resource Constraints

For tuning, we evaluate only two RC configurations (GPU/2 and GPU/4 in Section 7.3.2.1). Stricter RCs (GPU/8 and GPU/16) provided little benefit, likely because kernels become prohibitively slow at such low resources to provide any concurrency benefits. However, at the considerable rate with which GPU compute scales, stricter RCs may become necessary. We also tried constraining memory bandwidth (BW) using memory clock frequency (MCLK) as a proxy (constraining BW via specific memory allocations was beyond the scope of this work) but found limited additional benefits. This may be because constraining MCLK also impacts memory latency which may not be representative of concurrent execution environments. Constraining additional shared resources may provide more concurrency-amenable kernels.

7.6.4 Sparsity

Prior work has shown that significant sparsity exists in many of these networks [56, 85, 177, 209, 294]. Leveraging sparsity is especially useful for very large networks with large parameter matrices. Although evaluating the additional behavior when exploiting sparsity is beyond the scope of this work, we expect concurrency will become more important as sparsity reduces the amount of computation in GEMMs.

7.6.5 Other DNNs

GOLDYLOC can also help CNNs, Multilayer Perceptron (MLP) layers in recommendation models [82], and Graph Neural Network's [265]. Their inter-GEMM parallelism arises from gradient descent, checkpointing, and multi-instance runs (Section 2.4.1.5). For example, GOLDYLOC speeds up MLPerf's ResNet-50 and DLRM independent GEMMs by up to 21% and 36%, respectively.

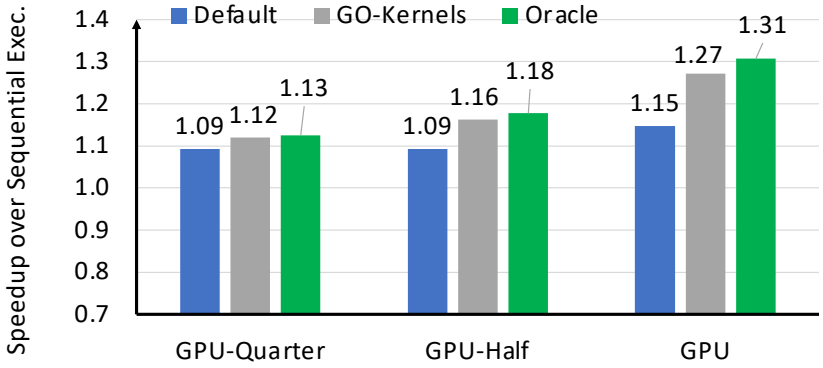


Figure 7.17: CD=4P speedups for multiple GPU configurations.

7.6.6 Scaling GPUs Configuration

Due to constantly scaling GPUs, we study GOLDYLOC's benefits by changing hardware resources. As shown in Figure 7.17, we use *GPU-Quarter* (32 CUs, 2 MB LLC), *GPU-Half* (64 CUs, 4 MB LLC), and the original *GPU* (120 CUs, 8 MB LLC) and find that GOLDYLOC's benefits are higher (benefits increase from 3% in *GPU-Quarter* to 12% in *GPU*) as GPUs scale up. Scaling GPU computing with memory bandwidth fixed increases contention, making GOLDYLOC more effective.

7.6.7 Power-aware tuning

Both GOLDYLOC and the baseline tuning consider performance to be the key determinant in choosing both the right kernel implementation as well as concurrency degree. However, other factors such as power may be the limiting factor [170]. If power-aware tuning is required, both Baseline and GOLDYLOC can be modified to additionally measure the peak power during the kernel executions. Kernels choices can then be narrowed down (e.g., in step 1 of Figure 7.5a) based on the desired maximum power, and the appropriate implementation can be selected given these constraints. Similarly, the concurrency control logic can also be trained to

Approach / Features	GPU Support	Globally Optimized	Dynamic Control	No App. Changes
Herald [136]	X	X	✓	✓
Magma [119]	X	X	✓	✓
Veltair [155]	X	✓	✓	✓
Queue Schedulers	✓	X	✓	✓
Wavefront Schedulers	✓	X	X	✓
Rammer [161]	✓	Partial	X	Partial
Elastic Kernels [206]	✓	Partial	X	✓
Batched-GEMM [183]	✓	Partial	X	X
GOLDYLOC	✓	✓	✓	✓

Table 7.4: Comparing GOLDYLOC to prior work.

avoid concurrency degrees which exceed the desired power threshold.

7.7 Related Work

Table 7.4 compares GOLDYLOC to other state-of-the-art schemes that use a range of solutions to exploit parallelism. We compare GOLDYLOC with these approaches based on four key features detailed below.

Other Devices: Similar to GOLDYLOC, Veltair [155] optimizes for multi-tenancy on CPUs, while MAGMA [119] and HERALD [136] focus on accelerators. Although a similar goal, their optimizations differ since they either target latency-oriented CPUs [155] or dataflow-based accelerators [119, 136].

GPU Scheduling: Other works improve concurrency via better wavefront- [74, 86, 103, 105, 113, 137, 144, 152, 178, 244–246, 283, 288] and queue- [5, 49, 50, 67, 75, 96, 120, 285] scheduling. They *dynamically* managing intra- and/or inter-kernel concurrency with heuristics (e.g., deadlines, synchronization, cache contention, and stalls). However, unlike GOLDYLOC, they use kernels optimized for *isolation* despite the number of concurrent operations and therefore are globally suboptimal.

Globally optimized kernels: Rammer [161] and Elastic Kernels (EK) partially design globally-optimized kernels. EK dynamically adjusts kernels' WorkGroup/grid sizes to maximize overlap but does not apply to kernels that use shared memory or local data share (LDS) like GEMMs [206]. Rammer re-compiles applications and their kernels to exploit operational parallelism within an application. However, Rammer uses simple GEMM implementations unlike those in state-of-the-art BLAS libraries [14, 201]. Additionally, batched-GEMMs [2, 111, 183] also execute small independent GEMMs within a kernel but require expensive data layout/application changes and are not applicable to heterogeneous and inter-application GEMMs. Finally, unlike RAMMER and batched-GEMMs, GOLDYLOC requires no application changes.

Collectively, these state-of-the-art schemes use a range of solutions to exploit parallelism. However, none of them select kernels optimized for *global* resource environments **and** consider the system's *dynamic* behavior. Moreover, GOLDYLOC is the only approach to provide all four important features.

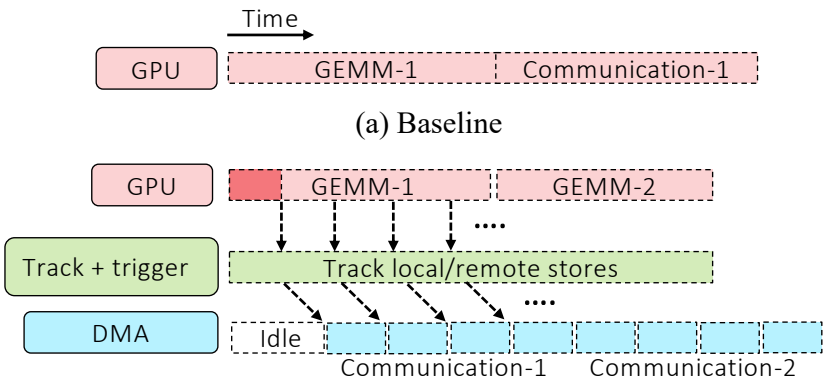
7.8 Chapter Summary

Exploiting concurrency is difficult in GPUs as they use kernels tuned in *isolation*, manage concurrency *statically*, or both. GOLDYLOC solves this for key GEMM operations by (1) tuning kernels for globally shared resources during concurrency, and (2) dynamically controlling how many GEMMs to execute concurrently. GOLDYLOC improves performance by 2.5× max (43% geomean per-app) over sequential execution and 2× max (18% geomean per-app) over concurrent execution in current GPUs.

8 T3: TRANSPARENT TRACKING & TRIGGERING FOR FINE-GRAINED OVERLAP OF COMPUTE & COLLECTIVES

In Chapter 5 we showed that Transformers frequently use distributed techniques which can add considerable communication overheads to their execution. These overheads are especially high with the TP which requires an all-reduce that is typically serialized with other compute during model execution, as shown in Figure 8.1(a). This all-reduce, as we show in Chapter 5, can be a significant proportion of runtime (~45%), resulting in a sub-linear increase in model throughput as the number of devices increases.

While some prior works have sped up communication by up to 2× with *in-network computation*, they are topology-dependent (e.g., requiring switches) and further, cannot eliminate serialized communication from the critical path [130]. Distributed techniques with abundant coarse-grained independent compute (e.g., DP) often overlap (and hide) communication with independent computations to improve efficiency. Although serialized communication scenarios also offer such potential, they require a fine-grained overlap of computation and communication, which presents its own challenges. Enabling their fine-grained overlap in current systems either requires expensive fine-grained synchronization [107] or changes to matrix multiplication (GEMMs) kernels which can be disruptive to GPU software infrastructure [278] (Section 8.2.1). Furthermore, overlapped compute and communication contend for both compute units and memory bandwidth, reducing overlap's efficacy [107, 278] (Section 8.2.2). Prior approaches that reduce contention only address coarse-grained overlap of compute and communication in cases like DP and lack support for fine-grained overlap in serialized collectives [235]. Moreover, they rely on dedicated accelerators. *Therefore, no existing technique achieves a transparent overlap of serialized communication with computation while minimizing resource*



(a) Baseline

(b) Proposal: T3 collectives

Figure 8.1: T3 overview.

contention.

To overcome these, we propose T3 (Figure 8.1(b)). T3 *transparently* fuses producer operations with the subsequent communication by *configuring the producer's output address space* to initiate communication directly on the producer's store, requiring minimal application changes. It uses a lightweight and programmable hardware *tracker* to track the producer/communication progress and *triggers* communication using pre-programmed DMA commands, requiring no additional GPU compute resources for communication. Furthermore, to reduce contention for memory bandwidth between the producer and communication, T3 leverages recently proposed compute-enhanced memories [125, 145] to atomically update memory on stores, thus reducing memory traffic due to communication-related reductions. Finally, T3 employs a simple yet effective arbitration policy between the producer and communication memory streams to minimize any remaining contention.

We extend Accel-Sim [124] to accurately model multi-GPU systems (6% error). Our results show that T3 speeds up sliced Transformer sub-layers from models like Mega-GPT-2 [256] and T-NLG [167] by 30% geomean (max 47%) and reduces data movement by 22% geomean (max 36%).

Furthermore, T3's benefits persist as models scale: geomean 29% for sub-layers in ~500-billion parameter models, PALM and MT-NLG. Overall, T3 transparently overlaps serialized communication with minimal resource contention. This improves compute and network utilization, and in turn, can enable better throughput scaling with increasing device count. This chapter is based on the paper, *T3: Transparent Tracking & Triggering for Fine-grained Overlap of Compute & Collectives*, which appears in ASPLOS 2024 [214].

The relevant background for this chapter is provided in Chapter 2. The rest of this chapter is organized as follows. Section 8.1 motivates the need to optimize for serialized communication. In Section 8.2, we describe challenges with fine-grained interleaving of compute and communication. Section 8.3 provides the details of our proposal T3. We describe the methodology used to evaluate the efficacy of T3 in Section 8.4 and show results in Section 8.5. In Sections 8.6 and 8.7 we discuss T3's applicability/extensions and related work, respectively. Finally, we summarize and conclude in Section 8.8.

8.1 Motivation

8.1.1 All-Reduce is on the Critical Path & can be Large

Transformers require TP [256] to increase the aggregate memory capacity available to them. However, it requires ARs on the critical path (between layers). Figures 2.4(b) and 2.4(c) show the FC sub-layer's original operations versus the operations when sliced across two devices (TP=2 in Figure 2.4(c)). Each device (dotted box) only has a slice of the weights. Since the GEMM corresponding to the second sliced weight only generates a partial output, it requires an AR before the next layer executes (highlighted by "Sliced GEMM→AR"). These GEMM and AR operations execute as separate kernels and are serialized.

These serialized ARs can become a bottleneck. Figure 8.2 shows the execution time breakdown of Transformers between "Sliced GEMM→AR" and other operations for multiple current and futuristic Transformers (setup detailed in Section 8.4.1.2, 8.4.2). For large models (e.g., Mega-GPT-2, T-NLG) we consider 8- and 16-device TP. For very large models (e.g., PALM, MT-NLG) we consider 32-way slicing, and for futuristic ones with one and ten trillion parameters, we consider 64-way sharding. The increasing TP slicing is necessary because these models' larger sizes cannot fit in 16 GPUs [213] and the increased slicing is also enabled by nodes with larger device counts [198, 278]. Like prior work [130, 169, 213], we find that communication is a considerable fraction of the overall runtime: Megatron-GPT-2 (Mega-GPT-2) and T-NLG spend up to 34% and 43% of their training and inference (prompt phase) time on communication. These trends also hold for the very large and futuristic Transformers: communication can be up to 46% and 44% of their runtime, respectively. Additionally, since compute FLOPS scales much more than network bandwidth [77], these proportions will only increase in the future (Section 5.3.3.6). For example, if the GEMMs become 2× faster, communication increases to 75% of model execution time – making scaling to multiple devices extremely inefficient and potentially leaving GPUs idle while communication happens. Thus, addressing serialized AR is critical to Transformer scaling.

8.1.2 Enabling Compute-Communication Overlap

Overlapping collective kernels with independent compute kernels has been key to scaling DNNs in other distributed approaches (e.g., GPipe [99], DP). While TP does not have independent kernels to overlap AR with, we observe that it can benefit from a *fine-grained overlap* with the producer GEMM itself. Transformer GEMMs have large outputs, which are tiled/blocked and require many GPU WGs to complete. Consequently, a GEMM cannot always execute all its WGs concurrently on the limited

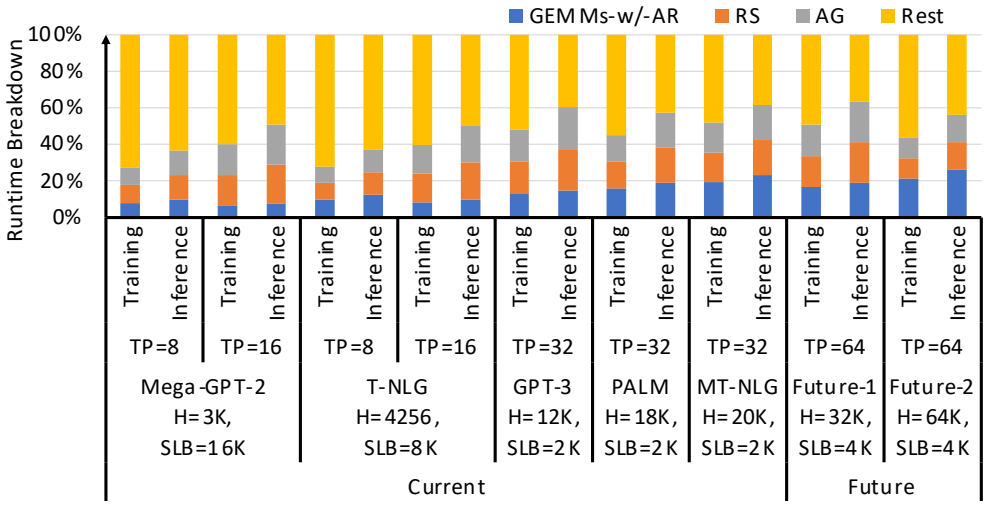


Figure 8.2: Transformer time spent on reduce-scatter (RS) and all-gather (AG) collectives as well as GEMMs which require collectives.

number of GPU CUs. Thus, a GEMM executes and generates output in multiple *stages*, where each stage is a set of WGs that the CUs can accommodate. This holds even for sliced GEMMs that require AR. As shown in Figure 8.3, GEMMs in TP are sliced in the K (or dot-product) dimension which decreases compute per WG, but the output size, WG count, and WG stages remain the same. We utilize this observation to enable fine-grained overlap: communication of one stage's output data can be overlapped with compute of the next stage. However, achieving practical and efficient fine-grained overlap is challenging as we describe in Section 8.2.

8.2 Challenges With Fine-grained Compute-Communication Overlap

This section details key challenges with the fine-grained overlap of compute and communication.

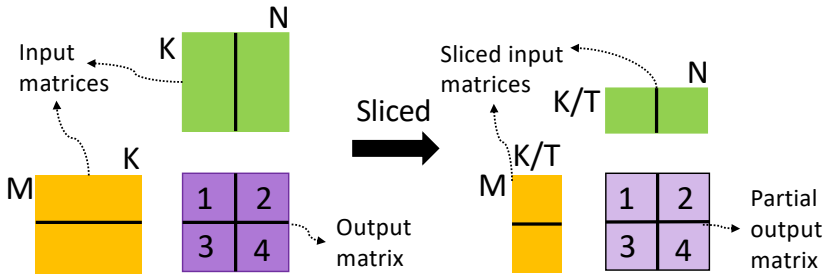


Figure 8.3: GEMM (left) when sliced in the dot-product dimension (right) still generates the same number of data blocks.

8.2.1 Complex & Expensive to Implement in Software

The producer and collective operations execute as separate kernels on GPUs; the producer (GEMM) generates the data, after which the collective orchestrates their bulk communication and reduction. Extending the software for their fine-grained interleaving can be complex and expensive. It would involve breaking the producer and collective into smaller kernels or using dynamic parallelism, both of which can increase launch overheads and synchronization costs. Alternatively, it can be achieved by writing fused GEMM and collective kernels, but this can incur significant programming effort [66, 72, 264, 275]. First, BLAS libraries have hundreds of GEMM kernels optimized for different input sizes and GPU architecture, generated via an expensive tuning process [23]. Second, collectives are also of different types, and each has implementations optimized for different topologies. Creating fused kernels for every combination of GEMM and collective implementations can thus be extremely complex and expensive. Hence, it is imperative to achieve a fine-grained overlap of compute and communication without altering GEMM implementations.

