

© 2025 Hyoungwook Nam

ARCHITECTURES FOR MACHINE LEARNING AND MACHINE LEARNING FOR
ARCHITECTURE

BY

HYOUNGWOOK NAM

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Siebel School of Computing
in the Graduate College of the
University of Illinois Urbana-Champaign, 2025

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair and Director of Research
Associate Professor Bo Li
Assistant Professor Charith Mendis
Distinguished Research Staff Manager Dr. Pradip Bose, IBM Research
Professor Nam Sung Kim
Assistant Professor Raghavendra Pradyumna Pothukuchi, UNC Chapel Hill

ABSTRACT

The unprecedented success of machine learning (ML) has brought a problem and an opportunity to computer architecture research. The problem is how to build efficient and scalable computer systems for ML computations. The opportunity is how can ML methods help solving research problems in computer architecture.

This dissertation explores both directions of research. In the direction of computer architecture for ML, this work focuses on two challenges for large-scale ML: scalability and power efficiency. This dissertation proposes two proposals for these: MeshSlice and PowerGrad. For the other direction, this dissertation proposes FriendlyFoe, which uses adversarial ML for hardware security.

The first proposal is *MeshSlice*, a framework for efficient 2D tensor parallelism (TP) in distributed DNN training. MeshSlice consists of a novel 2D GeMM algorithm and an autotuner. The MeshSlice GeMM algorithm slices the collective communications into multiple partial collectives that allow overlapping communications with computations. As a result, MeshSlice hides most of the communication latency. The MeshSlice LLM autotuner automates finding the optimal configuration of 2D GeMM dataflow, the mesh shape, and the communication granularity using an analytical cost model. MeshSlice shows significant speedup in LLM training workloads compared to the state-of-the-art 2D TP method.

The second proposal is *PowerGrad*, a gradient-based hierarchical power management framework for power-limited ML inference environments. The main idea of PowerGrad is simple: identify, at runtime, how much the performance of each workload benefits from extra power, and hierarchically shift power in the datacenter from workloads that benefit the least to those that benefit the most. In practical terms, PowerGrad dynamically computes the derivative of each compute unit’s performance over power (i.e., the performance *gradient*), and shifts power from lower-gradient units to higher-gradient ones. PowerGrad shows a promising result in local CPU power control, automatically achieving high power efficiency using only hardware performance counters.

The final proposal is *FriendlyFoe*, which dynamically applies Adversarial Machine Learning (AML) to obfuscate side channels. FriendlyFoe defines a workflow to design obfuscation DNNs called *Defenders* with low overhead and information leakage, and to customize them for different environments. Defenders are transferable, i.e., they thwart attacker classifiers that are different from those used to train the Defenders. They also resist adaptive attacks, where attackers train using the obfuscated signals collected while the Defender is active. Fi-

nally, the approach is general enough to be applicable to different environments. FriendlyFoe is demonstrated against two side channel attacks: one based on memory contention and one on system power. FriendlyFoe is an efficient obfuscation method to defend against hardware side channels. Compared to current defenses, FriendlyFoe either 1) reduces the performance overhead with a similar level of security or 2) improves the security with a similar level of performance overhead.

ACKNOWLEDGMENTS

This dissertation would not be possible without help and support from numerous people around me. I want to express my gratitude to my advisor, Prof. Josep Torrellas. I could focus on studying and researching computer architecture thanks to his support, grow my knowledge and skills as a researcher thanks to his advice, and continue the PhD progress thanks to his patience.

I thank Dr. Raghav Pothukuchi, for being my research mentor for my entire PhD years. He had been a great senior student who guided my research and has been helping my research even after he became a PhD. His help spans everywhere of my research — developing topics, discussing the ideas, analyzing the results, writing the papers and many more.

I can never appreciate enough the love and support from my parents. My long academic journey has been possible because they have always shown their belief in me and kept a safe and warm shelter to recharge. I also thank my fiancée Soo Yeon, for all the emotional support. I could remain mentally stable during the hardest years of my progress thanks to our intimate relationship.

In addition, I am grateful to various institutions that financially supported my research. This thesis work was supported by NSF under grants CNS 1956007, CCF 2107470, and CCF 2316233; by Intel Corporation under the Resilient Architectures and Robust Electronics (RARE) program; by the IBM-Illinois Discovery Accelerator Institute; and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Last but not least, I thank my peers in Iacoma group and co-authors of my research papers. Iacoma group has the greatest PhD students in the world, who gave me endless motivations and insights for research. Moreover, I learned a lot from my research collaborators thanks to their brilliant advises and discussions.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION	1
1.1 Overview	1
1.2 Large Scale Machine Learning	1
1.3 Scalability of Distributed DNN Training Systems	3
1.4 Power Efficiency of ML Inference in Datacenters	5
1.5 Adversarial ML for Computer Hardware Security	8
1.6 Thesis Contributions	10
CHAPTER 2 MESHSLICE: EFFICIENT 2D TENSOR PARALLELISM FOR DISTRIBUTED DNN TRAINING	12
2.1 Background	12
2.2 Problem Addressed	19
2.3 The MeshSlice 2D GeMM Algorithm	20
2.4 The MeshSlice LLM Autotuner	25
2.5 Evaluation Setup	29
2.6 Evaluation Results	32
2.7 Discussion and Related Work	41
CHAPTER 3 POWERGRAD: GRADIENT-BASED HIERARCHICAL POWER MANAGEMENT OF POWER-LIMITED ML INFERENCE SYSTEMS	42
3.1 Motivation	42
3.2 The PowerGrad Framework	43
3.3 Experimental Setup	53
3.4 PowerGrad Evaluation	56
3.5 Related Work	60
3.6 Discussion	61
CHAPTER 4 FRIENDLYFOE: ADVERSARIAL MACHINE LEARNING AS A PRACTICAL ARCHITECTURAL DEFENSE AGAINST SIDE CHANNEL ATTACKS	64
4.1 Background	64
4.2 FriendlyFoe	65
4.3 Two Target Side Channel Attacks	70
4.4 FriendlyFoe Workflow	72
4.5 Experimental Methodology	78
4.6 Memory Contention Side Channel Defense Evaluation	80
4.7 Power Side Channel Defense Evaluation	86
4.8 Related Work	88

CHAPTER 5 CONCLUSION AND FUTURE WORK	90
5.1 Conclusions	90
5.2 Future Work	91
REFERENCES	93

CHAPTER 1: INTRODUCTION

1.1 OVERVIEW

The unprecedented success of machine learning (ML) has brought a problem and an opportunity to computer architecture research. The problem is how to build efficient and scalable computer systems for ML computations, as the rapid expansion of generative AI and large language models (LLMs) is driving an exponential growth of computing and power demands. The opportunity is how can ML methods help solving research problems in computer architecture, as data-driven ML methods have been proven to provide better solutions to many problems compared to traditional methods.

This dissertation explores both directions of research. In terms of computer architecture for ML, this work focuses on two challenges for large-scale ML: scalability and power efficiency. The proposals are MeshSlice and PowerGrad. MeshSlice designs a novel 2D GeMM algorithm for efficient 2D tensor parallelism, and PowerGrad designs a hierarchical power management framework for ML inference clusters. For the other direction, this dissertation explores using ML for hardware security. The proposal is FriendlyFoe, which applies adversarial machine learning to secure computer hardware from side-channel attacks.

1.2 LARGE SCALE MACHINE LEARNING

The remarkable success of large language models (LLMs) and generative AI is fundamentally transforming the landscape of computational demand of high performance computing (HPC) datacenters. In natural language processing, LLMs have achieved near-human or superhuman performance in tasks ranging from reading comprehension and question answering to summarization and translation [2, 3]. Beyond traditional language tasks, these models have demonstrated surprising capabilities in complex reasoning domains such as mathematical problem-solving [4] and software code generation [5]. The models are also capable of handling multi-modal problems, effectively handling tasks involving images and audio with text, from audio translation to visual question answering [6].