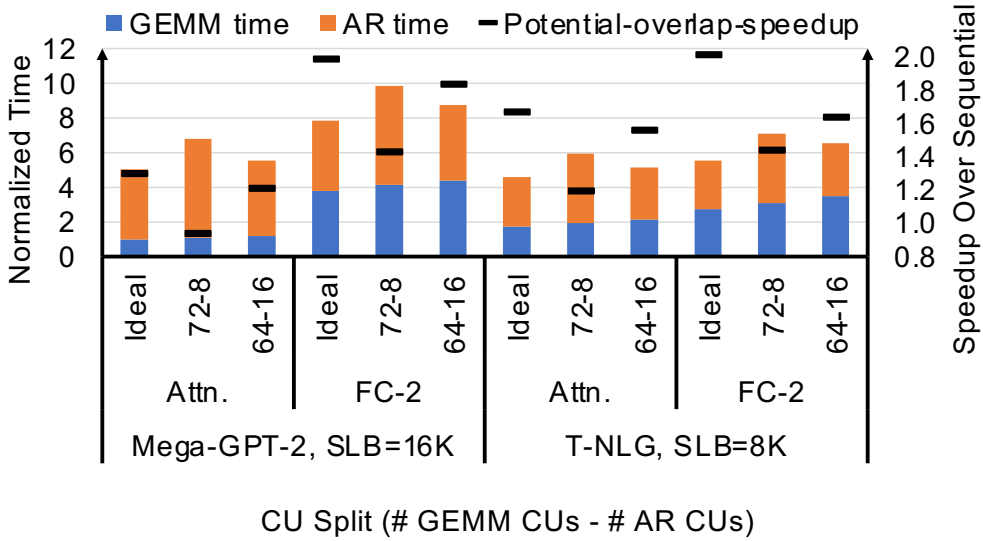


Figure 8.4: Evaluating how the benefits of overlapping GEMM and RS, across model layers, are impacted by CU sharing. The X-axis shows how CUs are split between GEMM and AR, using the GPU setup from Table 8.1, in the format A-B. A represents the number of CUs the GEMM uses, while B represents the number of CUs AR uses. Ideal assumes no sharing, the GEMM has all CUs, and AR is free.

8.2.2 Resource Contention Between Producer & Collective

Overlapped GEMM and AR contend for GPU resources, specifically CUs and memory bandwidth, which slow down overall execution.

8.2.2.1 Compute Sharing

Concurrently executing GEMM and AR kernels must share CUs and their components including L1 cache, LDS, and vector registers. This contention may affect their performance relative to their isolated execution. Figure 8.4 evaluates the impact of concurrently executing GEMM and AR using our setup in Section 8.4.1.1 and Table 8.1. Specifically, Figure 8.4 shows the (normalized) GEMM and AR time for Mega-GPT-2 and T-NLG (with

TP=8) sub-layers (Attn. and FC-2) when run in isolation with varying CU count splits (e.g., the 72-8 bars show GEMM's isolated execution time with 72 CUs and AR's with eight CUs). For each case, it also shows *potential-overlap-speedup*, the speedup overlapping AR and GEMM can obtain versus sequentially executing GEMM and AR when each has all 80 CUs. We calculate the overlapped time as max(GEMM time, AR time). The ideal case assumes no sharing impact: the GEMM has all the 80 CUs and the AR is fast but free (evaluated by running it with all 80 CUs in isolation). As a result, the ideal case has the maximum potential overlap speedup of 1.7× geomean. However, AR slows down considerably (geomean ~41% slowdown) when allocated only eight CUs (72-8 case) compared to when it had all CUs. This significantly decreases the potential-overlap-speedup to 1.2× geomean. While AR performance improves with 16 CUs (only ~7% slowdown in 64-16 case), GEMMs slow down (geomean ~21% slowdown) since they now only have 64 CUs. Overall, while better than the 72-8 case, potential speedups fall short (1.5× geomean) compared to the ideal case. Moreover, this assumes no contention due to memory bandwidth sharing (discussed next) and thus underestimates slowdowns. Overall, sharing of CUs reduces overlapping efficacy and it is crucial to preserve the compute resources dedicated to GEMMs.

8.2.2.2 Memory Bandwidth Sharing

GEMM and AR kernels also compete for memory bandwidth when run concurrently. As shown in Figure 2.9, at each step AR kernels a) read an array chunk from memory to send it to one neighbor GPU and also b) write to memory the chunk it received from another neighbor. Reduce-scatter (RS) additionally requires a memory read of the local chunk for reduction. Moreover, the memory traffic due to AR communication can be bursty. This additional, bursty memory traffic due to AR can slow down critical memory accesses by the producer GEMM, with the impact higher

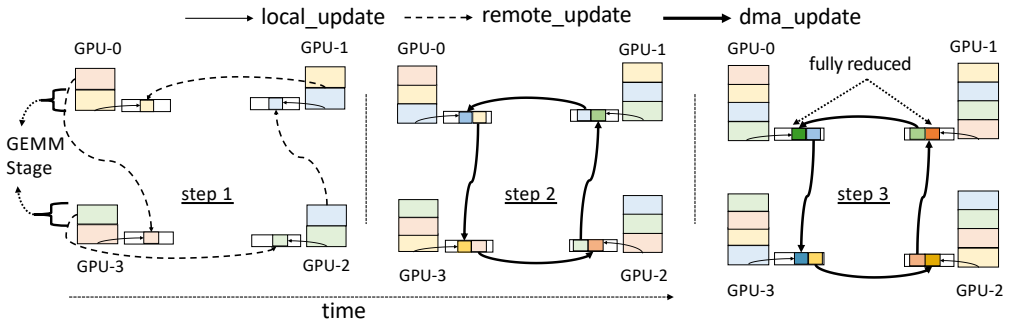


Figure 8.5: Overview of fused GEMM and ring reduce-scatter with T3 on a four-GPU node.

for GEMMs for which inputs do not fit in GPU's last level cache (LLC) as we will show in our evaluation in Section 8.5.1.2 and Figure 8.15. Thus, to enhance overlap efficiency, it is essential to limit memory traffic due to communication and/or limit their contention with GEMM.

Prior work also studied contention between communication and computation [235], albeit in DP setups with coarse-grained GEMM and AR overlap. They show that AR slows down by up to 2.4× when run concurrently with GEMMs, and the slowdown is even higher when run concurrently with memory-intensive embedding lookups in recommendation models. For TP, they obtain a 1.4× slowdown when executed concurrently with GEMMs.

8.3 T3: Transparent Tracking & Triggering

To overcome the aforementioned challenges of complex software and resource contention with fine-grained overlap of compute and communication, we propose T3.

8.3.1 T3 Overview

Modern GPUs first execute the producer GEMMs and store their outputs in their local memory. Afterwards they initiate the collective operation (Section 2.9). T3 instead initiates the collective immediately as GEMMs generate data to enable fine-grained overlap. It uses a ***track & trigger*** mechanism to monitor GEMM's/collective's progress and to orchestrate communication, requiring no additional CUs (Section 8.3.2). It leverages ***near-memory compute*** (NMC) for reductions to reduce memory traffic due to communication (Section 8.3.3). Finally, it does these ***transparently***, with minor kernel modifications (Section 8.3.4).

Figure 8.5 illustrates a four-device reduce-scatter (RS) overlapped with its producer GEMM. This GEMM executes in multiple *stages* of WGs dictated by its input and kernel implementation (Section 8.1.2), while RS executes in multiple *steps* dictated by the number of devices involved (Section 2.4.3.1). For simplicity of illustration, we show the number of GEMM stages to be one more than the number of required ring steps. In each step, a GEMM stage's execution and reduction of its output happen in parallel to the communication of the previous stage output. In the first step, the output is communicated to remote devices directly by the GEMM (*remote_update*). The later, *steady state*, steps require a DMA (*dma_update*). For N devices, this steady state step is performed $N-2$ times, on different chunks. Focusing on GPU-0 in the steady state, step-2, as shown in Figures 8.5, the GPU executes/generates output for GEMM stage-3 while also receiving (via DMA) a copy of stage'3 output (blue) from its neighbor, GPU-1. This occurs in parallel to GPU-0's DMA of the reduced copy of GEMM stage-2 data (yellow) to GPU-3, thus overlapping communication. T3 leverages *NMC* to atomically update memory locations on these local and DMA updates, resulting in a partially reduced copy of the stage-3 chunk without requiring additional reads or GPU CUs (Section 8.3.3). Once they complete, GPU-0 initiates a *dma_update* of the

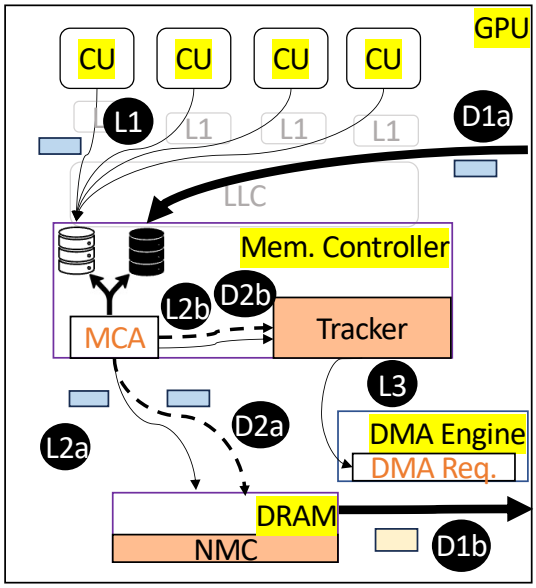


Figure 8.6: GPU with highlighted T3 enhancements (in orange) executing a steady-state fused GEMM-RS step.

chunk to its neighbor's (GPU-3) memory as shown in step-3. This automatic tracking of updates and DMA triggering is done using a lightweight and programmable hardware Tracker, further reducing dependency on GPU CUs (Section 8.3.2). These remote / DMA updates are done transparently by configuring the GEMM's output address mapping, with minor application and kernel modifications (Section 8.3.4).

We also make minor runtime and hardware changes to improve T3's performance. To enable the perfect overlap of GEMM and RS in Figure 8.5, we stagger the scheduling of GEMM WGs across GPUs (Section 8.3.4). Moreover, we also augment the memory system with a simple yet effective memory controller arbitration (MCA) policy to manage memory contention between compute and communication (Section 8.3.5).

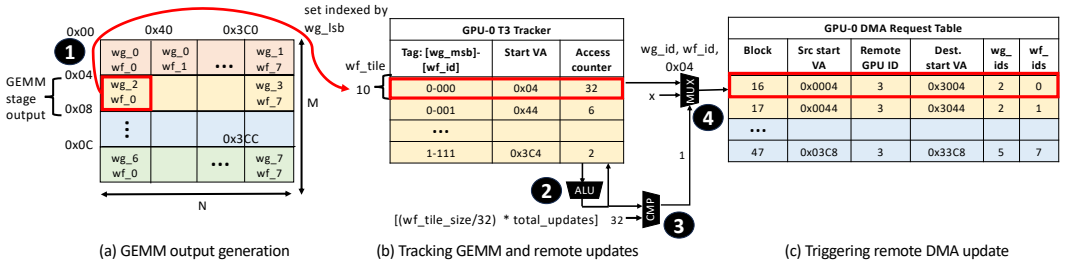
Figure 8.6 shows a GPU with T3's enhancements (in orange) executing the steady state step described above. The GPU executes the GEMM to

generate local updates for a stage (L1). Concurrently the GPU receives DMA updates for the same stage (D1a) and sends DMA updates for the previous stage (D1b). At the memory controller, the modified *MCA* arbitrates between the local and DMA traffic to prevent contention. Following this, the updates are sent to *NMC-enhanced DRAM* (L2a,D2a) while the *Tracker* is updated with their progress (L2b,D2b). Once the Tracker observes the required local and DMA updates to a memory region, it triggers their DMA transfer to the neighbor GPU (L3).

We use the 4-GPU GEMM-RS overlap as a running example to describe T3. RS is more challenging to overlap due to reductions and extra memory traffic. Further, the ring configuration is more complex than others. Thus, we detail T3 using ring-RS and discuss additional collectives in Section 8.6.1.

8.3.2 T3 Tracking & Triggering

T3's *programmable track & trigger* mechanism is key to transparently enabling fine-grained overlap of producer and collective without using compute resources. As shown in Figure 8.7, T3 automatically transfers copies of data between devices when ready (e.g., in Figure 8.5, T3 triggers DMA update of stage-2 data from GPU-0 to GPU-3 once both GPU-0's local and GPU-1's remote updates are complete). This is enabled by a lightweight *Tracker* at the memory controller, that tracks local and remote/DMA accesses to memory regions and triggers a DMA transfer once the required accesses are complete. Since the condition when a DMA is triggered (e.g., number of remote and local updates) and DMA transfer details (e.g., addresses, operation type) vary per collective type and implementation, they are programmed ahead of time using address space configuration (detailed in Section 8.3.4 and Figure 8.10).



(a) GEMM output generation (b) Tracking GEMM and remote updates (c) Triggering remote DMA update

Figure 8.7: T3 Track & Trigger.

8.3.2.1 Tracker

The Tracker tracks both local and remote memory updates of a GEMM stage and triggers its DMA. As shown in Figure 8.7(a) and (b), it does so at wavefront (WF, i.e., a group of threads that execute in lockstep) granularity ① – i.e., the Tracker tracks the memory region a WF updates. This assumes tiled GEMM implementations and that each WF/WG generates a complete tile of data, as is the case in all evaluated GEMMs [30, 201]. However, T3 can also handle other implementation (Section 8.6.7). An update increments the counter at its corresponding WF's (*wf_id*) Tracker entry ②. This is done by all local, remote, and DMA updates that arrive at the GPU's memory controller (e.g., GPU-0 does not track GEMM stage-1 as its WFs neither write locally nor are its remote updates received). The incremented counter value is checked for a maximum threshold, which is set to the product of WF output size (*wf_tile_size*) and the total updates expected per element ③. The *wf_tile_size* is determined by the GPU driver using the output size and WF count ($(M * N)/\#WF$). The total updates expected per element for ring-RS is two but changes with collective type/implementation and is thus configurable (detailed in Section 8.3.4). Once the threshold is reached, the final write triggers the DMA (④ in Figure 8.7(c) and detailed in Section 8.3.2.2). The Tracker is checked once the accesses are enqueued in the memory controller queue (MCQ) and thus are not in the critical path.

WF-based tracking is beneficial as a producer's (or GEMM's) stage may not update contiguous memory regions. As shown in Figure 8.7(a) this can happen due to column-major allocation of arrays in BLAS libraries [14, 201] and row-major scheduling. This makes address-based tracking expensive (requires storing several addresses or complex table indexing functions) which WF-based tracking avoids. The Tracker has a total of 256 entries, indexed using the WG ID's LSBs, wg_lsb (8 bits). Each entry is set associative and is tagged using wg_msb, wf_id. wg_msb is $log\_2(maxWGsPerStage/256)$ bits and wf_id is three bits for a maximum of eight WFs per WG. We set the maximum entries based on the maximum WGs possible in a producer stage. Each entry has a starting virtual address (smallest address per WF), and an accesses counter, making the Tracker size 19KB. The tracking additionally requires the source wg_id and wf_id as metadata in memory accesses and forwarding of their virtual addresses to the memory controller (to trigger the DMA in Section 8.3.2.2).

8.3.2.2 Triggering DMA

Once the required accesses to a WF's memory region are issued, T3 DMAs the data to the remote GPU (④ in Figure 8.7(c)). As shown in Figure 8.7(c), the DMA commands are pre-programmed by the GPU driver and are configurable (detailed in Section 8.3.4 and Figure 8.10) as the DMA regions/operations can differ based on the collective type and implementation. The granularity of the DMA block/table entry is set to be equal to or larger than the Tracker granularity (wf_tile). The memory access which completes the required accesses at the Tracker entry (Section 8.3.2.1) marks the corresponding DMA entry ready and also populates it with the wg_id and wf_id which are required by the destination GPU's Tracker. If DMA blocks are a multiple of wf_tile, an additional counter per DMA entry can track their completion. Using the pre-programmed starting source/destination virtual address, wf_tile_size, and the output

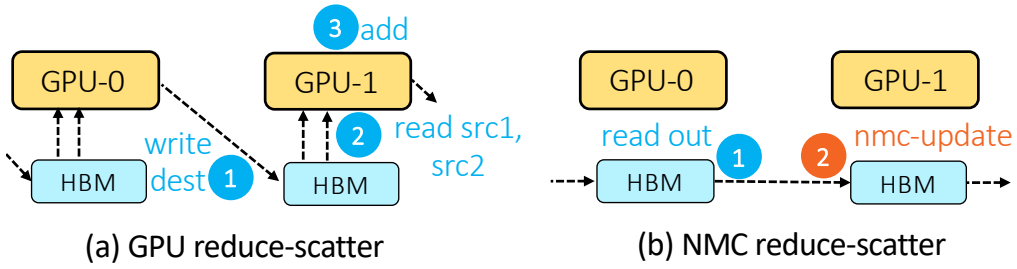


Figure 8.8: HBM reads & writes in steady-state reduce-scatter step.

dimensions (M, N), the DMA engine dynamically generates the remaining virtual addresses to initiate the DMA.

8.3.3 Near-Memory Reductions

To perform reductions on producer and DMA updates without occupying GPU compute resources, T3 leverages compute-enhanced memories. We assume an HBM-based DRAM architecture with near-memory op-and-store support as has been proposed by recent works [174, 220]. We envision such compute support to be implemented via ALUs near DRAM banks as has recently been proposed by memory vendors [125, 145]. However, T3 can also leverage other reduction substrates (Section 8.6.4).

T3 leverages this near-memory computing (NMC) capability to enable GEMM stores and DMA transfers to directly update and reduce copies of data, when required by collectives. For DMA transfers, the operation type (store vs. updates) is directly specified in the command (address space configuration in Figure 8.10 and Section 8.3.4). For GEMMs, we utilize two flags. First, we use an "uncached" flag during memory allocation to ensure that the output is not cached in any GPU's caches (such allocations are supported in existing GPUs). Thus, writes are directly sent to DRAM which acts as the point of aggregation for all (local, remote, DMA) updates. The queuing of updates in the memory controller queue guarantees their

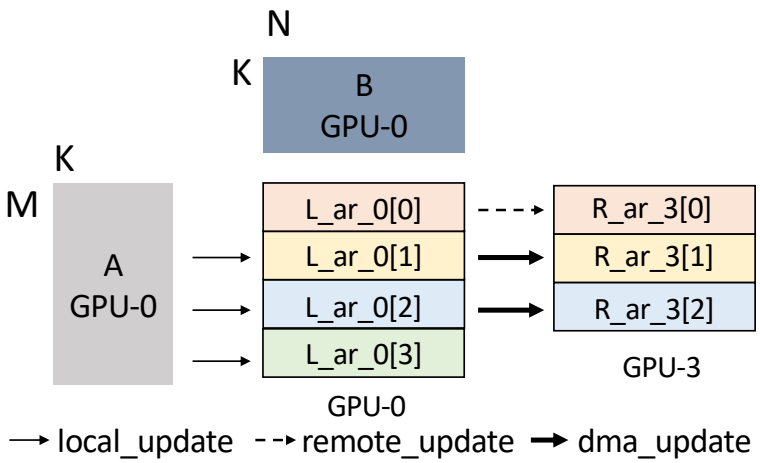


Figure 8.9: Remote address mapping for T3 GEMM-RS over four GPUs.

atomicity; at any given time, only a single instruction can be issued and executed by near-bank ALUs. Second, we use an "update" flag in the GEMM API call to enable stores of the GEMM to update the DRAM. The "update" flag is sent (via kernel packets [24]) to the CUs to tag the kernel's stores with one-bit "update" info (similar to prior work [112, 114, 219]). These are processed by the memory controller to generate the op-and-store commands.