The commercial success of LLMs and generative AI has driven the unprecedented growth of ML computing demands. Recently, the computational requirements for training and serving state-of-the-art language models have been exponentially growing every year, as shown in Figure 1.1. GPT-3 [7], which has 175 billion parameters (weights and biases) initiated this trend in 2020. It was followed by larger models with triple parameter sizes

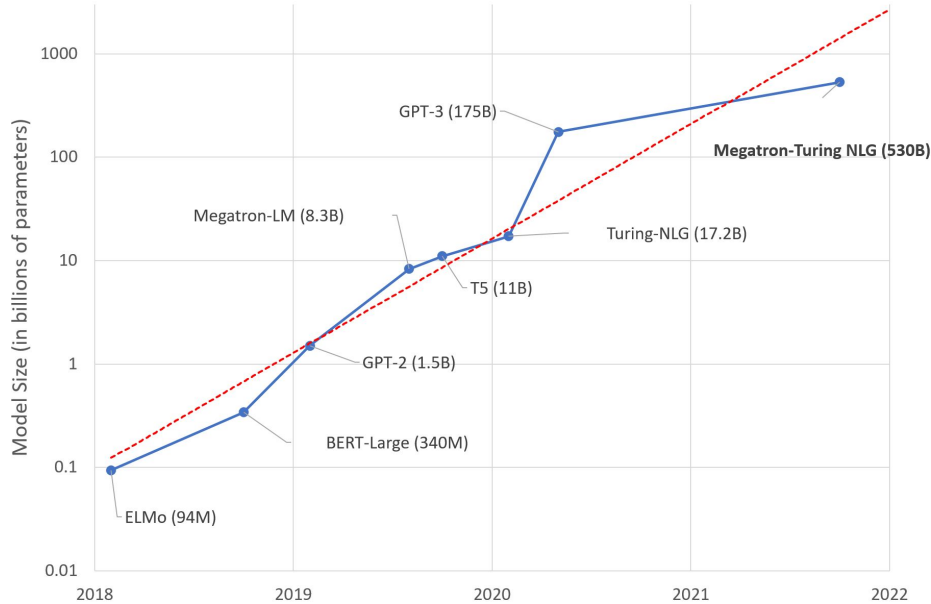


Figure 1.1: The scaling trend of large language models [1].

in 2022, such as Megatron-NLG [8] with 530 billion parameters and Pathways Language Model (PaLM) [9] with 540 billion parameters. Most recently, the scale of commercial LLMs are tripled again in 2023, including OpenAI’s GPT-4 [2] with 1.8 trillion parameters and Google’s Gemini-Ultra [3] with 1.5 trillion parameters. This exponential scaling has been driven by the empirical success of these larger models in achieving better performance across a wide range of tasks.

For ML workloads, there are two types of computing demands: training and inference. In *training*, the parameters of an ML model are updated with optimization algorithms such as stochastic gradient descent with backpropagation [10]. ML model training typically has massive data parallelism, as training a large deep neural network (DNN) uses billions of data inputs and iterates over the same operations numerous times. Therefore, training an ML model is extremely compute intensive and requires a large cluster of thousands of accelerator chips [8, 11].

After training, the model parameters are quantized to smaller precisions (e.g. 16-bit floating point to 8-bit integer) and the model is deployed to a cloud system or a user device. Then, the deployed model takes user inputs to make *inference*, which involves computing and generating outputs from input data that is unseen during training.

ML inference has different computational demands compared to ML training. As users send a variety of small requests, inference has abundant task parallelism with dynamic workload demands. Also, ML services fine-tune the models to create task-specific ML models,

generating further variations in workload characteristics and compute demands. Often, ML inference is served by a cloud system of multiple accelerator nodes, where each node owns an ML model to serve specific user needs.

This thesis work addresses two different efficiency problems of machine learning training and inference. For training, this work focuses on efficient scaling of distributed DNN training, focusing on optimizing the communication cost in a large distributed accelerator clusters. For inference, this work addresses the power efficiency of ML cloud inference clusters, focusing on handling the dynamic nature of ML inference workloads. As companies are spending billions of dollars annually to fulfill the compute and energy demands of large scale ML training and inference, the need for scalable, efficient computing systems becomes increasingly urgent for a sustainable future.

1.3 SCALABILITY OF DISTRIBUTED DNN TRAINING SYSTEMS

The first problem explored in this thesis is the *scalability* of ML training systems. DNN models are growing in size dramatically – especially, transformer-based [12] LLMs. These models must be trained using a distributed cluster, not only to satisfy their massive computing demands but also to hold their large memory footprint.

To parallelize a DNN running on a distributed cluster, three approaches exist: data parallelism (DP) [13, 14], pipeline parallelism (PP) [15], and tensor parallelism (TP) [16, 17]. Due to the large computing and memory demands of LLMs, most LLM training methods use all three types of parallelism together, forming 3D training clusters [8, 11, 18, 19] as illustrated in Figure 1.2. Of the three types of parallelism, TP usually has the least parallelism because it incurs the most communication cost. This is because TP requires communicating the input or output matrix shards at every general matrix multiplication (GeMM), and GeMMs account for the majority of DNN computations. To continue to increase the size of DNN models, it is necessary to scale all three types of parallelism and, in particular, resolve the communication bottleneck of TP.

One way to mitigate the communication cost of TP is to make it two dimensional (2D). 2D TP distributes matrix shards into a 2D mesh of accelerators, and performs 2D distributed GeMM computations. In a 2D distributed GeMM, a shard of a matrix is communicated only to the accelerators in the same row or column of the 2D mesh. As a result, 2D GeMM incurs less traffic than 1D GeMM, where a matrix shard must be communicated to all accelerators. Consequently, by replacing 1D TP with 2D TP, one can attain higher parallelism at a similar communication cost, or the same parallelism at a smaller communication cost. Additionally, a higher degree of parallelism via 2D TP reduces the communication overhead for DP, as

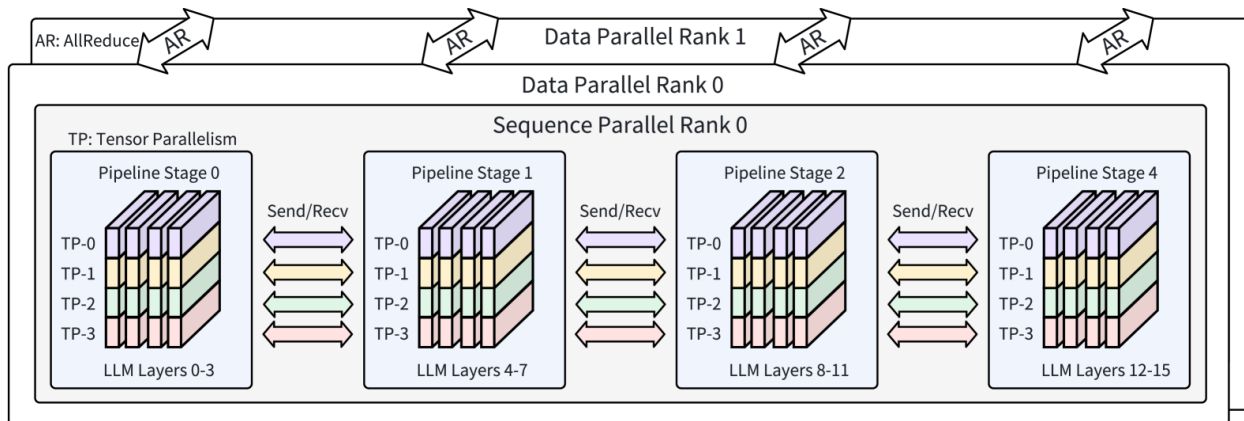


Figure 1.2: The structure of 3D distributed LLM training [20].

each accelerator holds smaller weight matrix shards.