In addition to freeing up CUs for GEMMs, NMC helps reduce memory traffic due to communication. Figure 8.8 shows memory accesses in a steady-state RS step in baseline and with T3. In baseline RS, CUs read two copies of data (local copy, and received copy from the previous neighbor) and write the reduced data to the next neighbor's memory. T3 only requires one read of the data to DMA update the neighbor GPU memory using NMC. Overall, T3 with NMC reduces the dependence on GPU CUs and further reduces (or eliminates if using *direct-RS*, discussed further in Section 8.6.1) data movement required for communication.

8.3.4 Configuring Producer's Output Address Space

Modifying producer kernels, especially for many GEMMs of different shapes and sizes, to fuse and overlap collectives, can be impractical (Sec-

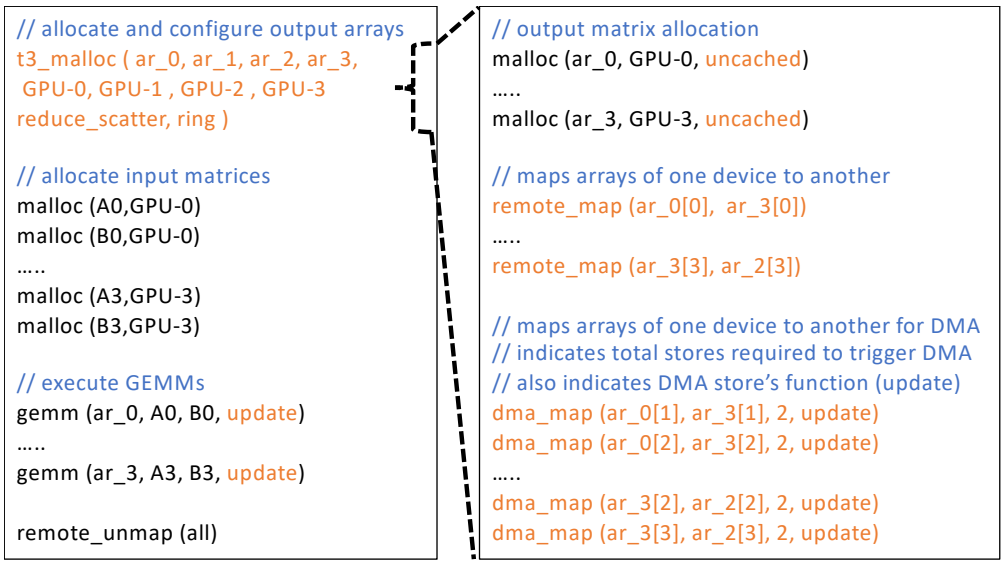


Figure 8.10: Configuring producer output for T3 GEMM-RS over four GPUs.

tion 8.2.1). T3 avoids this by configuring the producer's output address space mapping which is used to program the Tracker and DMA commands. Figures 8.9 and 8.10 show this configuration for GPU-0 from the fused GEMM-RS example in Figure 8.5.

Since there are four devices, GEMM's output array is chunked four ways. In GPU-0, the GEMM writes its stage-1 output directly to GPU-3's memory (step-1 in Figure 8.5), while its stage-2 and stage-3 output is first written to local memory and later DMA'd to GPU-3 (stage-4 is only written locally once and is not DMA'd). Thus, GPU-0 requires memory mappings of these chunks with that of GPU-3 as shown in Figure 8.9. This configuration differs per collective type and topology-optimized implementation (see Section 8.6.1) and, similar to modern collective implementations, can be pre-defined in collective libraries [10, 192]. Figure 8.10 shows an example of this using pseudo-code.

The configuration in Figure 8.10 defines this mapping for the GEMM

output using two different API calls: *remote_map* and *dma_map*. *remote_map* is used for fine-grained remote writes/updates (for stage-1), which uses existing GPU support for peer-to-peer load/store by threads [195]. Conversely, *dma_map* is used for coarse-grained DMA writes/updates (for stage-2,3) which leverages existing support for memory copies by DMA engines in GPUs (e.g., DirectGMA and others [171, 172, 195]). A *dma_map* call also defines the DMA functionality (store vs. update), and its triggering condition (number of stores/updates per element). It can also be extended to specify granularity (*wf_tiles* per DMA block in Figure 8.7(c)). These calls are used to pre-program the Tracker and DMA commands to enable automatic communication of data when ready (Section 8.3.2).

Fusion in ring-based collectives also benefits from producers (on different devices) generating data chunks in a staggered manner. In Figure 8.5, GPUs stagger the generated data by one stage; in step-1, GPU-0 executes stage-1, while GPU-1 executes stage-2, and so forth. This is enabled by staggering WG scheduling across devices. Alternatively, it can also be enabled by fetching appropriate implementation from BLAS libraries with staggered output tile-to-WG mapping amongst producer kernels. Overall, configuring the output address space mitigates the need to change GEMM implementations to enable fusion with collectives.

8.3.5 Communication-aware MC Arbitration (MCA):

Finally, careful scheduling of memory accesses by the producer kernel and those resulting from communication is crucial to efficiently overlap them. In Section 8.5.1 we show that a memory controller (MC) arbitration policy which a) round-robins between issuing memory accesses from the compute and communication streams and b) falls back to the other stream if the current stream is empty, results in producer kernel slowdowns. Communication-related memory accesses appear in bursts and can occupy DRAM queues, stalling the compute kernel's critical memory

reads/writes. Simply prioritizing producer kernel accesses as they appear is also insufficient as prior communication-related memory accesses may already occupy DRAM queues. Finally, giving the local compute stream dedicated access results in wasted cycles and memory bandwidth under-utilization. Thus, an efficient overlap of compute and communication requires a dynamic arbitration policy that addresses both contention and under-utilization.

We implement a simple yet dynamic arbitration policy to overcome this. The MC always prioritizes compute stream accesses, but if empty, falls back to communication stream. Additionally, it monitors the DRAM queue occupancy and only issues communication-related accesses if occupancy is below a threshold. This ensures sufficient room in the queues for future compute stream accesses and prevents their stalls. The occupancy threshold depends on the memory-intensiveness of compute kernels (e.g., smaller if memory-intensive, and vice-versa). This is determined dynamically: MC detects the memory intensiveness of a kernel by monitoring occupancy during its isolated execution (the first stage in Figure 8.5). Finally, the MC tracks cycles elapsed since the last issue from the communication stream and prioritizes it if it exceeds a limit to ensure it is not starved. Additionally, the communication stream is drained at the producer kernel boundary.

8.4 Methodology

8.4.1 Setup

8.4.1.1 Multi-GPU Simulation

Although a number of popular, open source GPU simulators are publicly available [37, 83, 147, 286], we chose to evaluate T3 using Accel-Sim [124] because it provides high fidelity for modern GPUs [122]. Like

System	
#GPUs	8, 16
Inter-GPU Interconnect	Ring, 150 GB/s Bi-directional 500 ns link latency
Per-GPU Config	
#CUs	80, 1.4 GHz
Per-CU Config	2K threads, 128KB unified LDS + L1 cache (with no write-allocate), 256KB RF
L2	16MB, 64 banks, 1.4 GHz
HBM2	1 TB/s, 1 GHz, CCDWL=4, Bank Grp.=4, rest [48]

Table 8.1: Simulation setup.

prior work [123], we extended Accel-Sim to simulate a multi-GPU system. We observe that in a multi-GPU DNN setup all GPU's executions are homogeneous (Figures 2.4 and 8.8). Thus, we evaluate both our multi-GPU baseline and T3 by modeling all the activities pertaining to a single GPU. This includes modeling the Tracker which is accessed/updated in parallel with the store/DMA operations and uncached NMC updates. Although we do not model the DMA engine in the simulator, we do model its inter-GPU communication (communication resulting from RS both in the baseline and T3's fused GEMM-RS) by executing the compute operations (e.g., GEMM [30, 201]) in Accel-Sim and using Accel-Sim's front-end tracing functionality to inject the additional inter-GPU communication traffic. The Tracker's DMA triggering overheads are negligible since the DMA commands are pre-queued during the setup process (Figure 8.10) as is often done, especially for ML operations which are repetitive [100]. Table 8.1 details the GPU configuration we use to evaluate T3, which is the latest GPU architecture Accel-Sim completely supports. Commercial GPUs with such a configuration support a 150 GB/s interconnection ring bandwidth [182]. Since recent GPUs frequently scale compute faster than other resources, we also evaluate another configuration with increased CU count while the other parameters stay the same in Section 8.6.5.

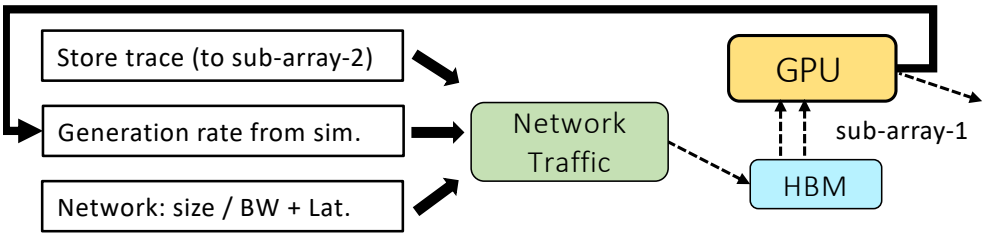


Figure 8.11: Simulating multi-GPU reduce-scatter.

Figure 8.11 describes our multi-GPU simulation of RS. In each RS step, a GPU performs a reduction of a sub-array and sends it to the neighbor GPU while also receiving a reduced sub-array (corresponding to a different chunk) from another neighbor GPU (Figures 2.9 and 8.8(a)). The simulator executes the reduction of the array as-is. Simulating the incoming network traffic requires: (a) determining packet addresses, (b) generating packets at the appropriate rate, and (c) modeling the interconnect costs. Packet addresses are determined using the store trace of WGs from the reduction kernel. Next, since GPU executions are homogeneous, remote traffic is generated at the same rate as the GPU generates the reduction output (which is filtered out to be sent to remote GPU). This also implicitly includes slowdowns due to compute/communication interference at the remote GPU. Finally, we add the interconnect costs to these packets as they arrive, assuming a simple link bandwidth and latency model of the interconnect. To validate this setup, we compare simulated RS times on four GPUs with hardware measurements from a four-GPU node with AMD Instinct™ MI210 GPUs [25] with same ring network bandwidth as simulated (Table 8.1). Figure 8.12 shows that simulation closely follows hardware trends for a range of sizes (6-192 MB): 6% geomean error versus the ideal dotted line.

Near-Memory Computing: We modify the simulator's HBM to model NMC updates. Further, memory vendor proposals indicate that NMC operations can be issued without a significant increase in DRAM timings;

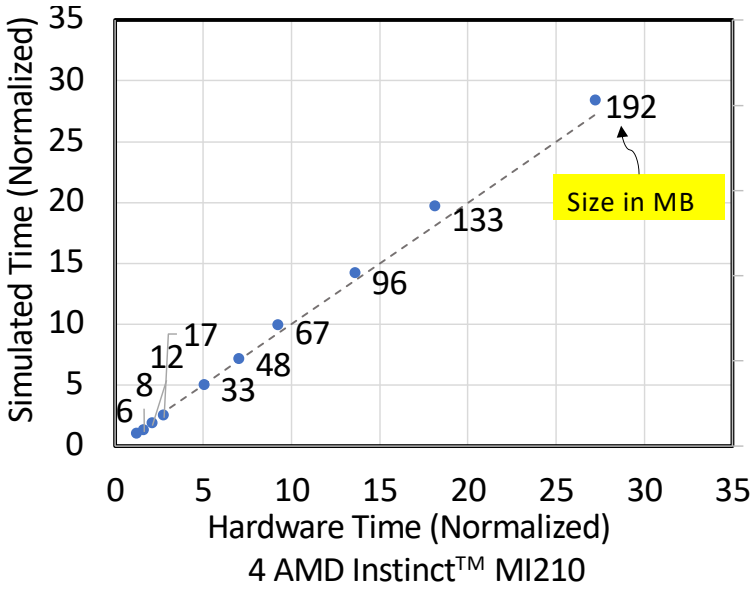


Figure 8.12: Validation of multi-GPU reduce-scatter simulation.

back-to-back NMC operations can be issued to the same bank group with the same column-to-column access (CCDL) delay [125]. To model the additional cost of NMC op-and-store operations (Section 8.3.3), we modify the simulator's HBM to use a 2× higher CCDL delay (termed CCDWL) following those operations (see Table 8.1).

8.4.1.2 End-to-End Transformer Iteration

To evaluate end-to-end iterations with T3, we scale the GEMMs and RS times in the baseline Transformer breakdown (shown in Figure 8.2) by their simulated speedups (described in Section 8.4.1.1). We leverage a combination of hardware data and analytical modeling as done by prior works [169, 213] to get the end-to-end breakdowns of models in their distributed setups. We use a single-GPU mixed-precision [165] execution of MLPerf v1.1 [168] BERT on an AMD Instinct™ MI210 accelerator (GPU) [25] and scale its operation times based on changing hyperparameters and setup (e.g., sliced GEMM). This is beneficial as it helps us

Model Name	Hyperparams	Inputs	TP degree
Mega-GPT-2	H=3072, L=74	SL=1K, B=16	8, 16
T-NLG	H=4256, L=78	SL=1K, B=8	8, 16
GPT-3	H=12K, L=96	SL=1K, B=2	32
PALM	H=18K, L=118	SL=1K, B=2	32
MT-NLG	H=20K, L=105	SL=1K, B=2	32

Table 8.2: Studied models, their hyperparameters & setup.

evaluate larger futuristic models (Transformer models are similar differing only in layers size/counts [169, 215]) and takes into account several GPU optimizations for Transformers [59, 65] already in MLPerf implementations. Our projections further match those measured by prior works. For example, AR's percentage runtime contribution projected for Mega-GPT-2 with TP-16 matches prior works' measurements on a similar system configuration [130].

8.4.2 Applications, Deployment & GEMMs

Models and their deployment: Since Transformers are fast-evolving, we evaluate T3's impact on a range of Transformer models and TP degrees (Table 8.2). For Megatron-GPT-2 (Mega-GPT-2) [256] and T-NLG [167] we use 16K and 8K input tokens (= input-length * batch-size) and TP degrees of eight and 16, given their modern intra-node setups [107, 130, 167, 256]. For larger Transformers like PALM [55], GPT-3 [41], and MT-NLG [263]) we use a higher slicing degree of 32 given their increasingly large memory capacity requirements [213] and availability of nodes with larger device counts that can enable this slicing [115, 198, 252]. Note that we consider similar TP degrees for both training and inference. While the inference memory capacity requirements are smaller compared to training a model (since there are no gradients, optimizer states, and master weight copies during inference), higher TP degrees are preferred to minimize latency, even though the reduction in latency may not be proportional to

the increase in TP. For instance, the LLaMA-7B model (with a size of 13GB using 16-bit parameters) can fit on a single device but is deployed with a TP of 8 for optimal performance [224]. Therefore, it is crucial that we evaluate and improve these setups. Finally, we evaluate mixed-precision training which entails half-precision (FP16) forward and backpropagation and single-precision (FP32) weight updates. Similarly, we evaluate FP16 inference.

GEMMs: GEMMs from the aforementioned applications are simulated using implementations from state-of-the-art BLAS libraries [30, 201]. Most GEMMs (including all GEMMs we evaluate) use a tiled GEMM implementation where each WG generates a complete tile of data (other implementations discussed in Section 8.6.7). Further, we evaluate GEMMs with both non-transposed (e.g., backward GEMMs) and transposed (e.g., forward GEMMs) input tensors, as observed in MLPerf's BERT [164, 237].

8.4.3 Configurations

To evaluate T3's efficacy we use the following configurations:

- ***Sequential***: is the baseline configuration. Like modern systems, sequential executes sliced GEMMs and the following AR kernels sequentially.
- ***T3***: is our proposal which fuses and overlaps GEMM with RS (as described in Section 8.3), followed by sequential all-gather (AG).
- ***T3-MCA***: uses fused GEMM-RS as in T3, but also includes the memory controller arbitration (MCA) discussed in Section 8.3.5.
- ***Ideal-GEMM-RS-Overlap***: represents ideal GEMM and RS overlap in software. Thus, its performance is the maximum of the GEMM's and the RS's isolated kernel execution times, followed by the AG

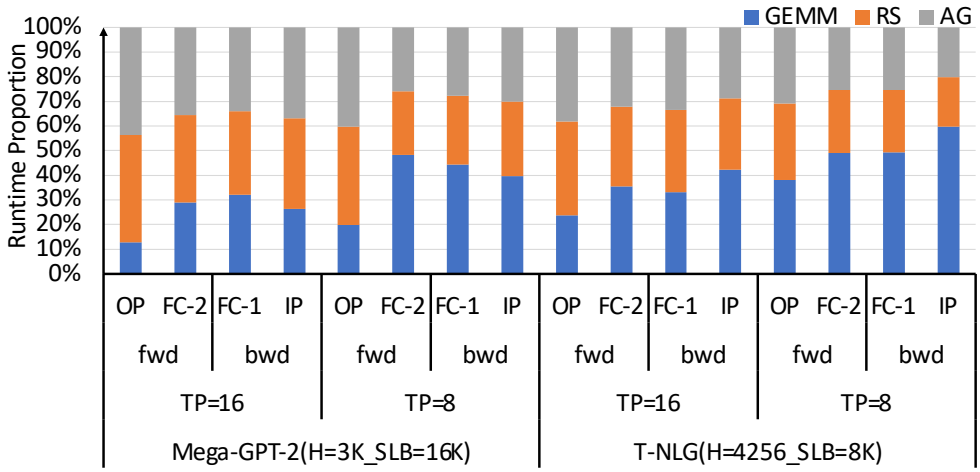


Figure 8.13: Transformer sub-layer runtime distribution.

time. Moreover, it assumes no dependency constraints or resource contention between GEMM and RS.