In practice, optimizing 2D TP is a challenging problem that requires careful architectural considerations. The first difficulty is to find an efficient 2D GeMM algorithm, as existing algorithms suffer from inefficiencies. Specifically, Cannon’s algorithm [21] is a traditional method for 2D GeMM that incurs high traffic because it requires skewing the matrix shards, and only supports square meshes. The SUMMA algorithm [22] has less traffic than Cannon by choosing an optimal mesh shape, but relies on fine-grain *broadcast* and *reduce* communication operations. These primitives pipeline the communications into small packet transfers, which result in many synchronizations. This is inefficient in the high-bandwidth inter-chip interconnects (ICI) of contemporary ML clusters with a large number of accelerators.

Google’s TPU clusters compute 2D GeMMs using *AllGather* (AG) and *ReduceScatter* (RdS) collective communication operations [23]. They fully utilize the ICI bandwidth via high communication parallelism. However, this approach combines all partial communications into a single collective communication. As a result, it is not possible to overlap communications with computations via software pipelining. Wang et al. [24] partially solve this problem by partitioning a collective communication into multiple *SendRecv* operations to overlap communications with computations. 2D GeMM involves communication operations in two directions (i.e., row and column). Unfortunately, Wang’s partitioning method can only be applied to one direction; applying the partitioning to both directions requires using Cannon’s algorithm.

The second difficulty of 2D TP is to find the optimal configurations of the many hyperparameters in the 2D GeMM algorithm. The hyperparameters that affect the communication cost of 2D GeMM include how matrices are sharded in a mesh (i.e., the *sharding*), how each shard moves in the mesh of accelerators (i.e., the *dataflow*), and what the row and column

sizes are (i.e., the mesh shape). Currently, finding the optimal configuration of these knobs requires expert knowledge and repeated experiments.

To address these challenges, this thesis proposes *MeshSlice*, a new, efficient distributed GeMM algorithm for 2D TP. MeshSlice *slices* the AG/RdS collective communication operations into multiple partial AG/RdS operations in both row and column directions. Developing a correct slicing algorithm for collective communications in both directions is a complex problem, and MeshSlice solves it for the first time. With this approach, MeshSlice overlaps most of the communications with computations—unlike Wang’s algorithm. Further, unlike Cannon, MeshSlice supports different mesh shapes to minimize the traffic cost. Finally, unlike SUMMA, MeshSlice uses efficient AG and RdS operations for high ICI bandwidth utilization.

This thesis also introduces the *MeshSlice LLM autotuner*, which automatically optimizes MeshSlice’s hyperparameters for distributed LLM training. The autotuner finds the hyperparameters in two phases. It begins by choosing an efficient 2D GeMM dataflow, and then uses analytical cost models to co-optimize the mesh shape and the communication granularity.

MeshSlice is evaluated against multiple baseline algorithms by simulating TPUv4 [25] clusters training the GPT-3 [7] and Megatron-NLG [8] LLM models. MeshSlice maintains high efficiency up to at least 256-way 2D TP. In a cluster of 256 TPUs, MeshSlice trains the GPT-3 and Megatron-NLG models 12.0% and 23.4% faster, respectively, than the state-of-the-art. MeshSlice has an implementation running on Google TPUv4 clusters. However, it does not bring speedups, because the current TPUv4 hardware and software stack cannot overlap AG and RdS communications with computations. Meanwhile, the evaluations on the real TPU cluster show that MeshSlice’s slicing incurs only a small overhead, and that the autotuner’s cost models can accurately estimate the communication and compute costs.

1.4 POWER EFFICIENCY OF ML INFERENCE IN DATACENTERS

The second problem explored in this thesis is the power efficiency of ML inference clusters. The rapid expansion of datacenters, driven by the increasing demand for computational power to serve ML workloads, has led to a significant rise in energy consumption, as shown in Figure 1.3. This has brought the issue of power management in datacenters to the forefront of sustainable computing.

The challenge is further compounded by the integration of renewable energy sources, which can often result in *power availability being lower* than the power demand, due to fluctuations [27] or due to Demand-Response (DR) actions [28]. In such *power-limited* environments,

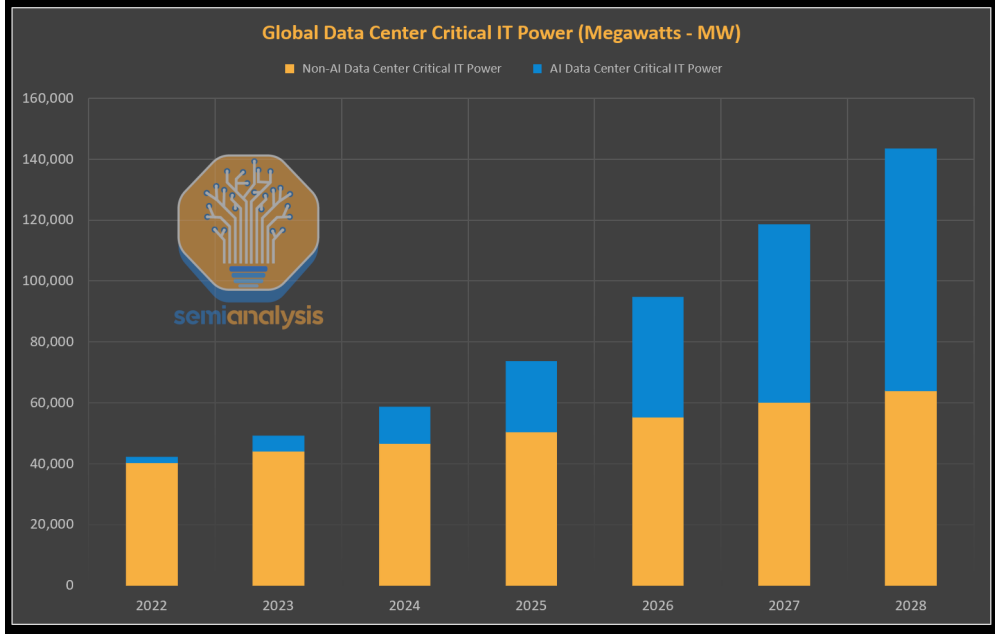


Figure 1.3: Global power demand forecast for datacenters [26].

the distribution of power between nodes in a computer cluster plays a critical role in overall efficiency. For better efficiency, the system should not naively distribute power equally, but intelligently allocate higher power to workloads whose performance is more sensitive to power, and lower power to power-insensitive workloads. Finding an optimal distribution of bounded system power that maximizes efficiency is a challenging resource control problem.

ML workloads are similar to high performance computing (HPC) workloads in that they exhibit high power consumption. However, ML workload behavior is more dynamic and unpredictable than that of traditional HPC workloads. A main reason is that datacenters usually serve a wide variety of different ML models. Such models are trained for different services and have different sizes. Also, even within the same model, the compute and power demand varies based on user inputs. For instance, there can be many homogeneous requests with long input prompts that can be merged into large batches, or there can be a variety of small-batch requests with short prompts. Moreover, within the same request, the workload behavior dynamically changes over time. For example, LLM computations are done in two phases: an encoding phase that is often compute bound and a decoding phase that is often memory bound.

The dynamic and non-deterministic nature of ML inference brings several power management challenges. First, the system administrator cannot profile the workload ahead of time, as users launch a variety of requests that have different power and performance characteristics. Moreover, an ML datacenter serves different models concurrently, and the mix

of models is frequently updated whenever a new model is introduced. Finally, most requests should be executed with a short response time. As a result, the power management system for ML inference clusters should be dynamic, scalable, agile, and not dependent on workload-specific parameters.

Most existing methods for datacenter power control do not satisfy these requirements. Many of them are *not software-transparent*, in that they either rely on application profiling [29, 30, 31] or work only for specific types of applications [32]. We cannot apply such algorithms to ML inference clusters because it is not possible to profile all combinations of different ML models and user inputs ahead of time.

On the other hand, there are software-transparent methods that rely on heuristics based on power usage [33, 34]. These heuristics are not intelligent enough to maximize the power efficiency of the cluster. They monitor power use but do not identify which workloads are compute-bound and which are memory-bound—which would be needed to assign the power in a more efficient manner. In addition, these methods typically use *centralized* algorithms that require information from all the nodes at every time step. Centralized algorithms are not scalable in that they are too slow in large-scale datacenters with rapidly-changing workloads.

To address these challenges, this thesis proposes an intelligent and scalable power management framework for ML inference clusters. We call the framework *PowerGrad*. PowerGrad intelligently shifts power between workloads using a *gradient-based* optimization algorithm. Instead of profiling the workloads ahead of time, PowerGrad dynamically analyzes hardware counters (e.g., instruction count, cache-misses) of individual CPUs to compute the *performance gradient* of the workloads they run—namely, the sensitivity of the workload performance to power changes. Then, PowerGrad shifts power from the CPUs with low-gradient workloads to the CPUs with high-gradient ones. The result is a net performance gain, since the former gain more performance than the performance lost from the latter.

A key strength of PowerGrad is that its controllers are *hierarchically* organized. Child controllers re-distribute the power among the CPU chips in a node. They also pass the gradients and measurements of the node to a parent controller, which re-distributes the power among the different nodes. Unlike centralized control methods, PowerGrad’s hierarchical control enables high-frequency actuation in the lower levels, which is crucial to handle rapidly-changing ML inference workloads.

We implement PowerGrad as a software framework with three components: Gradient Estimator, Local Controller, and Hierarchical Controller. The Gradient Estimator dynamically reads hardware performance counters and generates polynomial (thus differentiable) power and performance models online. Then, the Gradient Estimator differentiates the polynomial

models to estimate the performance gradient $\frac{\partial perf}{\partial power}$ of each CPU chip, which is the basis for the PowerGrad optimization algorithm. Using the gradients, the Local Controller re-distributes the power inside the node to maximize total performance within the node power budget. Then, the Hierarchical Controller collects the aggregated gradients and hardware metrics from its child controllers and re-distributes the power budgets across the nodes. Local and Hierarchical controllers use the same gradient-based algorithm so that one can recursively add extra levels of hierarchy to the system. The parent controller runs asynchronously with the child controllers, allowing a faster control frequency in lower levels. The PowerGrad control periodically runs to adjust the power budgets dynamically in runtime.

We evaluate PowerGrad on a CPU cluster running four different ML applications. In power-limited environments, PowerGrad is effective at maximizing the system performance. When the power budget per node is limited to 75–55W, PowerGrad reduces the average and tail latencies by an average of 20.3% and 20.4%, respectively, relative to a state-of-the-art baseline. When the node power budget is very low at 55W, PowerGrad reduces the average and tail latencies by an average of 25.7% and 26.3%, respectively, over the baseline. Furthermore, PowerGrad’s hierarchical structure makes it more robust to longer control periods, increasing its effectiveness over other schemes in large-scale clusters.

1.5 ADVERSARIAL ML FOR COMPUTER HARDWARE SECURITY

The third problem explored in this dissertation is using ML to secure computer hardware. Securing computers against information leakage has never been more challenging. Attackers can use complex analyses to exfiltrate sensitive information from a variety of side channels. Recently, attackers have begun using ML for side channel analysis—a move that greatly amplifies the risk of information leakage for many systems (e.g., [35, 36, 37, 38]).

ML-based side channel analysis (shown in Figure 1.4) is powerful for several reasons [35, 38, 39, 40, 41]. First, ML models learn information-carrying patterns from the dataset automatically, without domain-specific assumptions or feature engineering. In contrast, earlier approaches like template attacks assumed particular properties of the collected signals and targeted specific connections between the data [35]. As a result, they often fail to leverage all the correlations in the data, and become vulnerable to countermeasures. Second, ML networks can circumvent simple defenses like noise addition, masking, and shuffling. Finally, ML networks can work with raw data, lowering the efforts to construct leaky side channels and pre-process signals. Indeed, the use of ML is a game-changer that generally makes complex side channel analysis accessible, practical, and successful.

In contrast, most current approaches to obfuscate information against side channel analysis

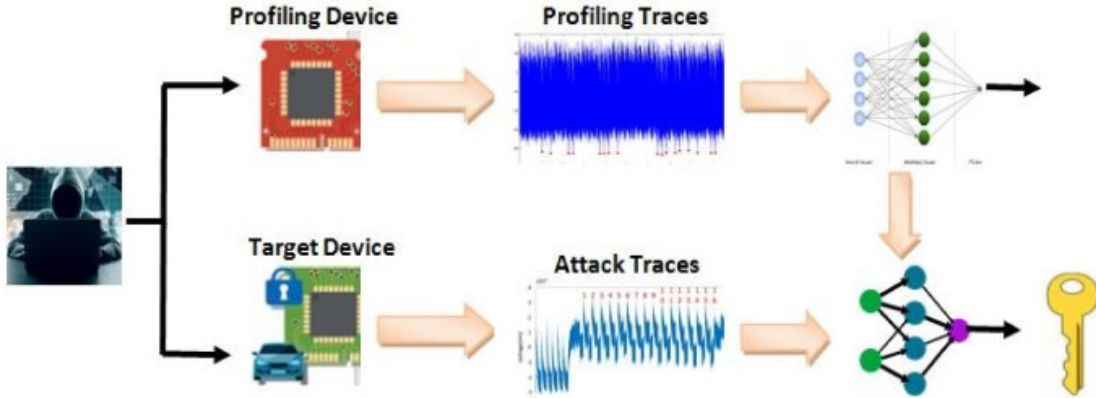


Figure 1.4: Overview of ML-based side channel attacks [42].

(e.g., [43, 44, 45]) are specific to particular attacks and circumstances. They are hard to generalize across different types of side channels. Further, the signal distortions they create often result in high performance overheads. Therefore, it is necessary to develop more intelligent defense strategies that are general and minimize performance overheads.

An intriguing approach is to use adversarial machine learning (AML) for defense. AML is a technique that intelligently perturbs the inputs of an ML classifier such that it causes a misclassification. AML has become popular in confusing image classifiers [46], and offers an opportunity to defeat ML-based side channel analyses with low performance overhead.

There are two prior works that use AML as a countermeasure [47, 48] against ML analysis of signals. These methods require the full trace of a signal to determine how it should be distorted. Such a trace is only available after the system execution. By the time these schemes start post-processing the signal, the information has already leaked to an attacker. To protect real systems, we need a dynamic defense that produces the perturbations *on the fly*.

This dissertation shows that on-the-fly AML is an effective and general architectural technique to obfuscate side channel signals. The approach is based on ML methodology and the ability to operate dynamically. We call the approach *FriendlyFoe* and the network to perturb signals *FriendlyFoe Defender*. We show the practicality, efficacy, and generality of *FriendlyFoe* against computer hardware side channel attacks.

The core of our proposal is to train a *FriendlyFoe Defender* using a Generative Adversarial Network (GAN) [46, 49]. This approach trains three networks together: the Defender, which adds noise to a signal, and two attackers (classifier and discriminator) that take the noisy signal and try to make a correct guess. All networks are trained adversarially: the Defender to add noise to mislead the attackers and the attackers to identify the correct label. Through

this, the Defender is trained to minimize information leakage with minimal noise addition, even against a strong adaptive attacker.