- ***Ideal-RS+NMC***: uses RS with near-memory computing, which can provide additional speedup beyond a perfect overlap. Thus, its performance is max(GEMM, RS+NMC) over Ideal-GEMM-RS-Overlap.

8.5 Results

8.5.1 Execution Time Distribution & Speedups

Figures 8.13 and 8.14 show results for all sliced sub-layers in Transformers which require an AR: output projection (OP) and fully-connected-2 (FC-2) in forward pass (fwd) and fully-connected-1 (FC-1) and input projection (IP) in backprop (bwd). We show these for Mega-GPT-2 and T-NLG, as well as two TP setups (TP of 8 and 16). Figure 8.13 shows each case's runtime distribution between the GEMM, RS, and AG. Figure 8.14 shows their speedup over *sequential* using *T3*, *T3-MCA*, as well as their speedups assuming an ideal overlap of GEMM with RS (*Ideal-GEMM-RS-Overlap*)

and additional speedups resulting from a faster RS with NMC (*Ideal RS+NMC*).

8.5.1.1 Ideal Speedups

Figure 8.14 shows the ideal possible speedups and breaks them into two parts: first from overlapping the GEMM and RS kernels (*Ideal-GEMM-RS-Overlap*) and second from improved RS performance due to NMC (*Ideal RS+NMC*).

In Figure 8.14 Ideal-GEMM-RS-Overlap (without resource and data-dependency constraints) shows considerable benefits from overlapping the producer GEMM and following RS: 50% max speedup and 35% geomean versus Sequential. Speedups vary both within and across models and depend on the isolated execution times of GEMM and RS (Figure 8.13). The situations where the GEMM and RS runtimes are similar (similar proportions in Figure 8.13) have the maximum potential since the GEMM hides all of RS's cost. For example, FC-1 in T-NLG with TP=16 obtains 50% speedup. Alternatively, the cases in which the GEMM and RS times are skewed show the least benefit since most of the GEMM or RS cost is exposed. For example, Ideal-GEMM-RS-Overlap speedup is only 15% for OP in Mega-GPT with TP=16. However, the latter is uncommon and is a consequence of slicing a very small layer (OP is the smallest among all). It does not hold for other sub-layers within the same model, or larger models as shown in the figures (also see Section 8.5.4). For a given hardware setup, these execution time ratios, and thus Ideal-GEMM-RS-Overlap speedups are dictated by layer parameters [213].

In Figure 8.14 Ideal-RS+NMC shows that additional speedup is possible beyond what perfect overlap provides. Besides freeing all the CUs for GEMMs, performing RS reductions near memory also lowers RS's memory traffic (described in Section 8.3.3). This speeds up RS by 7% and 3% with TP=8 and TP=16, respectively. NMC only reduces RS's final step

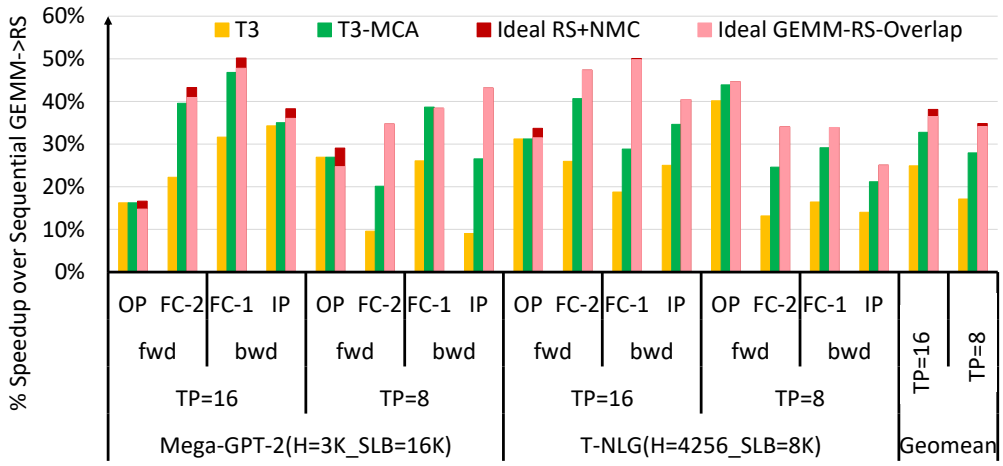


Figure 8.14: Transformer sub-layer speedups with T3

.

time as interconnect costs dominate all prior steps and thus its runtime benefit decreases as TP, and thus total steps, increases. As shown in Figure 8.14, this faster RS can reduce overlapped time and provide additional speedups of up to 4%. Intuitively, the impact of a faster RS is only evident in layers in which RS is longer running than GEMM and is otherwise hidden when overlapped.

8.5.1.2 T3 Speedups

T3 transparently overlaps GEMMs with their corresponding consumer RS in a fine-grained manner. Moreover, T3's lightweight track-&-trigger mechanism and use of near-memory compute frees all CUs for GEMMs and reduces DRAM traffic (Figure 8.16 and Section 8.5.2), respectively. Thus, T3 achieves speedups of up to 39% (20% geomean, yellow bars, Figure 8.14).

Individual speedups vary considerably and are largely impacted by the extent of contention between DRAM traffic from the GEMM and the concurrent, RS (details in Section 8.5.2). For OP layers, T3 achieves close to the Ideal-GEMM-RS-Overlap speedups, and even exceeds them in certain

cases. This happens because the OP GEMMs are small and fit largely in the LLC, having very small DRAM read traffic in Sequential (shown in Figure 8.16). Thus, the additional DRAM traffic from the overlapped RS in T3 has little impact on the GEMMs' progress/execution. Instead, T3 further improves RS runtimes in these cases via NMC and enables part of the additional Ideal-RS+NMC speedups. Finally, although the track & trigger mechanism operates at a small WF granularity, generally data from multiple WFs/WGs of a GEMM stage are ready to be sent concurrently, resulting in high network bandwidth utilization. Furthermore, even when this is not true, T3 can tolerate this because compute/GEMM execution and communication are overlapped, hiding the latency.

In many other cases, and especially the much larger FC layers, the benefits are far from those with Ideal-GEMM-RS-Overlap (>15% slower). Figure 8.15 shows the DRAM traffic (Y-axis) and the GEMM slowdown (X-axis) with fine-grained overlapping, compared to the GEMM's isolated execution. An isolated GEMM as shown in Figure 8.15(a) executes in multiple stages (Section 8.1.2), each with a read phase (blue) followed by a bursty write phase, which limit read traffic. Overlapping RS induces additional DRAM traffic, as shown in Figure 8.15(b). Besides additional traffic, in T3, GEMM, and RS writes directly update memory using NMC (Section 8.3.3). These additional bursts of reads (RS_reads for a stage are issued as soon as both the local and neighbors' copies have updated the memory) and updates (RS_updates for the next stage from the previous neighbor) can further stall local GEMM reads as shown, causing GEMM to slow down considerably.

8.5.1.3 T3-MCA Speedups

T3-MCA (Section 8.3.5) limits GEMM reads stalls due to bursty RS traffic (Section 8.5.1.2, Figure 8.15) using a simple arbitration logic. It prevents RS traffic from completely occupying DRAM queues by limiting

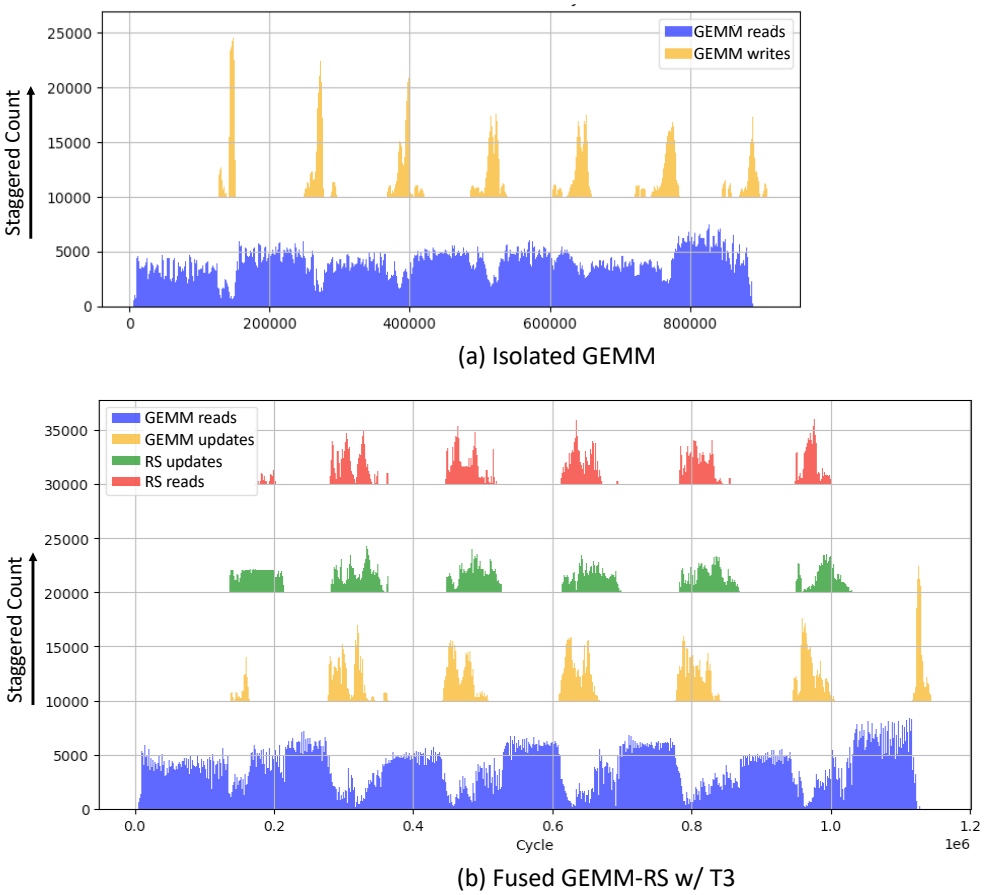


Figure 8.15: Overall DRAM traffic in (a) baseline GEMM, (b) T3, for T-NLG FC-2 with TP=8 and SLB=4K.

communication-related accesses when a DRAM queue occupancy reaches a threshold (5, 10, 30, or no limit) determined by the memory intensity of the GEMM kernel. T3-MCA provides considerable benefits over sequential execution; maximum of 47% and geomean of 30% (29% maximum and 13% geomean over T3). Furthermore, the geomean speedup with T3-MCA is only 5% smaller than Ideal-GEMM-RS-Overlap. There are individual cases where T3-MCA is far from ideal (e.g., FC-1 in T-NLG with TP=16). These represent cases where L2 bypassing (for near-memory update) of GEMM writes hurts the GEMM's performance. Consequently, the overall

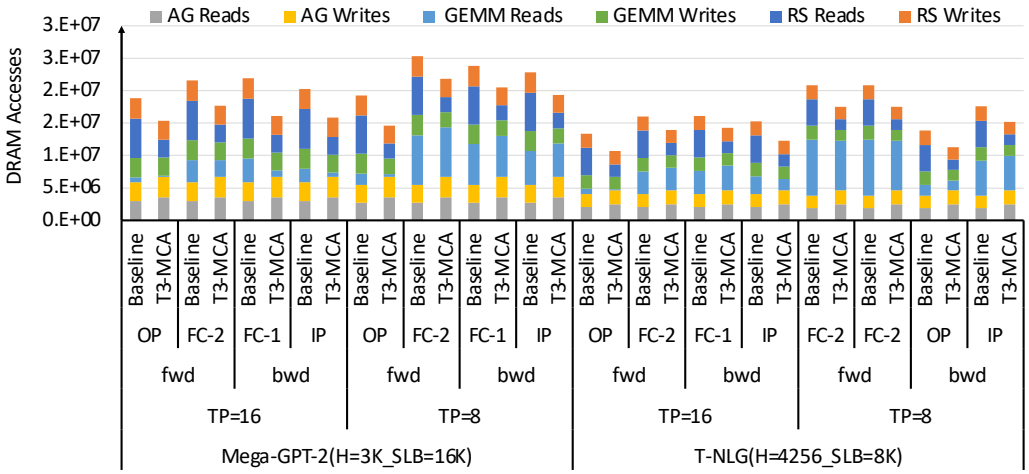


Figure 8.16: DRAM access per sub-layer.

overlapped runtime also increases.

8.5.2 Data Movement Reductions

Besides improved performance, T3 and T3-MCA also reduce data movement to and from DRAM by a maximum of 36% and an average of 22% for the sub-layers. Figure 8.16 shows the total memory accesses and their detailed breakdown (amongst GEMM, RS and AG reads/writes) for a single GPU across all cases. While the AG reads/write remain constant between baseline (sequential) and T3-MCA, there is a combination of reasons which impact the rest: (a) fusion of GEMM and RS eliminates local writes from GEMM's first stage and reads from RS's first step, (b) near-memory reductions eliminate reading of partial copies every RS step, as well as the reads and writes in the final step's reduction, and (c) LLC bypassing of GEMM's output writes improves input read caching for cache-sensitive GEMMs, reducing GEMM's local reads. These impacts also vary depending on the TP degree: the one-time reductions (in the first and last RS step) have a much higher impact with smaller TP degrees

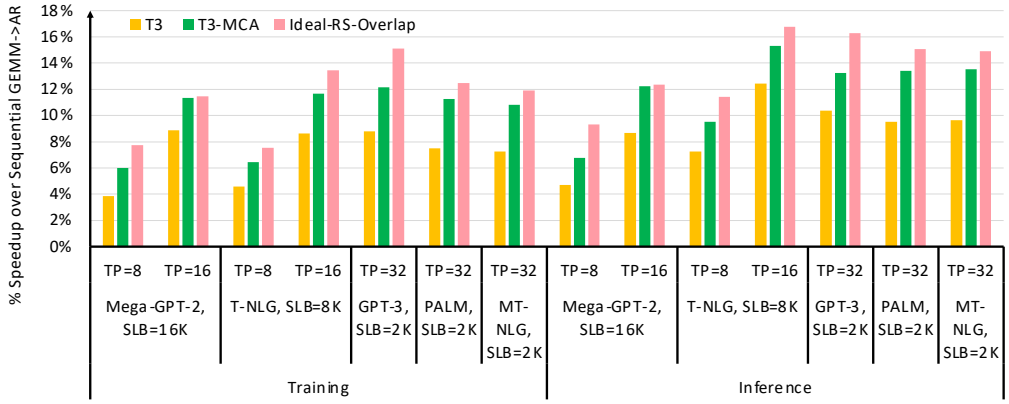


Figure 8.17: End-to-end model speedups.

due to fewer overall RS steps. Conversely, GEMM read caching impact is higher with a larger TP degree; larger TP/slicing leads to smaller, more LLC-amenable GEMMs. Overall, RS's reads reduce by 2.4× geomean (2.5× for TP=8, 2.2× for TP=16), both GEMM's and RS's writes reduce by 10% geomean (14% for TP=8, 7% for TP=16), and finally GEMM's reads decrease by 1.6× geomean (1.2× for TP=8, 2× for TP=16).

8.5.3 End-to-end Model Speedups

As shown in Figure 8.17, T3 and T3-MCA speed up model training by a maximum of 9% and 12%, and geomean of 7% and 10%, respectively. Benefits are higher at larger TPs due to the overall higher proportion of the sliced sub-layers requiring AR (Section 8.2). Similarly, prompt processing and/or large input token processing during inference is also sped up by a maximum of 12% and 15%, and geomean of 9% and 12% with T3 and T3-MCA, respectively. Inference speedups are better due to the overall higher proportion of sliced sub-layers resulting from no backprop compute. Finally, the MLPerfv1.1 implementation we evaluate does not include a key fusion optimization [59], which makes the non-sliced attention operations a significant 40-45% of execution time. Thus, we expect T3's and T3-MCA's

benefits to be much higher for newer MLPerf implementations.

8.5.4 Impact on Larger Transformers

We also evaluate larger Transformers with higher TP degrees. Similar to the smaller models, layer-level speedups are high; max 35% and geomean of 29% for GPT-3 (175B parameters), PALM (530B parameters), and MT-NLG (540B parameters). As shown in Figure 8.17, these lead to up to 12% and 14% end-to-end speedup in their training and prompt phase of inference, respectively. Thus, T3-MCA also effectively speeds up larger models.

8.6 Discussion

8.6.1 Other Collectives Implementation & Types

T3 supports other collectives and implementations via the configuration of GEMM's output address space (Section 8.3.4).

Other implementations: Collectives can have multiple implementations optimized for different topologies. We focus on ring since it is commonly used in intra-node setups where tensor slicing is employed [109]. T3 can also support the *direct* RS implementation in a fully-connected topology. At every GEMM stage, the output from each device is scattered across the remaining devices using dedicated links and reduced at the destination. This is accomplished by changing the configuration in Figure 8.10 to slice each GEMM stage output and remote_map each slice to a remote device. In this case T3 eliminates memory accesses by the collective as it is completely orchestrated using GEMM stores. Similarly, it can also support other, inter-node implementations via appropriate programming of the track & trigger mechanism.

Other types: Similarly, T3 also supports other collectives. A ring/direct all-gather (AG) reuses ring-RS's configuration and setup, except the GEMMs and DMA transfers do not update memory locations. Similar to AG, T3 can also support an all-to-all collective where devices exchange sub-arrays, except here the remote/dma_mapped GEMM output is not written to local memory.

8.6.2 Other Distributed Techniques

Although we focus on communication in tensor-parallel (TP) setups, T3 is also applicable in other distributed setups where a producer's output is communicated via a collective.

Expert Parallelism: Similar to TP, expert parallelism in Mixture-of-experts (MoEs) [69, 228] require serialized all-to-all communication which can be fused with T3 as discussed in Section 8.6.1.