We propose a FriendlyFoe workflow to build Defenders for different environments. Such Defenders are *transferable*—i.e., they are effective against other types of attacker classifiers beyond those they trained with. Moreover, they also resist *adaptive* attacks, where attacker classifiers train using the obfuscated signals collected while the Defender is in operation. In addition, Defenders can exhibit inter-application transferability, where a Defender trained for one victim application can provide a reasonable level of protection to other victim applications. Finally, the approach is general, as it can be applied to different side channel analyses that collect time-series signals and apply pattern recognition to decode secret information.

We demonstrate FriendlyFoe by applying it to two different side channel attacks: one based on memory contention and one on system power. The first example uses a hardware Defender module with ns-level response time that, for the same level of security as a Pad-to-Constant scheme, has 27% and 64% lower performance overhead for single- and multi-threaded workloads, respectively. The second example uses a software Defender with ms-level response time that reduces leakage by $3.7\times$ over a state-of-the-art scheme while reducing the energy overhead by 22.5%.

1.6 THESIS CONTRIBUTIONS

This dissertation includes three main contributions: MeshSlice, PowerGrad, and FriendlyFoe.

MeshSlice MeshSlice is a novel 2D GeMM algorithm that solves the inefficiency problems of existing 2D GeMM algorithms. It partitions the 2D GeMM communications into partial AG and RdS communications using its slicing mechanism. MeshSlice also has an LLM autotuner, which finds an efficient combination of dataflow, mesh shape, and communication granularity. Finding an efficient configuration for a 2D GeMM has been a hard problem that requires human intervention, but the MeshSlice LLM autotuner automatically solves the problem for LLM training. MeshSlice has a working implementation for TPUs and we also present the simulated MeshSlice evaluation on large TPU clusters.

PowerGrad PowerGrad is a hierarchical power management framework that tries to maximize the performance of a cluster in a power-limited environment. PowerGrad proposes mathematical methods to estimate the performance gradients using runtime hardware counter measurements. PowerGrad leverages the estimated performance gradients for its

gradient-based control, whose structure is scalable in a hierarchical manner. PowerGrad is implemented in software for CPU clusters. It improves the power efficiency of ML inference clusters running different types of ML inference workloads.

FriendlyFoe FriendlyFoe is an architectural defense framework against side channel attacks. It is the first AML-based on-the-fly architectural side channel defense. FriendlyFoe presents the Defender architecture that is designed to maximize the security with low implementation cost. FriendlyFoe trains the Defender in a GAN structure to maximize its transferability to various adaptive attackers. FriendlyFoe can be applied to systems to thwart various side channel attacks. The evaluations show the efficacy and practicality of FriendlyFoe for two side channels: memory-contention side channel and application power side channel.

CHAPTER 2: MESHSLICE: EFFICIENT 2D TENSOR PARALLELISM FOR DISTRIBUTED DNN TRAINING

2.1 BACKGROUND

2.1.1 Distributed Training Methods

Large-scale DNNs, especially LLMs, are typically trained in distributed systems of computing devices. In this work, we will refer to a computing device as a chip, which can be an ML accelerator, a GPU, or a CPU. There are three major types of parallelism when distributing DNN computations among chips: data, pipeline, and tensor parallelism. To maximize the scalability of training, contemporary LLM training methods form 3D networks of chips by leveraging all three types of parallelism together [8, 18, 19, 50].

Data parallelism (DP) partitions the input data among different chips [13]. Because the DNN parameters (weights and biases) are replicated among the chips, the only communication happens when the parameters are updated and synchronized. During parameter update, the parameter gradients are reduced among all chips (AllReduce) so that every chip can perform stochastic gradient descent (SGD) with identical gradient values. The communication cost of DP can be effectively hidden because the parameter update communication of one layer can be done in parallel with the computation of another layer.

Pipeline parallelism (PP) gives different DNN layers to different chips [15]. The communication happens only at the boundary between pipeline stages. PP incurs the least communication traffic among the three types of parallelism, but its scalability is inherently limited by the network structure of the DNN model. The overhead of PP increases with the number of pipeline stages.

Tensor parallelism (TP) partitions all matrices (weight, input, and output) of a DNN layer among different chips. Because all matrices are partitioned, TP requires the least memory footprint, but incurs the most communication traffic out of the three types of parallelism. TP generates communication traffic in every GeMM computation. In LLM training, this communication traffic is generated by the fully-connected (FC) layers [16].

In 1D TP GeMMs, the weight matrix is partitioned by either its input or output dimension into a 1D network of chips. In the former case, the input and weight shards are multiplied to compute the partial outputs, and the partial outputs are accumulated via a ReduceScatter (RdS) communication. In the latter case, the input shards are collected from all chips via an AllGather (AG) communication, and are then multiplied by the weight shards.

Figure 2.1 illustrates the functionalities of the AllGather and ReduceScatter operations.

AllGather copies each local shard (in0 to in3) to all other ranks (chips). As a result, every chip holds the same copy of the gathered shards. ReduceScatter reduces the shards in all the ranks (in0 to in3), and the reduced shard is scattered to the ranks. Hence, each rank holds a part of the reduced output (out0 to out3).

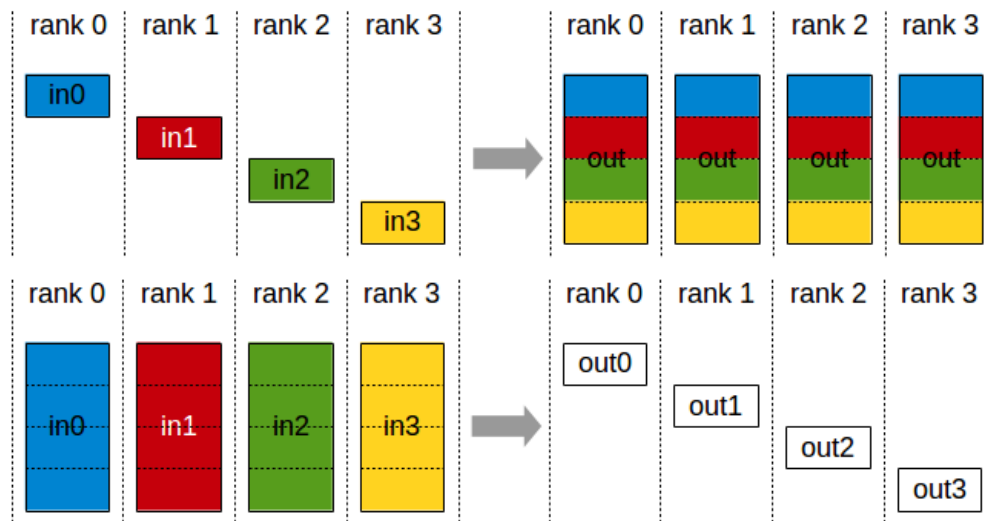


Figure 2.1: Illustrations of AllGather (top) and ReduceScatter (bottom) collective communication operations. [51]. AllReduce is the combined operation where an AllGather is followed by a ReduceScatter.

Unfortunately, the scalability of 1D TP is limited because the communication traffic grows linearly with the number of chips. This is because either the output shard must be accumulated across all chips during RdS, or the input shard must be copied to all chips during AG. Therefore, to achieve linear performance scaling with 1D TP, the communication bandwidth must scale quadratically with the number of chips. One example solution is NVIDIA’s NVSwitch [52], which connects 8 GPUs in a fully-connected ICI network. This approach has limited scalability because it becomes quadratically harder to build a fully-connected switch with a larger number of GPUs. Consequently, most 3D LLM training clusters limit TP to 8-way [8, 11, 18, 50].

2.1.2 2D Tensor Parallelism

A more scalable solution for TP is 2D TP, where the matrices are partitioned among chips organized in a 2D mesh (connected as a 2D torus). Each chip locally holds one shard of each matrix. Then, the FC layers use a 2D distributed GeMM algorithm.

In 2D GeMM, each shard is communicated only to the chips in the same row or in the same column—unlike in 1D distributed GeMM, where a shard is communicated to all other

chips. This is possible because computing one element of the output matrix only depends on one row of the left input matrix and one column of the right input matrix. Therefore, 2D TP incurs less communication traffic than 1D TP. As a result, 2D TP is more scalable for a given communication cost, or has lower communication cost for a given number of chips. Moreover, the hardware cost to build a large mesh is lower than the cost to build a large fully-connected network.

Figure 2.2 shows an example of 2D GeMM in a 3×3 device mesh connected via inter-chip interconnects (ICIs). Hereby, the matrices are sharded (A_{ij}, B_{ij}, C_{ij}) and stored in the local device memories. One of the shards (A_{ij}) flows horizontally (i.e. inter-column direction), one of the shards (B_{ij}) flows vertically (i.e. inter-row direction), and the last shard (C_{ij}) remains stationary.

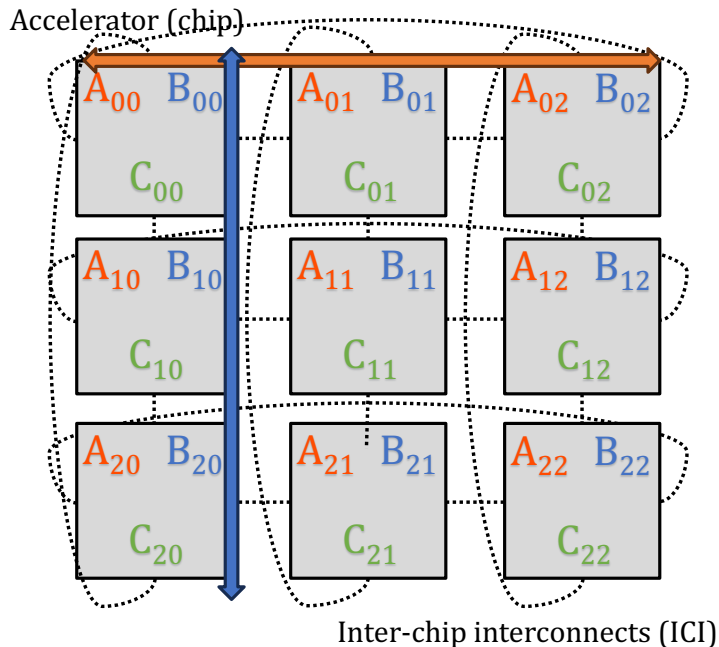


Figure 2.2: Overview of 2D distributed GeMM in 2D TP.

The extra scalability of 2D TP can be exploited in different ways. For example, Llama 3 is trained using a 16K GPU cluster exploiting 3D parallelism (DP+PP+TP) with only 8-way 1D TP [11]. If we leverage 128-way 2D TP instead of 8-way 1D TP, we can build a $16\times$ larger cluster with 256K chips. This not only attains a larger scale but also reduces the communication cost of DP. Indeed, because each chip now holds a shard of 1/128th of the weight matrix instead of 1/8th, the per-chip DP traffic is $16\times$ smaller than before.

Alternatively, we can keep the same total number of chips, apply 128-way 2D TP, and decrease the degrees of both DP and PP by $4\times$. In this cluster, the per-chip DP traffic is $64\times$

smaller than in the cluster using the 8-way 1D TP. As a result of the reduced communication cost and fewer pipeline stages, we can expect a better compute utilization than in the original cluster.

2.1.3 2D GeMM Algorithms

General Aspects. 2D GeMM is the heart of 2D TP in distributed DNN training. While there are a variety of 2D GeMM algorithms, they share many common aspects. Assume that we compute the output matrix $C = AB$ as the multiplication of the left input A and the right input B . All three matrices are partitioned in both their row and column dimensions into shards, and the shards are assigned to a 2D mesh of chips. That is, in a mesh with P_r rows and P_c columns, A is partitioned into the shards (i.e., sub-matrices) $A_{00} \dots A_{(P_r-1)(P_c-1)}$. Then, A_{ij} is stored in chip (i, j) at the i -th row and j -th column of the mesh. The same applies to B and C .

2D GeMM algorithms can compute GeMMs in three possible dataflows [22, 53] as illustrated in Figure 2.3. In each dataflow, the shards of one of the three matrices (A, B, C) remain *stationary* in their chips, and the shards of the other two are communicated through either the vertical (inter-row) direction or the horizontal (inter-column) direction.

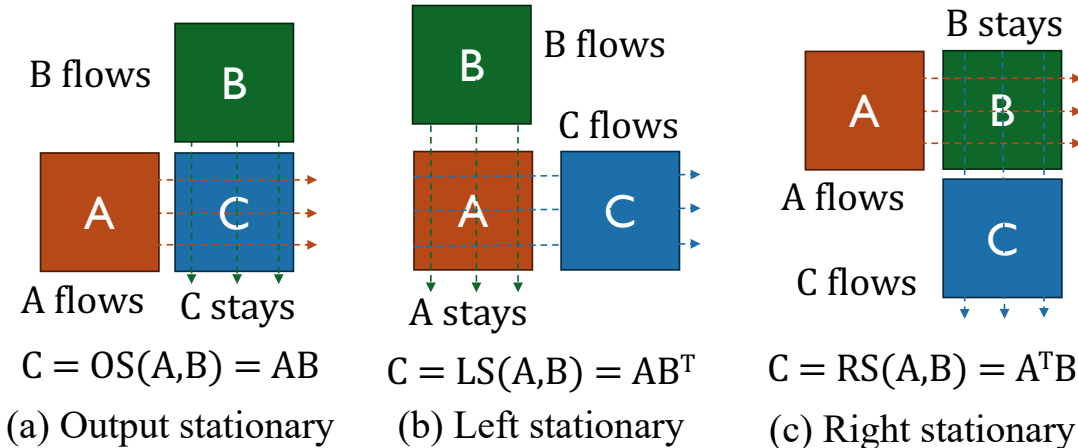


Figure 2.3: Three dataflows of 2D GeMM algorithms.

In the output-stationary (OS) dataflow (Figure 2.3a), the output C is stationary, each shard of A is transferred to the chips in the same row (inter-column), and each shard of B is transferred to the chips in the same column (inter-row). This results in $C = AB$. In the left-stationary (LS) dataflow (Figure 2.3b), the left input A is stationary, each shard of the right input B is transferred to the chips in the same column, and each shard of the output

C is transferred and accumulated into the chips in the same row. This results in $C = AB^\top$. Finally, the right-stationary (RS) dataflow (Figure 2.3c) is the symmetric version of the LS dataflow, resulting in $C = A^\top B$. LS and RS dataflows are equivalent to input-stationary and weight-stationary dataflows in systolic arrays [54], respectively.

The 2D GeMM communication traffic depends on the dataflow and the shape of the mesh. Assume that we have a mesh of P_r rows and P_c columns and two matrices M_r and M_c that flow in the inter-row direction (i.e., vertically) and in the inter-column direction (i.e., horizontally), respectively. Each matrix shard must be communicated to all other chips in either its row or its column. We can compute the time taken to transfer the shards (excluding synchronization and other overheads) as follows. For the inter-row (i.e., vertical) transfers, the time is $(P_r - 1) \times \text{sizeof}(M_r) / (P_r \times P_c) / BW_{row}$; for the inter-column (i.e., horizontal) transfers, it is $(P_c - 1) \times \text{sizeof}(M_c) / (P_r \times P_c) / BW_{col}$. Here, BW_{row} is inter-row link bandwidth, and BW_{col} is inter-column link bandwidth. We refer to these times as the *traffic costs*. The 2D GeMM traffic cost is the maximum of the inter-row and inter-column traffic costs, as we need to wait until the communications in both directions complete.