Data & Pipeline Parallelism: T3 also applies to data-parallel and pipeline-parallel setups which require RS, and peer-to-peer transfers, respectively. While T3's overlapping benefits may not provide additional benefits in such cases (these communications can be overlapped with other independent kernels), T3's NMC and MCA techniques can help reduce memory bandwidth contention in these cases as well.

TP with All-gather: T3 can be extended for distributed setups where the collective's output requires overlapping with a long-running consumer operation. This is required if the producer is short-running (e.g., TP which all-gather's activations). Overlapping collective-consumer pairs is similar in principle to overlapping producer-collective and requires similar tracking/triggering mechanisms. The Tracker would track "all-gathered-input→GEMM-WG" instead of "GEMM-WG→all-reduced-output". Moreover, instead of triggering a DMA, it would trigger a WG scheduling event (such as in Lustig & Martonosi [159]). This can be challenging since the

"all-gathered-input→GEMM-WG" mapping can be kernel implementation dependent. However, additional programming hints could overcome this.

8.6.3 Generative Inference

While we focus on the communication-heavy training and prompt phase of inference, T3 is also applicable in the generation phase of inference. Due to smaller input token counts (Section 2.2.2.2), these phases are bound by memory accesses of model weights and can benefit from the aggregate memory bandwidth of multiple devices that TP provides [28]. The resulting all-reduce of activations, while smaller than those in training and thus potentially latency-bound (due to small token counts), can still be overlapped and hidden with GEMM executions using T3.

8.6.4 Other Reduction Substrates

While T3 leverages NMC for atomic updates required in reduction-based collectives (e.g., RS, AR), it is not a requirement. Such updates could also be handled via system-wide atomics on uncached data without significant loss in performance. Similarly, T3 can also leverage switches for reductions as shown by prior works [130].

8.6.5 Future Hardware & Lower Precision

Since compute FLOPS have scaled much more than network link bandwidths across hardware generations [77, 188, 194, 200], communication will likely be a larger proportion of the end-to-end execution both in more recent systems than the one we evaluate and in the future. Similarly, lowering precision [166, 247] decreases compute time much more (quadratically) than communication (linearly). Thus, the benefits of hiding communication with techniques like T3 will also apply to other GPU configurations and datatypes besides 16b.

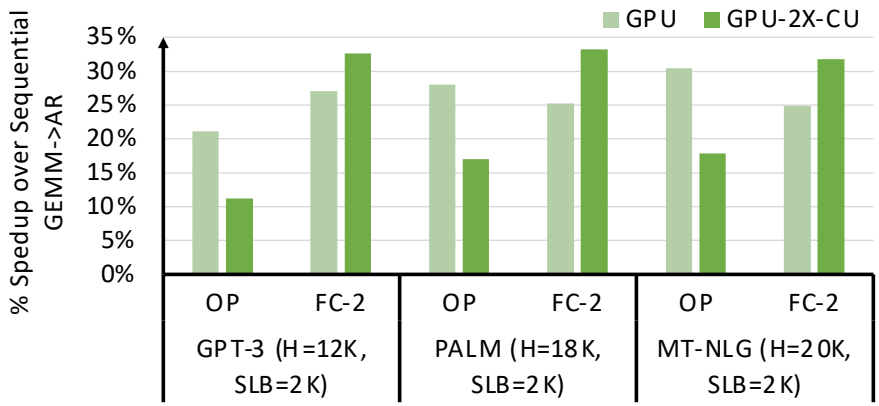


Figure 8.18: T3 on future hardware with 2× compute.

To evaluate T3's hiding capability in future systems, we study a system configuration where compute FLOPS scale more than network link bandwidth (2×), which we term GPU-2X-CU. While the scaling GPU FLOPs across generations largely result from more powerful CUs (larger/faster tensor processing), we simulate it by scaling the number of CUs and keeping the underlying network the same. This enables us to use the latest/validated GPU model and GEMM traces that Accel-Sim supports [124]. Figure 8.18 shows that for larger layers (FC-2) where compute time dominates, compute becomes faster with 2× CUs which lowers the compute:communication ratio across the models. This shortens the critical path and leads to larger benefits with overlapping compute and communication with T3. Conversely, for smaller layers (OP), where compute and communication are more balanced, faster compute exposes communication on critical path, lowering T3's benefits. Note that, for such scenarios, communication optimizations will be necessary [44, 253]. Nevertheless, the larger layers have a more prominent impact on overall execution and for these, T3's benefits only improve.

8.6.6 NMC for Following Operations

Collectives, specifically all-reduce in Transformers, are usually followed by other memory-intensive operations on all devices (e.g., parameter updates in DP [215] or residual/dropout layers in TP). These operations redundantly operate on the entire all-reduced array on each device. With T3, these following memory-intensive operations can be executed using NMC [215] on (reduced) sub-arrays of data before they are all-gathered/broadcasted to the remaining devices, thus reducing redundancy, and further accelerating distributed Transformer models.

8.6.7 Other GEMM Implementations

T3 focuses on the most common tiled GEMM implementation with a WG/WF responsible to generate an entire tile/sub-tile of data. However, T3 can support other implementations, such as split-K [197]. A split-K implementation slices work in the accumulation or K dimension, such that multiple WGs are responsible for a single tile, each generating a partial tile that is reduced after. Split-K increases parallelism when the output size (MxN) is small but the K dimension is large. However, tensor-sliced GEMMs, which require AR, have large output sizes and small K dimensions. Naively, T3 with a split-K implementation (with more than one update to an element) will cause multiple local and remote updates per memory location. To prevent this, T3 can use the kernel packets' tile-size metadata to deduce split-k degree (=(#WGs * tile-size)/(M*N)), i.e., the number of updates per element. The virtual addresses in the tracker (Section 8.3.2.1) can be used to determine WFs/WGs/tracker entries that update the same tile, allowing the tracker to trigger remote DMA only after all updates to the tile are complete.

8.6.8 Multi-node Setups

Tensor-parallelism, with serialized communication is usually employed within a node, which generally has high-speed homogeneous links. However, T3 can also be applied to serialized communication in inter-node setups with slower and often heterogeneous links. Consequently, communication costs can be much larger than GEMM executions, potentially limiting the benefits from fine-grained overlap: once the computation is completely overlapped, the remaining communication costs will be exposed [278]. Nevertheless, T3 can still provide benefits from hiding the GEMM execution cost as much as possible.

8.6.9 Communication in High Performance Computing (HPC)

There have been several works to optimize for inter-node communication in supercomputers for HPC applications. Similar to our work, their goal has been to overlap communication with compute, while also reducing the overhead to initiate communication via NIC. The closest to T3 are works such as GPUDirect Async [7] and GPU-TN [139] in which the GPU initiates communication and are classified as GPU native networking [139]. While the former enables initiation at kernel boundaries and thus only enables coarse-grained overlap with compute, the latter enables intra-kernel communication and finer-grained overlap. However, GPU-TN requires kernel changes to distinguish stores which initiate communication as well as to tag them with an identifier. T3 avoids this by creating appropriate mappings of the output array and leverages the workgroup / wavefront IDs for tracking. This in turn avoids the modification of hundreds of kernels in BLAS libraries. Additionally, it leverages NMC for reductions required by communication. Thus works such GPU-TN for NIC-based communication can be extended with ideas from T3.

Approach / Features	GPU Support	Transparent Comm.	Overlap Comm.	Reduce Contention	No Addl. Accelerator	Topology-independent
In-switch [130]	✓	X	X	✓	X	X
ACE [235]	✓	X	X	✓	X	X
CoCoNet [107]	✓	X	✓	X	✓	✓
Overlap Google [278]	X	X	✓	X	✓	✓
T3-MCA	✓	✓	✓	✓	✓	✓

Table 8.3: Comparing T3-MCA to prior work.

8.7 Related Work

Table 8.3 compares T3-MCA with prior works across several key metrics. Some prior work has designed *in-switch collectives* to speed up communication by up to 2× [130]. However, this cannot eliminate serialized communication from the critical path. Furthermore, they are topology-dependent, requiring switches. Enabling fine-grained overlap of compute and communication is essential to effectively hide the cost of communication. Existing attempts to do this, like *CocoNet*[107] and *Google Decomposition* [278], have limitations. *Google Decomposition* requires changes to matrix multiplication (GEMMs) kernels which can be disruptive to GPU software infrastructure (Section 8.2.1).

Furthermore, both approaches can suffer from hardware resource contention between compute and communication (Section 8.2.2). Works that reduce contention only address coarse-grained overlap of compute and communication in cases like DP, lacking support for fine-grained overlap in serialized collectives [235]. Moreover, they rely on dedicated accelerators. Other recent work fuses communication within the computation kernel to enable fine-grained overlap, such that a GPU kernel performs both computation and dependent communication at the WG level [221]. However, this requires explicit changes to the compute kernels and is not readily applicable for collectives involving simple arithmetic operation such as reduce-scatter – which will still be limited by inter-GPU synchronization.

Finally, other work like Syndicate increases coarse-grained overlap opportunities and efficiency in distributed training. However, Syndicate cannot hide serialized communication [163]. *T3-MCA overcomes these shortcomings and achieves a transparent overlap of serialized communication with compute, while minimizing resource contention.*

8.8 Chapter Summary

Transformer models increasingly rely on distributed techniques, requiring communication between multiple devices. As we showed in Chapter 5, this communication can limit scaling efficiency, especially for techniques like TP which serializes communication with model execution. While a fine-grained overlap of the serialized communication with its producer computation can help hide the cost, realizing it with GPUs is challenging due to software complexities and resource contention between compute and communication. To overcome this, in this chapter, we proposed T3, which transparently and efficiently fuses and overlaps serialized inter-device communication with the producer's compute. It orchestrates communication on the producer's stores by configuring the producer's output address space mapping and using a programmable track and trigger mechanism in hardware. This reduces application impact and also eliminates contention for GPU compute resources. T3 additionally uses near-memory computing and a memory-controller arbitration policy to reduce memory-bandwidth contention. Overall, T3 improved performance by 30% geomean (max 47%) and reduced data movement by 22% geomean (max 36%) over state-of-the-art approaches. Moreover, T3's benefits hold as models and hardware scale.

9 CONCLUSION & FUTURE WORK

In contemporary computing, DNNs, particularly sequence-based networks, have emerged as predominant workloads on hardware devices. The size of these networks, the scale of the datasets they are trained on, and the range of tasks they are applied to have grown exponentially. Consequently, the computational demand for training and deploying these models has also surged. For instance, some of the largest NLP models today require over a month for training on approximately 4500 GPUs. While the common strategy has been to increase the computational capabilities of GPUs and/or to increase the number of GPUs, the fundamental question is: do these application indeed utilize all these resources? In this dissertation, we answer this question through detailed characterization and identify opportunities to enhance their execution efficiency.

In Chapters 3, 4 and 5 of the dissertation, we profiled and characterized contemporary sequence-based models to identify hardware inefficiencies in their execution. This analysis specifically focused on recurrent-based and state-of-the-art Transformer models on GPUs. Given the input length-dependent, heterogeneous, training iterations of these sequence-based models and the thousands of training iterations resulting from large datasets, we devised SeqPoint for their quick yet accurate profiling. Moreover, recognizing the ever-evolving and increasingly large nature of these models, we developed strategies to project their future behaviors. Their characterizations exposed three primary inefficient phases in sequence-based models. These include memory-bound weight update algorithms, underutilization of concurrent matrix-multiplication (GEMM) operations, and the idling of compute resources during serialized inter-device communication phases.

In Chapters 6, 7 and 8 of the dissertation, we introduced cross-stack approaches aimed at mitigating these identified inefficiencies. First, we

demonstrated the efficacy of selectively offloading memory-bound weight update algorithms to emerging compute-enhanced memories. This strategy significantly enhanced performance while concurrently reducing data movement and also making GPU compute units available for other operations. Second, we addressed sub-optimal concurrent GEMM executions on GPUs via GOLDYLOC. GOLDYLOC uses runtime information to choose kernel implementations optimized for shared resource environments as well as to dynamically control the amount of concurrency to exploit. This improved concurrent GEMM throughput and execution efficiency. Finally, we tackled the challenge of serialized communication phases with T3 which interleaves communication with their preceding producer operations in a fine-grained manner. T3 uses a hardware track and trigger mechanism to mitigate software complexities as well as leverages near-memory computing and DMA engines to minimize resource contention resulting from the overlap. Overall, these optimizations highlight the need for information flow between different layers of compute abstractions (application, libraries, runtime, and hardware) to improve GPU resource utilization.

In this chapter, we first summarize each part of this dissertation (Section 9.1) and present the various lessons we learned through the course of this dissertation work (Section 9.2). We then discuss some directions for future work (Section 9.3).

9.1 Summary

9.1.1 Analysis of Sequence-based Models on GPUs

9.1.1.1 SeqPoint: Identifying Representative Training Iterations to Profile

Profiling and characterizing entire training runs of sequence-based models (e.g., natural language models) is challenging given their hours-to-days native runs. We observed that prior works which characterize them are oblivious to the heterogeneity in training iterations and, as such, are ill-equipped to create small, representative training runs that faithfully summarize entire training phases.

To address this, we first observed that input SL is a key factor that dictates the heterogeneity in training iterations of sequence-based models. Then, we designed a new scheme that clusters unique SLs and selects representative points in each cluster, termed a SeqPoint. We showed our identified SeqPoints are representative of the entire training run with low error. Finally, this mechanism reduced profiled training iterations by up to two orders of magnitude for state-of-the-art, end-to-end sequence-based models. Overall, this mechanism not only makes profiling and characterization of these models training tractable but also paves the way for their network-level simulations.

9.1.1.2 Characterization of Transformers Models

NLP uses increasingly large Transformer models that tackle challenging problems and are driving the requirements of future systems. To this end, we studied the computationally and time-intensive training phase of NLP models and identified how its algorithmic behavior can guide future accelerator design. We focused on BERT (Bi-directional Encoder Representations from Transformer), one of the most popular Transformer-based

NLP models, and identified key operations which are worthy of attention in accelerator design. In particular, we focused on the manifestation, size, and arithmetic behavior of these operations which remain constant irrespective of hardware choice.

Our results showed that although computations which manifest as matrix multiplications dominate BERT's execution, they have considerable heterogeneity and may not always fully utilize accelerators. Furthermore, we characterized memory-intensive computations which also feature prominently in BERT but have received less attention, especially the optimizer algorithms used to update model weights. To capture future Transformer trends, we vary key model hyperparameters and showed implications of these behaviors as networks and their inputs get larger. Moreover, we study the impact of key training techniques like distributed training, checkpointing, and mixed-precision training. Overall, this work identified several inefficiencies and opportunities to accelerate Transformer-based models, which we address in subsequent chapters.

9.1.1.3 Characterizing the Scaling Communication in Multi-GPU Transformers Setups

Scaling of Transformers models and their datasets has necessitated very large-scale distributed setups, which raises the key question: *how will compute vs. communication* (*Comp-vs.-Comm*) *scale as models scale and hardware evolves?* We conducted a multi-axial (algorithmic, experimental, hardware evolution) analysis of Comp-vs.-Comm scaling for Transformer models. Our system-agnostic, algorithmic analysis highlighted that while compute has enjoyed an edge over communication, future model and hardware trends are likely to make communication dominant soon. We also empirically studied Comp-vs.-Comm for future Transformer models as hardware evolves. By extracting specific regions of interest and modeling future operator runtimes, we enabled the study of hundreds of future Transform-

ers/hardware scenarios with 2100× less profiling costs. These experiments validated that communication will play an increasingly large role (40-75%) in a distributed training setup as models scale. Overall, our multi-axial analysis shows the need for effective scaling of communication capabilities, and discusses how our analysis influences some promising techniques and technologies.

9.1.2 Improving Sequence-based Models' Efficiency on GPUs

9.1.2.1 Offloading Optimizer Updates to Compute Units Near Memory

Our characterization revealed how memory-bound gradient descent updates of billions of Transformer parameters can under-utilize modern accelerators like GPU. To overcome this, we ***offloaded updates to near-memory compute units***. Mapping a sequence of operations to memory required few expensive synchronizations with GPU compute units, and provided increased data access bandwidth along with concurrency of multiple DRAM banks. Thus, it accelerated weight updates by 3.8× for a popular Transformer, BERT. Finally, it considerably reduced (~13×) expensive data movement between DRAM and GPU compute units.

9.1.2.2 Globally Optimized Libraries & Scheduling Logic for Concurrent GEMMs

Modern accelerators like GPUs are increasingly executing independent operations concurrently to improve the device's compute utilization. However, effectively harnessing it on GPUs for important primitives such as GEMMs remains challenging. Although modern GPUs have significant hardware and software support for GEMMs, their kernel implementations typically assume each kernel executes in *isolation* and can utilize all GPU resources. This approach is highly efficient when kernels execute in iso-

lation, but causes significant resource contention when kernels execute concurrently. Moreover, current approaches often only *statically* expose and control parallelism within an application, without considering the *dynamic* execution environment (e.g., varying input size) – often exacerbating contention. These issues limit performance benefits from concurrently executing independent operations.

Accordingly, we proposed GOLDYLOC which considers both the *globally optimized* kernel implementations which are aware of the shared GPU resources during concurrent executions. Further we introduced a *lightweight dynamic logic* to schedule only performant concurrent operations. Overall, GOLDYLOC improved performance of concurrent GEMMs on real hardware by up to 2.5× (43% geomean per workload) over sequential execution and up to 2× (18% geomean per workload) over static, isolated parallelism in GPUs.

9.1.2.3 Transparent Track & Trigger for Fine-grained Overlap of Compute & Communication

Transformer models increasingly rely on distributed techniques, requiring communication between multiple devices. This communication can limit scaling efficiency, especially for techniques like Tensor Parallelism (TP) which serialize communication with model execution. While a fine-grained overlap of the serialized communication with its producer computation can help hide the cost, realizing it with GPUs is challenging due to software complexities and resource contention between compute and communication. To overcome this, we proposed T3, which transparently and efficiently fuses and overlaps serialized inter-device communication with the producer's compute. It orchestrates communication on the producer's stores by configuring the producer's output address space mapping and using a programmable track and trigger mechanism in hardware. This reduces application impact and also eliminates contention for GPU compute

resources. T3 additionally uses near-memory computing and a memory-controller arbitration policy to reduce memory-bandwidth contention. Overall, T3 improved performance by 30% geomean (max 47%) and reduces data movement by 22% geomean (max 36%) over state-of-the-art approaches. Moreover, T3's benefits hold as models and hardware scale.