If $BW_{row} = BW_{col}$, the traffic cost is minimized when $(P_r - 1) / (P_c - 1) = \text{sizeof}(M_c) / \text{sizeof}(M_r)$. However, the values of P_r and P_c that minimize the traffic cost may not minimize the overall communication cost due to synchronizations and other overheads.

Cannon’s Algorithm. Cannon’s algorithm [21] is one of the first 2D GeMM algorithms. It only works for square meshes. Before the computation begins, the matrix shards are shifted in a skewed manner. Then, Cannon systolically shifts the shards (using SendRecv communication operations) while computing the partial multiplications. Cannon is the base algorithm for systolic arrays [55] and 3D/2.5D GeMM algorithms [56, 57].

The major limitation of Cannon is that it has a higher traffic cost than other 2D GeMM algorithms for two reasons. First, skewing the matrix shards at the beginning incurs extra communication traffic that is not required in other algorithms. Second, while different mesh shapes change the traffic cost of 2D GeMM algorithms, Cannon only works for square meshes. Hence, Cannon incurs a higher traffic cost than other 2D GeMM algorithms when the matrix shapes are significantly imbalanced.

SUMMA Algorithm. SUMMA [22] solves the two limitations of Cannon in that it does not need to skew the matrix shards and it can support any mesh shapes. The pseudocode of SUMMA for different dataflows is shown in Figure 2.4a. SUMMA splits the matrices into $P \times P$ shards, where P is a common multiple of P_r and P_c . Then, it performs communications and computations in a loop of P iterations. SUMMA uses broadcast (bcast) and reduce communication operations on shards across the same row or column. Because each row or

column of a 2D torus is connected in a ring topology, SUMMA runs ring bcast and reduce algorithms.

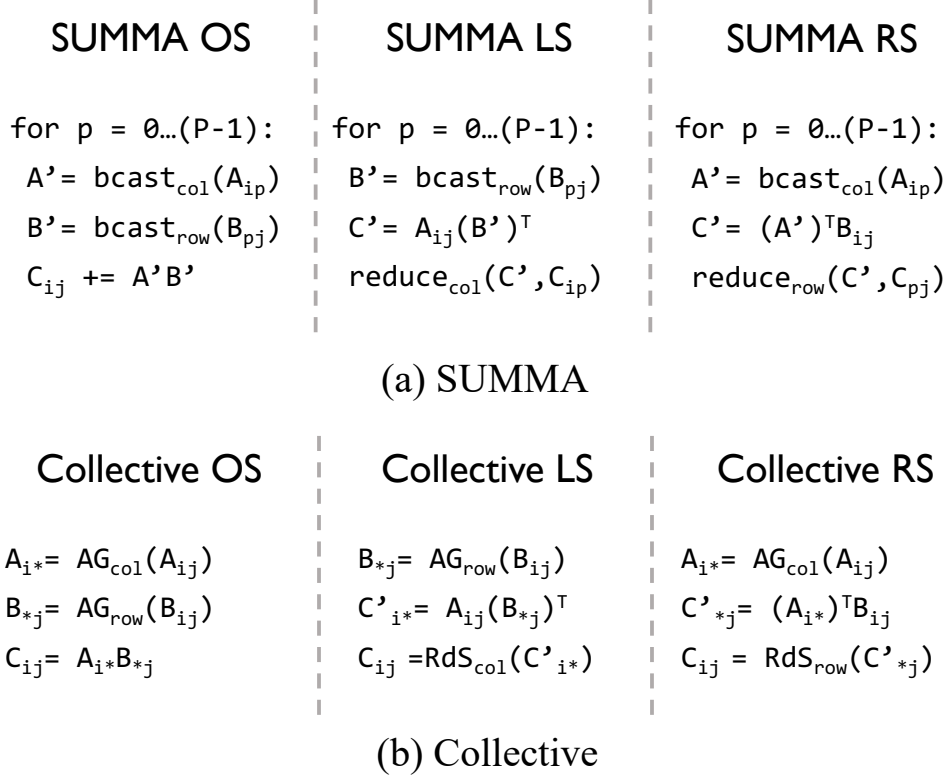


Figure 2.4: Pseudocode of SUMMA and Collective 2D GeMM algorithms for the three dataflows. In the code, expressions with *row* as subscript are inter-row communications in the same column, while those with *col* are inter-column communications in the same row. A_{ij} is a shard of A located in the chip at i -th row and j -th column of the mesh.

For example, assume that we run the SUMMA LS algorithm (Figure 2.4a, center) in a $P \times P$ mesh. A_{ij} is a shard of A located in the chip at the i -th row and j -th column of the mesh. In the p -th iteration, the following occurs. First, the chips (p, j) for all j broadcast their B_{pj} shards to the chips in the j -th column. Then, in all chips, the local A_{ij} shard is multiplied with the transpose of B_{pj} , producing the partial result C' . Finally, for each row i , the partial results C' in all the chips in the row are reduced to the shard C_{ip} in the chip (i, p) in the p -th column.

Unfortunately, SUMMA's one-to-all bcast and all-to-one reduce communications are inefficient in a large mesh connected with high-bandwidth network links. Consider a bcast, shown in the left part of Figure 2.5. To utilize all the links in a ring (a row or column) during the transfer, the shard to be broadcasted is broken down into D packets that are streamed over the ring as fine-grain transfers. The streaming is done in $P + D - 1$ pipeline stages.

There are two sources of overhead: bubbles in the pipeline and synchronizations. Each link suffers $P - 1$ bubbles—some at the beginning and some at the end of the transfer. Further, each pipeline stage requires a synchronization and therefore the broadcast needs $P + D - 1$ synchronizations. The reduce operation has the same communication pattern and suffers from the same overheads. Since there are P iterations in SUMMA, the total synchronization overhead grows as $O(P^2)$.

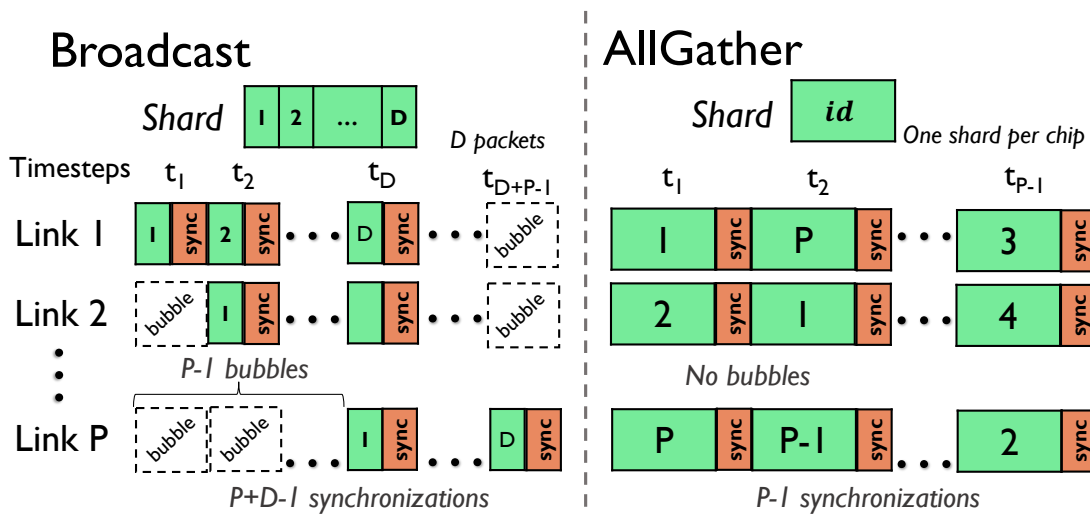


Figure 2.5: Communication patterns of Broadcast and AllGather operations in a P -chip ring.