9.2 Reflection

In this section, we present our observations and lessons we learned while working on this dissertation.

Characterizing new workloads is important. Characterization of new workloads, as we did with the then emerging sequence-based networks, is important as each new model architecture comes with unique opportunities. As an example, CNNs, which were well studied, are constituted of several GEMMs and weight updates. Although sequence-based models (RNNs and Transformers) also consist of similar primitives, our characterization of these models revealed unique concurrency and memory-bandwidth challenges in GPUs and opportunities to improve their performance as shown in Chapters 7 and 6. Such characterization can also help find pertinent primitives that can be independently simulated on hardware simulators as shown in Chapter 8, rather than executing end-to-end models which is impractical. Finally, detailed characterization can also help provide feedback to application/algorithms/software engineers and enable a stronger ML algorithm and system co-design across the computing stack. For example, in Chapter 5 we identified that scaling the hyperparameter sequence length (SL) has a smaller impact in terms of added communication overheads than the scaling of layer width (hidden dimension or H).

Algorithmic understanding of applications helps. Algorithmic understanding of applications is also important as it can help avoid exhaustive profiling of applications. Understanding the impact of hyperparameters helped us identify as well as explain some of the trends in Chapters 3, 4, and 5. With a fixed model architecture, and number of kernels, these can also help project trends for other models as we showed in Chapters 4 and 5. However, this can be challenging since the behavior of certain operations can change with input. For example many GEMMs' efficiency improve with scaling input size until they saturates. Thus, while profiling is important to accurately measure their runtime, once they reach the achievable peak efficiency / FLOPS, they become predictable with high accuracy. In these scenarios, using a combination of both hardware and analytical model can be sufficient as we show in Chapter 5.

Need for information flow between different layers of system abstractions. Abstractions are important as they help divide and conquer the development of complex systems. However, they can leave considerable system performance on the table. We observe this as one of the common cause across several of the inefficiencies discussed in the this dissertation. In Chapter 6 we showed how offloading entire end-to-end applications to a single accelerator (e.g., GPUs) for programmability rather than carefully identifying individual function properties and mapping them to appropriate architectures (e.g., near-memory compute units) can not only be power inefficient and have excessive data movement, but also limit performance. Similarly in Chapters 7 and 8 we show that although dedicated libraries for key ML primitives (e.g., cuBLAS/rocBLAS for GEMM, NCCL/RCCL for collectives) which are independently optimized for different hardware architectures are performant, they can be inefficient considering global and dynamic execution environments. Information about globally shared resources during execution helped improved concurrent GEMM perfor-

mance in Chapter 7. Similarly, we showed in Chapter 8 that information about preceding and succeeding operations in applications can enable additional fusion and parallelization opportunities which are otherwise lost in abstractions. Thus, information flow between different levels of abstractions – application, libraries, hardware – is needed to fully utilize hardware resources.

Need to better use existing, powerful hardware Similar to the considerable push towards building better software stack for domain-specific accelerators, a similar push for better GPU hardware-software integration is needed to maximally use their compute capabilities. While GPUs have been built with high memory bandwidth technology, applications at the best only use 70-80% of the provided bandwidth. Thus is due to inefficient software implementations, shared bus architecture, and other overheads. Similarly, GPUs provide considerably high FLOPS capacity which only scales every generation, especially with the adoption of chiplet-based architectures. However, application utilizations of these FLOPS is low even when there is not a scarcity of work due to inefficient runtimes. Finally, while scaling inter-device bandwidth considerably speeds up communication, they remain idle for a significant amount of time while the GPU computes. Thus, mechanisms for fine-grained compute-communication, and even leveraging idle links/GPUs to distribute work dynamically can help. Overall, these findings underscore the importance of improving the utilization of our existing, powerful, hardware.

Power & energy implications: The focus of this dissertation was to improve GPU resource efficiency and thus performance of applications. However, energy and power are also critical aspects when it comes to system design. While not evaluated, each of our proposals in Chapters 6, 7 and 8 uniquely impact system power and energy. Offloading weight updates

to NMC units in Chapter 6 reduced how much data is transferred across the memory interface as well as the memory requests sent from the GPU to memory. This has the potential to reduce both GPU power and energy consumption. GOLDYLOC in Chapter 7 is also energy efficient as it improved overall GPU throughput via concurrency. However, it can potentially increase the GPU's dynamic power. Considering the power limits of the GPU when choosing the amount of concurrency to exploit can be a potential future direction for GOLDYLOC. Finally, T3 can also considerably improve energy efficiency as it overlaps communication with compute (limiting idle compute units and interconnects) and reduces data movement to memory. However, similar to GOLDYLOC, it has the potential to increase dynamic power consumption by keeping GPU compute units, memory (concurrent communication-related traffic) as well as interconnects active at the same time.

Focus on key primitives increases applicability beyond Sequence-based models: While the motivation for this dissertation has been to optimize for sequence-based models on GPUs, these proposals can also be applied to other DNNs as well as high-performance computing (HPC) applications. The three key primitives that are the focus of this dissertation – optimizer/weight updates, GEMMs, and communication collectives – are fundamental to deep learning. As a result, these primitives are present in most other DNN models, such as CNNs and recommendation systems. Thus, all the proposals – NMC offloading, GOLDYLOC, and T3 can be applied to other DNNs, although with varying benefits. Similarly, these optimizations would remain applicable even with the extremely dynamic field of ML, with constantly evolving model architectures. Finally, communication is extremely important even in large-scale HPC applications with many scenarios having a similar producer-consumer relationship we leveraged in T3 [139, 141]. While there has been considerable work to re-

duce overheads of GPU-initiated communication, especially in inter-node scenarios, key ideas of T3 can be applied to HPC applications as well.

9.3 Future Work

In this section, we outline directions in which our work could be extended in the future.

Characterization:

- While this dissertation focused largely on dense models, there are several flavors of these models that have been introduced since, including sparse attention and mixture-of-experts (MoEs), to reduce their computational complexity. Understanding the impact of these on current hardware and whether GPU systems are able to leverage this introduced sparsity will be a useful next step, especially given the limited hardware support for sparsity on GPUs. Similarly, understanding the sparsity support on hardware to design better sparse ML algorithms is another avenue for a stronger ML-systems co-design.
- Similarly, the recent increase in interest and wide-spread adoption of generative inference both across application domains and hardware substrates (datacenter, personal laptops, edge devices) also demands that we characterize them.

Near-memory Computing (NMC):

- In Chapter 8 we showed that concurrent GPU and NMC operations which share memory bandwidth can cause slowdowns and reduce the efficacy of overlap. A more exhaustive study of the interference

including when NMC commands are generated by GPU compute units, the different types of NMC commands with varying latencies, as well as the different types of NMC command orchestrations (broadcast vs fine-grained) is worth pursuing.

- Similarly, extending this to other substrates such as CPUs and accelerators which also stand to benefit from NMC.
- Many memory-bound phases are interspersed throughput training execution; gradient accumulation, weight updates, quantization [101]. As mentioned in Chapter 6, finding contiguous portions of applications reduce GPU-NMC synchronization overheads. Thus designing end-to-end ML algorithms to enable better use of such hardware support can considerably improve performance.
- Finally, while training efficiency thrives on large input batching, the latter is not favorable during inference/deployment as it can increase execution latency and thus the response time of applications. Small or no batching results in matrix-vector (GEMV) or skinny GEMMs which are usually memory-bound. Finding ways to leverage NMC during inference can alleviate some performance challenges during inference.

GPU Concurrency:

- While we focus on the dominant GEMM operations in Chapter 7, GOLDYLOC can be extended to other operators such as activation functions, reduction, and other prominent element-wise operations. The latter, since memory-bound, may provide additional challenges and opportunities to improve performance (concurrent compute-bound GEMM with memory-bound activation functions).
- Sparsity, which reduces the amount of data read as well as compute performed, further frees resources for and provides opportunity for

concurrency. Thus applying GOLDYLOC to sparse computations can unlock additional performance benefits.

- Finally, GOLDYLOC leverages existing tuning infrastructure to find optimized concurrency-aware kernels optimized for global resources. While we present some potential benefits of using techniques such K-Nearest Neighbor to reduce tuning overheads, a more strategic use of ML techniques can beneficial to automatically find appropriate kernels not only for shared, concurrent scenarios but also to augment the baseline isolated performance tuning.

Communication:

- Extending T3 to inter-node setups without shared memory support can add additional overheads of address translations. Furthermore, inter-node setups can have complex topologies and heterogeneous links, requiring smarter orchestration of the compute kernels and communication.

- Finally, similar to how the fine-grained interleaving of GEMMs and communication unlocked performance benefits from better utilizing compute and interconnect resources, other producer-consumer pairs (such as a compute-bound GEMM followed by memory-bound element-wise operation) can also benefit from the same [135, 212].

BIBLIOGRAPHY

[1] Ahmad Abdelfattah, Marc Baboulin, Veselin Dobrev, Jack Dongarra, Christopher Earl, Joel Falcou, Azzam Haidar, Ian Karlin, Tz Kolev, Ian Masliah, and Stanimire Tomov. High-performance tensor contractions for GPUs. *Procedia Computer Science*, 80:108–118, 2016.

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Performance, design, and autotuning of batched GEMM for GPUs. In *International Conference on High Performance Computing*, pages 21–38. Springer, 2016.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-yeon Wei, and David Brooks. Fathom: Reference Workloads for Modern Deep Learning Methods. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 1–10, Sept 2016.

[5] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. The Case for GPGPU Spatial Multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*, HPCA, pages 1–12, Washington, DC, USA, 2012. IEEE, IEEE Computer Society.

[6] Shaizeen Aga, Nuwan Jayasena, and Mike Ignatowski. Co-ML: A Case for Collaborative ML Acceleration Using near-Data Processing. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 506–517, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Elena Agostini, Davide Rossetti, and Sreeram Potluri. GPUDirect Async: Exploring GPU synchronous communication techniques for InfiniBand clusters. *Journal of Parallel and Distributed Computing*, 114:28–45, 2018.

[8] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. Lazy Release Consistency for GPUs. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 26:1–26:13, New York, NY, USA, 2016. Association for Computing Machinery.

[9] AMD. AMD INSTINCT™ MI50 ACCELERATOR. `https://www.amd.com/en/products/professional-graphics/instinct-mi50`, 2018.

[10] AMD. AMD's ROCm Communication Collectives Library. "`https://github.com/ROCmSoftwarePlatform/rccl/wiki`", 2018.

[11] AMD. HIP: Heterogeneous-computing Interface for Portability, 2018.

[12] AMD. AMD ROCm Profiler. `\rocmdocs.amd.com/en/latest/ROCm_Tools/ROCm-Tools.html`, 2019.

[13] AMD. AMD Ryzen™ Threadripper 2950X Processor. `https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-2950x`, 2019.

[14] AMD. AMD's BLAS Library. "`https://github.com/ROCmSoftwarePlatform/rocBLAS`", 2019.

[15] AMD. AMD's Machine Intelligence Library. `https://github.com/ROCmSoftwarePlatform/MIOpen`, 2019.

[16] AMD. Radeon compute profiler (rcp). "`https://gpuopen.com/compute-product/radeon-compute-profiler-rcp/`", 2019.

[17] AMD. Radeon™ Vega Frontier Edition Graphics. "`https://www.amd.com/en/graphics/workstations-radeon-pro-vega-frontier-edition`", 2019.

[18] AMD. ROCm, a New Era in Open GPU Computing. "https://rocm.github.io/", 2019.

[19] AMD. ROCm/HIP enabled Tensorflow. "https://github.com/soyers/hiptensorflow", 2019.

[20] AMD. AMD CDNA ARCHITECTURE. https://amd.com/system/files/documents/amd-cdna-whitepaper.pdf, 2020.

[21] AMD. AMD Instinct™ MI100 Accelerator. "https://www.amd.com/en/products/server-accelerators/instinct-mi100", 2020.

[22] AMD. AMD MxGPU and VMware. https://drivers.amd.com/relnotes/amd_mxgpu_deploymentguide_vmware.pdf, 2020.

[23] AMD. AMD's tool for creating a benchmark-driven backend library for GEMMs. "https://github.com/ROCmSoftwarePlatform/Tensile/", 2020.

[24] AMD. AMD HSA Code Object Format. "https://rocmdocs.amd.com/en/latest/ROCm_Compiler_SDK/ROCm-Codeobj-format.html", 2021.

[25] AMD. AMD INSTINCT™ MI210 ACCELERATOR. https://www.amd.com/en/products/server-accelerators/amd-instinct-mi210, 2022.

[26] AMD. AMD INSTINCT™ MI300X PLATFORM™. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/data-sheets/amd-instinct-mi300x-platform-data-sheet.pdf, 2023.

[27] AMD. Distributed Services Card (DSC). https://www.amd.com/system/files/documents/pensando-dsc-200-product-brief.pdf, 2023.

[28] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 1–15. IEEE, 2022.

[29] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Chong Wang, Yi Wang, Zhiqian Wang, Bo Xiao, Yan Xie, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *Proceedings of the 33nd International Conference on Machine Learning*, ICML, pages 173–182, 2016.

[30] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. CUTLASS: Fast linear algebra in CUDA C++. `https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/`, 2017.

[31] Jeremy Appleyard, Tomas Kocisky, and Phil Blunsom. Optimizing Performance of Recurrent Neural Networks on GPUs, 2016.

[32] Cesar Avalos Baddouh, Mahmoud Khairy, Roland N Green, Mathias Payer, and Timothy G Rogers. Principal Kernel Analysis: A tractable methodology to simulate scaled GPU workloads. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 724–737, 2021.

[33] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization. *CoRR*, 1607.06450, 2016.

[34] Dzmitry Bahdanau, KyungHyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of the Third International Conference on Learning Representation*, ICLR, 2015.

[35] Baidu. Baidu All-Reduce. `https://github.com/baidu-research/baidu-allreduce`, 2017.

[36] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 163–174, April 2009.

[37] Yuhui Bao, Yifan Sun, Zlatan Feric, Michael Tian Shen, Micah Weston, José L. Abellán, Trinayan Baruah, John Kim, Ajay Joshi, and David Kaeli. NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, page 333–345, New York, NY, USA, 2023. Association for Computing Machinery.

[38] Nathan Benaich and Ian Hogarth. State of AI Report 2022. `https://www.stateof.ai/`, 2022.

[39] James Betker, Gabriel Goh, Li Jing, Tim Brooks, Jianfeng Wang, Linjie Li, Long Ouyang, Juntang Zhuang, Joyce Lee, Yufei Guo, Wesam Manassra, Prafulla Dhariwal, Casey Chu, Yunxin Jiao, and Aditya Ramesh. Improving Image Generation with Better Captions. *Computer Science. https://cdn. openai. com/papers/dall-e-3. pdf*, 2(3):8, 2023.

[40] Ondej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurelie Neveol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, and Karin Verspoor-Marcos Zampieri. Findings of the 2016 Conference on Machine Translation. In *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*, pages 131–198, 2016.

[41] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33 of *NeurIPS*, pages 1877–1901, Red Hook, NY, USA, 2020. Curran Associates, Inc.

[42] Bobby R. Bruce, Ayaz Akram, Hoa Nguyen, Kyle Roarty, Mahyar Samani, Marjan Fariborz, Trivikram Reddy, Matthew D. Sinclair, and Jason Lowe-Power. Enabling Reproducible and Agile Full-System Simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, 2021.

[43] Business Wire. Samsung Electronics Introduces New High Bandwidth Memory Technology Tailored to Data Centers, Graphic Applications, and AI. `https://www.businesswire.com/news/home/20190319005767/en/Samsung-Electronics-Introduces-New-High-Bandwidth-Memory-Technology-Tailored-to-Data-Centers-Graphic-Applications-and-AI`, 2019.

[44] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing Optimal Collective Algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP, pages 62–75, 2021.

[45] Trevor E Carlson, Wim Heirman, Kenzo Van Craeynest, and Lieven Eeckhout. BarrierPoint: Sampled Simulation of Multi-threaded Applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12. IEEE, 2014.

[46] Raymond J Carroll and Shane Pederson. On Robustness in the Logistic Regression Model. *Journal of the Royal Statistical Society: Series B (Methodological)*, 55(3):693–706, 1993.

[47] Mauro Cettolo, Niehues Jan, Stüker Sebastian, Luisa Bentivogli, Rol dano Cattoni, and Marcello Federico. The IWSLT 2015 Evaluation Campaign. In *International Workshop on Spoken Language Translation*, 2015.

[48] Niladrish Chatterjee, Mike O'Connor, Donghyuk Lee, Daniel R Johnson, Stephen W Keckler, Minsoo Rhu, and William J Dally. Architecting an Energy-Efficient DRAM System for GPUs. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA, pages 73–84, Washington, DC, USA, 2017. IEEE, IEEE Computer Society.

[49] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 17–32, New York, NY, USA, 2017. ACM.

[50] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 681–696, New York, NY, USA, 2016. ACM.

[51] Benjamin Y. Cho, Jeageun Jung, and Mattan Erez. Accelerating Bandwidth-Bound Deep Learning Inference with Main-Memory Accelerators. *CoRR*, 2012.00158, 2020.

[52] Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, EMNLP, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[53] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. LazyBatching: An SLA-aware Batching System for Cloud Machine Learning Inference. In *HPCA*, HPCA, pages 493–506, Los Alamitos, CA, USA, 2020. IEEE Computer Society.

[54] Yujeong Choi and Minsoo Rhu. PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units. In *26th IEEE International Symposium on High Performance Computer Architecture*, HPCA, pages 220–233, 2020.

[55] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[56] Gonçalo M. Correia, Vlad Niculae, and André F. T. Martins. Adaptively Sparse Transformers. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, EMNLP-IJCNP, 2019.

[57] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. MSCCLang: Microsoft Collective Communication Language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 502–514, 2023.

[58] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. *CoRR*, 1901.02860, 2019.

[59] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-efficient Exact Attention with IO-Awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[60] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent. In *ISCA*, ISCA, pages 561–574, New York, NY, USA, 2017. ACM.

[61] Dell Technologies. MLPerf™ v1.1 Inference on Virtualized and Multi-Instance GPUs. https://infohub.delltechnologies.com/p/mlperf-tm-v1-1-inference-on-virtualized-and-multi-instance-gpus/, 2022.

[62] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NAACL-HLT, pages 4171–4186. Association for Computational Linguistics, 2019.

[63] Dave Dice and Alex Kogan. Optimizing Inference Performance of Transformers on CPUs. *CoRR*, abs/2102.06621, 2021.

[64] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, 2021.

[65] Shraf Eassa and Sukru Burc Eryilmaz. The Full Stack Optimization Powering NVIDIA MLPerf Training v2.0 Performance. https://developer.nvidia.com/blog/boosting-mlperf-training-performance-with-full-stack-optimization/, 2022.

[66] Izzat El Hajj, Juan Gomez-Luna, Cheng Li, Li-Wen Chang, Dejan Milojicic, and Wen-mei Hwu. KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 1–12, 2016.

[67] G. A. Elliott, B. C. Ward, and J. H. Anderson. GPUSync: A Framework for Real-Time GPU Management. In *IEEE 34th Real-Time Systems Symposium*, RTSS, pages 33–44, Dec 2013.

[68] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. TurboTransformers: An Efficient GPU Serving System for Transformer Models. In *PPoPP*, PPoPP, page 389–402, New York, NY, USA, 2021. Association for Computing Machinery.

[69] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *CoRR*, abs/2101.03961, 2021.

[70] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. Mobius: Fine Tuning Large-Scale Models on Commodity GPU Servers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 489–501, 2023.

[71] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Ludundefinedk Matyska. Optimizing CUDA Code by Kernel Fusion: Application on BLAS. *The Journal of Supercomputing*, 71(10):3934–3957, October 2015.

[72] Jan Fousek, Jiři Filipovič, and Matuš Madzin. Automatic Fusions of CUDA-GPU Kernels for Parallel Map. *SIGARCH Comput. Archit. News*, 39(4):98–99, December 2011.

[73] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A Configurable Cloud-scale DNN Processor for Real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA, pages 1–14, Piscataway, NJ, USA, 2018. IEEE Press.

[74] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 407–420, 2007.

[75] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low Latency RNN Inference with Cellular Batching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 31:1–31:15, New York, NY, USA, 2018. ACM.

[76] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional Sequence to Sequence Learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1243–1252. JMLR. org, 2017.

[77] Amir Gholami. AI and Memory Wall, 2021.

[78] Amir Gholami. Memory Footprint and FLOPs for SOTA Models in CV/NLP/Speech. "`https://github.com/amirgholami/ai_and_memory_wall`", 2021.

[79] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A Survey of Quantization Methods for Efficient Neural Network Inference, 2021.

[80] Andrew Gibiansky. Bringing HPC Techniques to Deep Learning. `https://github.com/baidu-research/baidu-allreduce`, 2017.

[81] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10, 2016.

[82] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995. IEEE, 2020.

[83] Anthony Gutierrez, Bradford M. Beckmann, Alexandru Dutu, Joseph Gross, Michael LeBeane, John Kalamatianos, Onur Kayiran, Matthew Poremba, Brandon Potter, Sooraj Puthoor, Matthew D. Sinclair, Michael Wyse, Jieming Yin, Xianwei Zhang, Akshay Jain, and Timothy Rogers. Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level. In *24th IEEE International Symposium on High Performance Computer Architecture*, HPCA, pages 608–619, Feb 2018.

[84] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H. Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W. Lee, and Deog-Kyoon Jeong. A$\hat{3}$: Accelerating Attention Mechanisms in Neural Networks with Approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341, 2020.

[85] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, 2017.

[86] Benjamin Hao and David Pearson. Instruction Scheduling and Global Register Allocation for SIMD Multiprocessors. In *2nd Int'l Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 81–86, 1995.

[87] Hany Hassan Awadalla, Anthony Aue, Chang Chen, Vishal Chowdhary, Jonathan Clark, Christian Federmann, Xuedong Huang, Marcin Junczys-Dowmunt, Will Lewis, Mu Li, Shujie Liu, Tie-Yan Liu, Renqian Luo, Arul Menezes, Tao Qin, Frank Seide, Xu Tan, Fei Tian, Lijun Wu, Shuangzhi Wu, Yingce Xia, Dongdong Zhang, Zhirui Zhang, and Ming Zhou. Achieving Human Parity on Automatic Chinese to English News Translation. *arXiv preprint arXiv:1803.05567*, March 2018.

[88] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385, 2015.

[89] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and TN Vijaykumar. Newton: A DRAM-maker's Accelerator-in-memory (AIM) Architecture for Machine Learning. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 372–385, Washington, DC, USA, 2020. IEEE, IEEE Computer Society.

[90] Yanzhang He, Tara N. Sainath, Rohit Prabhavalkar, Ian McGraw, Raziel Alvarez, Ding Zhao, David Rybach, Anjuli Kannan, Yonghui Wu, Ruoming Pang, Qiao Liang, Deepti Bhatia, Yuan Shangguan, Bo Li, Golan Pundak, Khe Chai Sim, Tom Bagby, Shuo yiin Chang, Kanishka Rao, and Alexander Gruenstein. Streaming end-to-end speech recognition for mobile devices. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6381–6385. IEEE, 2019.

[91] B.A. Hechtman, Shuai Che, D.R. Hower, Yingying Tian, B.M. Beckmann, M.D. Hill, S.K. Reinhardt, and D.A. Wood. QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs. In *20th International Symposium on High Performance Computer Architecture*, HPCA, pages 189–200, Washington, DC, USA, Feb 2014. IEEE Computer Society.

[92] Dan Hendrycks and Kevin Gimpel. Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units. *CoRR*, abs/1606.08415, 2016.

[93] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. *arXiv preprint arXiv:1503.02531*, 2015.

[94] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[95] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training Compute-Optimal Large Language Models. *arXiv preprint arXiv:2203.15556*, 2022.

[96] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. GRNN: Low-Latency and Scalable RNN Inference on GPUs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 41:1–41:16, New York, NY, USA, 2019. ACM.

[97] David W Hosmer, Trina Hosmer, Saskia Le Cessie, and Stanley Lemeshow. A Comparison of Goodness-of-fit Tests for the Logistic Regression Model. *Statistics in medicine*, 16(9):965–980, 1997.

[98] Jen-Cheng Huang, Lifeng Nai, Hyesoon Kim, and Hsien-Hsin S Lee. TBPoint: Reducing simulation time for large-scale GPGPU kernels. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 437–446. IEEE, 2014.

[99] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, volume 32 of *NeurIPS*, 2019.

[100] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. ARK: GPU-driven Code Execution for Distributed Deep Learning. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, pages 87–101, 2023.

[101] Mohamed Assem Ibrahim, Shaizeen Aga, Ada Li, Suchita Pati, and Mahzabeen Islam. Just-in-time Quantization with Processing-In-Memory for Efficient ML Training. *arXiv preprint arXiv:2311.05034*, 2023.

[102] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data Movement Is All You Need: A Case Study on Optimizing Transformers. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3 of *MLSys*, pages 711–732, 2021.

[103] James A. Jablin, Thomas B. Jablin, Onur Mutlu, and Maurice Herlihy. Warp-aware Trace Scheduling for GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (*PACT*), pages 163–174, 2014.

[104] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic Space-Time Scheduling for GPU Inference. *arXiv preprint arXiv:1901.00041*, 2018.

[105] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. CRUISE: Cache Replacement and Utility-Aware Scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems*, ASPLOS, pages 249–260, 2012.

[106] Charles Jamieson, Anushka Chandrashekar, Ian McDougall, and M. D. Sinclair. GAP: gem5 GPU Accuracy Profiler. In *4th gem5 Users' Workshop*, June 2022.

[107] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the Computation and Communication Abstraction Barrier in Distributed Machine Learning Workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 402–416, 2022.

[108] Abhinav Jangda, Saeed Maleki, Maryam Mehri Dehnavi, Madan Musuvathi, and Olli Saarikivi. A Framework for Fine-Grained Synchronization of Dependent GPU Kernels. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, pages 93–105, Los Alamitos, CA, USA, March 2024. IEEE Computer Society.

[109] Sylvain Jeaugey. How is tree reduction implemented? https://github.com/NVIDIA/nccl/issues/545#issuecomment-1006361565, 2022.

[110] JEDEC. High Bandwidth Memory DRAM (HBM1, HBM2). jedec.org/standards-documents/docs/jesd235a, 2019.

[111] Chetan Jhurani and Paul Mullowney. A GEMM interface and implementation on NVIDIA GPUs for multiple small matrices. *Journal of Parallel and Distributed Computing*, 75:133–140, 2015.

[112] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W Keckler, Mahmut T Kandemir, and Chita R Das. Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In *Proceedings of Workshop on General Purpose Processing using GPUs*, GPGPU, pages 1–8, 2014.

[113] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems*, ASPLOS, 2013.

[114] Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Exploiting Core Criticality for Enhanced GPU Performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, pages 351–363, 2016.

[115] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Nishant Patil, Sushma Prasad, Clifford Young, Zongwei Zhou, and David Patterson. Ten Lessons from Three Generations Shaped Google's TPUv4i. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA, 2021.

[116] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA, pages 1–12, New York, NY, USA, 2017. ACM.

[117] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural Machine Translation in Linear Time. *arXiv preprint arXiv:1610.10099*, 2016.

[118] Melanie Kambadur, Sunpyo Hong, Juan Cabral, Harish Patil, Chi-Keung Luk, Sohaib Sajid, and Martha A Kim. Fast Computational GPU Design with GT-Pin. In *2015 IEEE International Symposium on Workload Characterization*, pages 76–86. IEEE, 2015.

[119] Sheng-Chun Kao and Tushar Krishna. MAGMA: An Optimization Framework for Mapping Multiple DNNs on Multiple Accelerator Cores. In *28th IEEE International Symposium on High-Performance Computer Architecture*, HPCA, pages 814–830, 2022.

[120] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *USENIX Annual Technical Conference*, USENIX ATC, Portland, OR, June 2011. USENIX Association.

[121] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. RecNMP: Accelerating Personalized Recommendation with near-Memory Processing. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA, page 790–803. IEEE Press, 2020.

[122] Mahmoud Khairy, Akshay Jain, Tor M. Aamodt, and Timothy G. Rogers. Exploring Modern GPU Memory System Design Challenges through Accurate Modeling. *CoRR*, abs/1810.07269, 2018.

[123] Mahmoud Khairy, Vadim Nikiforov, David Nellans, and Timothy G. Rogers. Locality-Centric Data and Threadblock Management for Massive GPUs. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 1022–1036, 2020.

[124] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA, pages 473–486, 2020.

[125] Heesu Kim, Hanmin Park, Taehyun Kim, Kwanheum cho, Eojin Lee, Soojung Ryu, Hyuk-Jae Lee, Kiyoung Choi, and Jinho Lee. GradPIM: A Practical Processing-in-DRAM Architecture for Gradient Descent. In *HPCA*, HPCA, Washington, DC, USA, 2021. IEEE Computer Society.

[126] Jinsung Kim, Aravind Sukumaran-Rajam, Changwan Hong, Ajay Panyala, Rohit Kumar Srivastava, Sriram Krishnamoorthy, and P. Sadayappan. Optimizing Tensor Contractions in CCSD(T) for Efficient Execution on GPUs. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS, page 96–106, New York, NY, USA, 2018. Association for Computing Machinery.

[127] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. A Code Generator for High-Performance Tensor Contractions on GPUs. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, pages 85–95, 2019.

[128] Young Jin Kim, Ammar Ahmad Awan, Alexandre Muzio, Andres Felipe Cruz Salinas, Liyang Lu, Amr Hendy, Samyam Rajbhandari, Yuxiong He, and Hany Hassan Awadalla. Scalable and Efficient MoE Training for Multitask Multilingual Models, 2021.

[129] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[130] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives. In *ISCA*, ISCA, pages 996–1009, Washington, DC, USA, 2020. IEEE, IEEE Computer Society.

[131] Jagadish B Kotra, Michael LeBeane, Mahmut T Kandemir, and Gabriel H Loh. Increasing GPU Translation Reach by Leveraging Under-Utilized On-Chip Resources. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 1169–1181, 2021.

[132] Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead. *ACM Transactions on Architecture & Code Optimization*, 13(1):1:1–1:22, March 2016.

[133] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[134] Rajesh Kumar, Suchita Pati, and Kanishka Lahiri. DARTS: Performance-counter driven sampling using binary translators. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 131–132. IEEE, 2017.

[135] Reese Kuper, Suchita Pati, and Matthew D. Sinclair. Improving GPU Utilization in ML Workloads Through Finer-Grained Synchronization. In *3rd Young Architects Workshop*, YArch, April 2021.

[136] Hyoukjun Kwon, Liangzhen Lai, Tushar Krishna, and Vikas Chandra. HERALD: Optimizing Heterogeneous DNN Accelerators for Edge Devices. *arXiv preprint arXiv:1909.07437*, 57, 2019.

[137] Nagesh B. Lakshminarayana and Hyesoon Kim. Effect of Instruction Fetch and Memory Scheduling on GPU Performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU*, 2010.

[138] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *Proceedings of the Seventh International Conference on Learning Representation*, ICLR. OpenReview.net, 2019.

[139] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K Reinhardt, and Lizy K John. GPU triggered networking for intra-kernel communications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[140] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. ComP-Net: Command Processor Networking for Efficient Intra-Kernel Communications on GPUs. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[141] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K Reinhardt, and Lizy K John. ComP-net: Command processor networking for efficient intra-kernel communications on GPUs. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–13, 2018.

[142] Michael LeBeane, Brandon Potter, Abhisek Pan, Alexandru Dutu, Vinay Agarwala, Wonchan Lee, Deepak Majeti, Bibek Ghimire, Eric Van Tassell, Samuel Wasmundt, Brad Benton, Mauricio Breternitz, Michael L. Chu, Mithuna Thottethodi, Lizy K. John, and Steven K. Reinhardt. Extended Task Queuing: Active Messages for Heterogeneous Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 933–944, 2016.

[143] Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Jaewook Lee, Donguc Ko, Younggun Jun, Keewon Cho, Ilwoong Kim, Choungki Song, Chunseok Jeong, Daehan Kwon, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. A 1ynm 1.25 V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications. In *ISSCC*, volume 65 of *ISSCC*, pages 1–3. IEEE, 2022.

[144] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 515–527, 2015.

[145] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product. In *ISCA*, ISCA, pages 43–56, 2021.

[146] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[147] Jonathan Lew, Deval A. Shah, Suchita Pati, Shaylin Cattell, Mengchi Zhang, Amruth Sandhupatla, Christopher Ng, Negar Goli, Matthew D. Sinclair, Timothy G. Rogers, and Tor M. Aamodt. Analyzing Machine Learning Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 151–152. IEEE, 2019.

[148] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic Horizontal Fusion for GPU Kernels. *CoRR*, 2007.01277, 2020.

[149] Min Lin, Qiang Chen, and Shuicheng Yan. Network In Network. *CoRR*, abs/1312.4400, 2013.

[150] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 751–766, New York, NY, USA, 2018. ACM.

[151] Jie Liu, Jiawen Liu, Wan Du, and Dong Li. Performance Analysis and Characterization of Training Deep Learning Models on Mobile Device. In *IEEE 25th International Conference on Parallel and Distributed Systems*, ICPADS, pages 506–515. IEEE, 2019.

[152] Jiwei Liu, Jun Yang, and Rami Melhem. SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 383–394, 2015.

[153] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray CC Cheung, and Jianfei He. In-Network Aggregation with Transport Transparency for Distributed Training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 376–391, 2023.

[154] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR*, 1907.11692, 2019.

[155] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 388–401, 2022.

[156] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+, 2020.

[157] Justin Luitjens. CUDA Streams: Best Practices and Common Pitfalls, 2014.

[158] Minh-Thang Luong, Ilya Sutskever, Quoc V Le, Oriol Vinyals, and Wojciech Zaremba. Addressing the Rare Word Problem in Neural Machine Translation. *arXiv preprint arXiv:1410.8206*, 2014.

[159] Daniel Lustig and Margaret Martonosi. Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization. In *Proceedings of the 19th International Symposium on High Performance Computer Architecture*, HPCA, page 354–365, USA, 2013. IEEE Computer Society.

[160] Sangkug Lym, Donghyuk Lee, Mike O'Connor, Niladrish Chatterjee, and Mattan Erez. DeLTA: GPU Performance Model for Deep Learning Applications with In-depth Memory System Traffic Analysis. In *ISPASS*, ISPASS, pages 293–303, Washington, DC, USA, 2019. IEEE, IEEE Computer Society.

[161] 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 *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, pages 881–897. USENIX Association, November 2020.

[162] James MacQueen. Some Methods For Classification and Analysis of Multivariate Observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[163] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. Better Together: Jointly Optimizing ML Collective Scheduling and Execution Planning using SYNDICATE. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, pages 809–824, 2023.

[164] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David A. Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim M. Hazelwood, Andrew Hock, Xinyuan Huang, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. MLPerf Training Benchmark. *CoRR*, abs/1910.01500, 2019.

[165] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. *CoRR*, abs/1710.03740, 2018.

[166] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. FP8 Formats for Deep Learning. *CoRR*, abs/2209.05433, 2022.

[167] Microsoft. Turing-NLG: A 17-billion-parameter language model by Microsoft. *Microsoft Research Blog*, 1(8), 2020.

[168] MLPerf. MLPerf Benchmark Suite. `https://mlperf.org/`, 2018.

[169] Diksha Moolchandani, Joyjit Kundu, Frederik Ruelens, Peter Vrancx, Timon Evenblij, and Manu Perumkunnil. AMPeD: An Analytical Model for Performance in Distributed Training of Transformers. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 306–315. IEEE, 2023.

[170] Trevor Mudge. Power: A First-class Architectural Design Constraint. *Computer*, 34(4):52–58, 2001.

[171] Harini Muthukrishnan, Daniel Lustig, David Nellans, and Thomas Wenisch. GPS: A Global Publish-Subscribe Model for Multi-GPU Memory Management. In *54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 46–58, 2021.

[172] Harini Muthukrishnan, David Nellans, Daniel Lustig, Jeffrey A Fessler, and Thomas F Wenisch. Efficient Multi-GPU Shared Memory via Automatic Optimization of Fine-grained Rransfers. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture*, ISCA, pages 139–152. IEEE, 2021.

[173] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families : Industrial Product. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture*, ISCA, pages 57–70, 2021.

[174] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. GraphPIM: Enabling Instruction-level PIM Offloading in Graph Computing Frameworks. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA, pages 457–468. IEEE, 2017.

[175] Sharan Narang. DeepBench. `https://svail.github.io/DeepBench`, 2016.

[176] Sharan Narang and Gregory Diamos. An update to DeepBench with a focus on deep learning inference. `https://svail.github.io/DeepBench-update/`, 2017.

[177] Sharan Narang, Gregory F. Diamos, Shubho Sengupta, and Erich Elsen. Exploring Sparsity in Recurrent Neural Networks. *CoRR*, abs/1704.05119, 2017.

[178] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *Proceedings of the International Symposium on Microarchitecture* (*MICRO*), pages 308–317, December 2011.

[179] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep Learning Recommendation Model for Personalization and Recommendation Systems, 2019.

[180] Thomas Nelson, Axel Rivera, Prasanna Balaprakash, Mary Hall, Paul D. Hovland, Elizabeth Jessup, and Boyana Norris. Generating Efficient Tensor Contractions for GPUs. In *44th International Conference on Parallel Processing*, ICPP, pages 969–978, 2015.

[181] NVIDIA. NVIDIA RISC-V Story. *RISC-V Workshop 2016*, 2016.

[182] NVIDIA. NVIDIA DGX-1 With Tesla V100 System Architecture. https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf, 2017.

[183] NVIDIA. Pro Tip: cuBLAS Strided Batched Matrix Multiply. `https://developer.nvidia.com/blog/cublas-strided-batched-matrix-multiply/`, 2017.

[184] NVIDIA. Apex (A PyTorch Extension). "`https://nvidia.github.io/apex/`", 2018.

[185] NVIDIA. CUDA Stream Management, 2018.

[186] NVIDIA. Megatron-LM Github. `https://github.com/NVIDIA/Megatron-LM`, 2018.

[187] NVIDIA. NVIDIA cuDNN: GPU Accelerated Deep Learning. "`https://developer.nvidia.com/cudnn`", 2018.

[188] NVIDIA. NVIDIA TESLA V100 GPU ACCELERATOR. `https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf`, 2018.

[189] NVIDIA. Easily Deploy Deep Learning Models in Production. "https://www.kdnuggets.com/2019/08/nvidia-deploy-deep-learning-models-production.html", 2019.

[190] NVIDIA. NVIDIA Deep Learning Performance. "https://docs.nvidia.com/deeplearning/performance/index.html", 2019.

[191] NVIDIA. NVIDIA FasterTransformer. "https://github.com/NVIDIA/FasterTransformer/", 2020.

[192] NVIDIA. NVIDIA NCCL, 2020.

[193] NVIDIA. Ride the Fast Lane to AI Productivity with Multi-Instance GPUs. "https://blogs.nvidia.com/blog/2020/05/14/multi-instance-gpus/", 2020.

[194] NVIDIA. NVIDIA A100 TENSOR CORE GPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf, 2021.

[195] NVIDIA. GPUDirect. "https://developer.nvidia. com/gpudirect", 2022.

[196] NVIDIA. H100 Transformer Engine Supercharges AI Training, Delivering Up to 6x Higher Performance Without Losing Accuracy. https://blogs.nvidia.com/blog/h100-transformer-engine/, 2022.

[197] NVIDIA. Efficient GEMM in CUDA. https://github.com/NVIDIA/cutlass/blob/main/media/docs/efficient_gemm.md#parallelized-reductions, 2023.

[198] NVIDIA. NVIDIA Announces DGX GH200 AI Supercomputer. https://nvidianews.nvidia.com/news/nvidia-grace-hopper-superchips-designed-for-accelerated-generative-ai-enter-full-production, 2023.

[199] NVIDIA. NVIDIA DGX H100. https://www.nvidia.com/en-us/data-center/dgx-h100/, 2023.

[200] NVIDIA. NVIDIA H100 TENSOR CORE GPU. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet, 2023.

[201] NVIDIA Corp. NVIDIA cuBLAS. https://developer.nvidia.com/cublas, 2016. Accessed August 6, 2016.

[202] NVIDIA Corp. NVIDIA Multi-Instance GPU (MIG). https://docs.nvidia.com/cuda/mig/index.html, 2021.

[203] Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. Scaling Neural Machine Translation. *CoRR*, abs/1806.00187, 2018.

[204] Nathan Otterness and James H. Anderson. AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems*, volume 165 of *ECRTS*, pages 10:1–10:23, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[205] Nathan Otterness and James H. Anderson. Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices". In *29th International Conference on Real-Time Networks and Systems*, RTNS, page 24–34, New York, NY, USA, 2021. Association for Computing Machinery.

[206] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 407–418, New York, NY, USA, 2013. Association for Computing Machinery.

[207] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210. IEEE, 2015.

[208] 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 *ISPASS*, ISPASS, pages 304–315, Washington, DC, USA, 2019. IEEE, IEEE Computer Society.

[209] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA, pages 27–40, New York, NY, USA, 2017. ACM.

[210] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications, 2018.

[211] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

[212] Suchita Pati. Exploring GPU Architectural Optimizations for RNNs. In *1st Young Architects Workshop*, YArch, February 2019.

[213] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D Sinclair. Tale of Two Cs: Computation vs. Communication Scaling for Future Transformers on Future Hardware. In *IEEE International Symposium on Workload Characterization*, IISWC, October 2023.

[214] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D. Sinclair. T3: Transparent Tracking & Triggering for Fine-grained Overlap of Compute & Collectives. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 1146–1164, New York, NY, USA, 2024. Association for Computing Machinery.

[215] Suchita Pati, Shaizeen Aga, Nuwan Jayasena, and Matthew D. Sinclair. Demystifying BERT: Implications for Accelerator Design. *CoRR*, abs/2104.08335, 2021.

[216] Suchita Pati, Shaizeen Aga, Nuwan Jayasena, and Matthew D. Sinclair. Demystifying BERT: System Design Implications. In *IEEE International Symposium on Workload Characterization*, IISWC, 2022.

[217] Suchita Pati, Shaizeen Aga, Matthew D. Sinclair, and Nuwan Jayasena. SeqPoint: Identifying Representative Iterations of Sequence-based Neural Networks. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 69–80, Washington, DC, USA, August 2020. IEEE Computer Society.

[218] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 81–92, Washington, DC, USA, 2004. IEEE Computer Society.

[219] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. Opportunistic Computing in GPU Architectures. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA, pages 210–223, 2019.

[220] J Thomas Pawlowski. Hybrid Memory Cube (HMC). In *2011 IEEE Hot Chips 23 Symposium*, HotChips, pages 1–24. IEEE, 2011.

[221] Kishore Punniyamurthy, Bradford M Beckmann, and Khaled Hamidouche. GPU-initiated Fine-grained Overlap of Collective Communication with Computation. *arXiv preprint arXiv:2305.06942*, 2023.

[222] Sooraj Puthoor, Xulong Tang, Joseph Gross, and Bradford M. Beckmann. Oversubscribed Command Queues in GPUs. In *Proceedings of the 11th Workshop on General Purpose GPUs*, GPGPU-11, pages 50–60, New York, NY, USA, 2018. ACM.

[223] PyTorch. Pytorch Automatic Mixed Precision Package. `https://pytorch.org/docs/stable/amp.html`, 2019.

[224] PyTorch. gpt-fast. `https://github.com/pytorch-labs/gpt-fast`, 2023.

[225] 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 *26th IEEE International Symposium on High Performance Computer Architecture*, HPCA, pages 58–70. IEEE, 2020.

[226] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI blog*, 1(8):9, 2019.

[227] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019.

[228] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MOE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. In *International Conference on Machine Learning*, ICML, pages 18332–18346. PMLR, 2022.

[229] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. *CoRR*, 1910.02054, 2019.

[230] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[231] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100, 000+ Questions for Machine Comprehension of Text. *CoRR*, abs/1606.05250, 2016.

[232] Vishnu Ramadas, Daniel Kouchekinia, Ndubuisi Osuji, and Matthew D. Sinclair. Closing the Gap: Improving the Accuracy of gem5's GPU Models. In *5th gem5 Users' Workshop*, June 2023.

[233] Vishnu Ramadas, Daniel Kouchekinia, Ndubuisi Osuji, and Matthew D. Sinclair. Further Closing the Gap: Improving the Accuracy of gem5's GPU Models. In *5th Young Architects Workshop*, YArch, April 2024.

[234] Vinay Ramakrishnaiah, Bradford Beckmann, Pete Ehrett, Rene Van Oostrum, and Keith Lowery. Cache Cohort GPU Scheduling. In *Proceedings of the 16th Workshop on General Purpose Processing Using GPU*, GPGPU '24, page 19–25, New York, NY, USA, 2024. Association for Computing Machinery.

[235] Saeed Rashidi, Matthew Denton, Srinivas Sridharan, Sudarshan Srinivasan, Amoghavarsha Suresh, Jade Nie, and Tushar Krishna. Enabling compute-communication overlap in distributed deep learning training platforms. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, ISCA, pages 540–553. IEEE, 2021.

[236] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. In *ISPASS*, ISPASS, pages 81–92, Washington, DC, USA, 2020. IEEE, IEEE Computer Society.

[237] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. MLPerf Inference Benchmark. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA, pages 446–459, Washington, DC, USA, 2020. IEEE Press.

[238] Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. A Generalist Agent. *CoRR*, 2022.

[239] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *HPCA*, HPCA, pages 598–611, Washington, DC, USA, 2021. IEEE Computer Society.

[240] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training, 2021.

[241] Jens Rettkowski and Diana Göhringer. Data Stream Processing in Networks-on-Chip. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 633–638, 2017.

[242] Sia Rezaei. Context is everything: Why maximum sequence length matters. `https://www.cerebras.net/chip/context-is-everything-why-maximum-sequence-length-matters/`, 2022.

[243] Kyle Roarty and Matthew D. Sinclair. Modeling Modern GPU Applications in gem5. In *3rd gem5 Users' Workshop*, June 2020.

[244] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-Conscious Thread Scheduling for Massively Multithreaded Processors. *IEEE Micro, Special Issue: Micro's Top Picks from 2012 Computer Architecture Conferences*, 2013.

[245] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Divergence-Aware Warp Scheduling. In *In 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 99–110, 2013.

[246] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Learning Your Limit: Managing Massively Multithreaded Caches Through Scheduling. *Communications of the ACM*, December 2014.

[247] Bita Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, Alessandro Forin, Haishan Zhu, Taesik Na, Prerak Patel, Shuai Che, Lok Chand Koppaka, Xia Song, Subhojit Som, Kaustav Das, Saurabh Tiwary, Steve Reinhardt, Sitaram Lanka, Eric Chung, and Doug Burger. Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NeurIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[248] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988.

[249] Charbel Sakr, Steve Dai, Rangha Venkatesan, Brian Zimmer, William Dally, and Brucek Khailany. Optimal Clipping and Magnitude-aware Differentiation for Improved Quantization-aware Training. In *ICML*, ICML, pages 19123–19138. PMLR, 2022.

[250] Samsung. GPU with HBM PIM. `https://semiconductor.samsung.com/newsroom/tech-blog/samsung-electronics-semiconductor-unveils-cutting-edge-memory-technology-to-accelerate-next-generation-ai/`, 2022.

[251] Karthik Sangaiah, Michael Lui, Ragh Kuttappa, Baris Taskin, and Mark Hempstead. Snacknoc: Processing in the communication layer. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 461–473. IEEE, 2020.

[252] Aarush Selvan and Pankaj Kanwar. Google showcases Cloud TPU v4 Pods for large model training. `https://cloud.google.com/blog/topics/tpus/google-showcases-cloud-tpu-v4-pods-for-large-model-training`, 2022.

[253] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, pages 593–612, 2023.

[254] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, 2002.

[255] Yang Shi, U. N. Niranjan, Animashree Anandkumar, and Cris Cecka. Tensor Contractions with Extended BLAS Kernels on CPU and GPU. In *IEEE 23rd International Conference on High Performance Computing*, HiPC, pages 193–202, 2016.

[256] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR*, abs/1909.08053, 2019.

[257] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2014.

[258] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 647–659, Washington, DC, USA, December 2015. IEEE Computer Society.

[259] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. Cache Coherence for GPU Architectures. In *19th International Symposium on High Performance Computer Architecture*, HPCA, pages 578–590, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[260] 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*, ASPLOS, page 909–923, New York, NY, USA, 2019. Association for Computing Machinery.

[261] SK Hynix. GDDR-PIM. `https://news.skhynix.com/sk-hynix-develops-pim-next-generation-ai-accelerator/`, 2022.

[262] Sklearn. Sklearn Multi-class Logistic Regression. `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression`, 2019.

[263] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv preprint arXiv:2201.11990*, 2022.

[264] Matthias Springer, Peter Wauligmann, and Hidehiko Masuhara. Modular Array-Based GPU Computing in a Dynamically-Typed Language. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2017, page 48–55, New York, NY, USA, 2017. Association for Computing Machinery.

[265] Qingxiao Sun, Yi Liu, Hailong Yang, Ruizhe Zhang, Ming Dun, Mingzhen Li, Xiaoyan Liu, Wencong Xiaoy, Yong Liy, Zhongzhi Luan, et al. CoGNN: Efficient Scheduling for Concurrent GNN Training on GPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 538–552. IEEE Computer Society, 2022.

[266] Yu Sun, Shuohuan Wang, Yukun Li, Shikun Feng, Hao Tian, Hua Wu, and Haifeng Wang. ERNIE 2.0: A Continual Pre-training Framework for Language Understanding. *CoRR*, 1907.12412, 2019.

[267] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, CVPR, pages 1–9, 2015.

[268] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *CoRR*, abs/1512.00567, 2015.

[269] Xiaodan Tan. *GPUPool: A Holistic Approach to Fine-Grained GPU Sharing in the Cloud*. PhD thesis, University of Toronto (Canada), 2021.

[270] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805*, 2023.

[271] TIRIAS Research. Why Your AI infrastructure Needs Both Training and Inference. "https://www.ibm.com/downloads/cas/QM4BYOPP", 2019.

[272] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-Read Students Learn Better: On the Importance of Pre-training Compact Models. *CoRR*, abs/1908.08962, 2019.

[273] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NeurIPS, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[274] Snehil Verma, Qinzhe Wu, Bagus Hanindhito, Gunjan Jha, Eugene B John, Ramesh Radhakrishnan, and Lizy K John. Demystifying the MLPerf Benchmark Suite. In *ISPASS*, ISPASS, pages 24–33. IEEE, 2020.

[275] Guibin Wang, YiSong Lin, and Wei Yi. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, page 344–350, USA, 2010. IEEE Computer Society.

[276] Hanrui Wang, Zhekai Zhang, and Song Han. SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning. In *HPCA*, HPCA, pages 97–110, Los Alamitos, CA, USA, mar 2021. IEEE Computer Society.

[277] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training Deep Neural Networks with 8-bit Floating Point Numbers. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NeurIPS, pages 7686–7695, 2018.

[278] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap Communication with Dependent Computation via Decomposition in Large Deep Learning Models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS, pages 93–106, 2022.

[279] Yu Wang, Gu-Yeon Wei, and David Brooks. A Systematic Methodology for Analysis of Deep Learning Hardware and Software Platforms. *Proceedings of Machine Learning and Systems*, 2020:30–43, 2020.

[280] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR*, abs/1609.08144, 2016.

[281] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, page 84–97, New York, NY, USA, 2003. Association for Computing Machinery.

[282] Wayne Xiong, , Xuedong Huang, Frank Seide, , and Andreas Stolcke. Toward Human Parity in Conversational Speech Recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 25:2410–2423, Sept 2017.

[283] Qiumin Xu and Murali Annavaram. PATS: Pattern Aware Scheduling and Power Gating for GPGPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (*PACT*), pages 225–236, 2014.

[284] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *CoRR*, 1906.08237, 2020.

[285] Tsung Tai Yeh, Matthew D. Sinclair, Bradford M. Beckmann, and Timothy G. Rogers. Deadline-Aware Offloading for High-Throughput Accelerators. In *27th IEEE International Symposium on High Performance Computer Architecture*, HPCA, pages 479–492, 2021.

[286] Bobbi W Yogatama, Matthew D Sinclair, and Michael M Swift. Enabling multi-gpu support in gem5. In *gem5 Users Workshop*, 2020.

[287] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Reducing BERT Pre-Training Time from 3 Days to 76 Minutes. *CoRR*, abs/1904.00962, 2019.

[288] Yulong Yu, Weijun Xiao, Xubin He, He Guo, Yuxin Wang, and Xin Chen. A Stall-Aware Warp Scheduling for Dynamically Optimizing Thread-level Parallelism in GPGPUs. In *Proceedings of the ACM international conference on Supercomputing*, pages 15–24, 2015.

[289] Ali Hadi Zadeh, Zissis Poulos, and Andreas Moshovos. Deep Learning Language Modeling Workloads: Where Time Goes on Graphics Processors. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 131–142, Washington, DC, USA, 2019. IEEE, IEEE Computer Society.

[290] Claire Zhang and Rebecca Lewington. Genomics in Unparalleled Resolution: Cerebras Wafer-Scale Cluster Trains Large Language Models on the Full COVID Genome Sequence. `https://www.cerebras.net/blog/genomics-in-unparalleled-resolution-cerebras-wafer-scale-cluster-trains-large-language-models-on-the-full-covid-genome-sequence`, 2022.

[291] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster. In *2018 USENIX Annual Technical Conference*, USENIX ATC 18, pages 951–965, Boston, MA, 2018. USENIX Association.

[292] Yu Zhang, Wei Han, James Qin, Yongqiang Wang, Ankur Bapna, Zhehuai Chen, Nanxin Chen, Bo Li, Vera Axelrod, Gary Wang, et al. Google USM: Scaling Automatic Speech Recognition Beyond 100 Languages. *arXiv preprint arXiv:2303.01037*, 2023.

[293] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. Echo: Compiler-Based GPU Memory Footprint Reduction for LSTM RNN Training. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA, page 1089–1102, Piscataway, NJ, USA, 2020. IEEE Press.

[294] Feiwen Zhu, Jeff Pool, Michael Andersch, Jeremy Appleyard, and Fung Xie. Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-Chip. In *Proceedings of 6th International Conference on Learning Representations*, ICLR, 2018.

[295] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. TBD: Benchmarking and Analyzing Deep Neural Network Training. In *IEEE International Symposium on Workload Characterization*, IISWC, Washington, DC, USA, October 2018. IEEE Computer Society.