The pipeline stalls and synchronization overheads were negligible in the 1990s, when supercomputers only had a few hundred cores with low bandwidth interconnects. However, contemporary ML training clusters have high-bandwidth ICIs and thousands of manycore accelerators. As a result, the pipeline stalls and synchronizations have become significant overheads.

Collective 2D GeMM. To avoid the SUMMA’s overheads, a popular approach is to perform the 2D GeMM using the AllGather (AG) and ReduceScatter (RdS) [23, 58] collective communication operations. AG involves the parallel execution of all the per-column or per-row bcasts. Likewise, RdS is the parallel execution of all the per-column or per-row reduce operations. We call this approach *Collective 2D GeMM*.

For each dataflow of the SUMMA algorithm, there is a Collective 2D GeMM counterpart [58]. The algorithms for the three dataflows are shown in Figure 2.4b. For example, in the LS dataflow (Figure 2.4b, middle), multiple $bcast_{row}$ operations in SUMMA LS (Figure 2.4a, middle) are merged into a single AG_{row} operation, and multiple $reduce_{col}$ operations in SUMMA LS are merged into a single RdS_{col} operation. The computation is also done in

a single step rather than in P iterations as in SUMMA.

Using collective AG/RdS communications solves the inefficiency problems of the broadcast/reduce operations. Consider the AG operation in a ring (a row or a column) of P chips, as shown in the right part of Figure 2.5. In each of the $P - 1$ steps, each link transfers an entire shard to a neighbor. Hence, compared to broadcast, AG eliminates pipeline bubbles, transfers larger packets, and invokes fewer synchronizations. The same is true for RdS compared to reduce, because they follow identical communication patterns with broadcast and AG, respectively. Moreover, since Collective 2D GeMM only calls AG or RdS once per each direction, its total synchronization overhead grows as $O(P)$. Hence, AG/RdS are more efficient and attain higher bandwidth utilization than broadcast/reduce.

The major limitation of Collective 2D GeMM is that it cannot *overlap* communications with computations. One cannot apply software pipelining to overlap them because there are no loop iterations, and there are true dependencies between the computation and the communication operations.

Wang et al. [24] present a partial solution to this problem by splitting the collective communication in one direction into multiple *SendRecv* communications. Then, by applying software pipelining, the SendRecv communications are overlapped with the partial GeMM computations. Such a 2D GeMM is equivalent to a combination of FSDP [14] and 1D TP. However, this solution can partition and overlap only the communication operation in one direction; it cannot overlap the communication operation in the other direction. To partition the AG/RdS operations into multiple SendRecv operations in both directions, one needs to use Cannon, whose limitations were discussed before.

2.2 PROBLEM ADDRESSED

In this chapter, we make two contributions to 2D TP. First, we propose a new 2D GeMM algorithm that solves the limitations of existing 2D GeMM algorithms. Second, we design an LLM autotuner that finds an efficient 2D TP configuration for LLM training. The LLM autotuner optimizes the configuration of the dataflow, mesh shape, and communication granularity.

Our proposed 2D GeMM algorithm is called MeshSlice. Figure 2.6 visualizes the timelines of the previous algorithms, and compares them to MeshSlice. The figure shows the time progression of the computation, inter-row communications, and inter-column communications. Cannon requires skewing and only supports square mesh shapes. As a result, it has higher traffic than the other algorithms, which increases overall execution time. SUMMA uses inefficient broadcast/reduce communication operations, which incur pipeline bubbles and

synchronization overhead due to fine-grained packets. Collective does not overlap the collective communications with computation. Wang’s algorithm partitions the collective communication in only one direction, so it leaves the communication in the other direction non-overlapped. Finally, MeshSlice is able to overlap the communication with computation in both directions and attains the fastest execution.

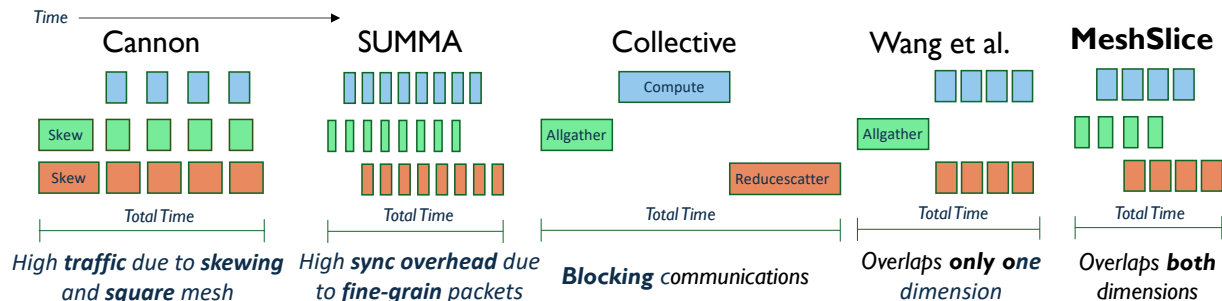


Figure 2.6: Comparing the timelines of five 2D GeMM algorithms: Cannon, SUMMA, Collective, Wang, and MeshSlice.

2.3 THE MESHSLICE 2D GEMM ALGORITHM

The MeshSlice algorithm has three characteristics. First, it partitions and overlaps communication operations in both directions. Second, it uses efficient AG and RdS communication operations instead of `bcast`, `reduce`, or `SendRecv` operations. Finally, it supports any mesh shape and a flexible granularity of communication.

To overlap communications with computations, we need to *partition* the collective communications into smaller communications and apply software pipelining. There are two existing methods to partition collective AG/RdS operations. The first one is to break them down into multiple `SendRecv` communications. Wang’s algorithm [24] applies this approach to a single direction, and applying it to both directions requires using Cannon’s algorithm [21]. The second method is to partition AG and RdS into multiple `bcast` and `reduce` operations, respectively. This approach results in SUMMA [22].

Instead, MeshSlice introduces a *new* partitioning method: partitioning AG and RdS operations into *partial* AG and RdS operations. The core of this algorithm is *slicing* the matrix shards into S sub-shards. In each iteration of an S -way loop, we apply a partial AG or RdS to a sub-shard, and compute a partial GeMM. The algorithm completes when all S sub-shards have been processed.

Figure 2.7 shows, for each of the three dataflows, the pseudocode of the MeshSlice algo-

rithm running in every chip (i, j) of the mesh. Each algorithm executes in a loop with S iterations. In here, we give the high-level intuition of the algorithms; in subsequent sections, we will explain the operations in detail. In the OS algorithm (Figure 2.7, left), for each $s = 0 \dots S - 1$, the following occurs. First, each chip uses $slice_{col}$ to slice its local matrix shard A_{ij} along the column dimension to fetch its local s -th sub-shard, A_s . Similarly, it uses $slice_{row}$ to slice its local B_{ij} along the row dimension to fetch its local s -th sub-shard, B_s . Then, each chip collects A_s sub-shards from all the chips in the same row and B_s sub-shards from all the chips in the same column using AG_{col} and AG_{row} operations, respectively. Finally, each chip computes the partial GeMM with the collected sub-shards and accumulates the result into its local output shard C_{ij} . Crucially, the partial GeMM operation in one iteration is overlapped with the AG and slicing operations in another iteration via software pipelining of the loop.

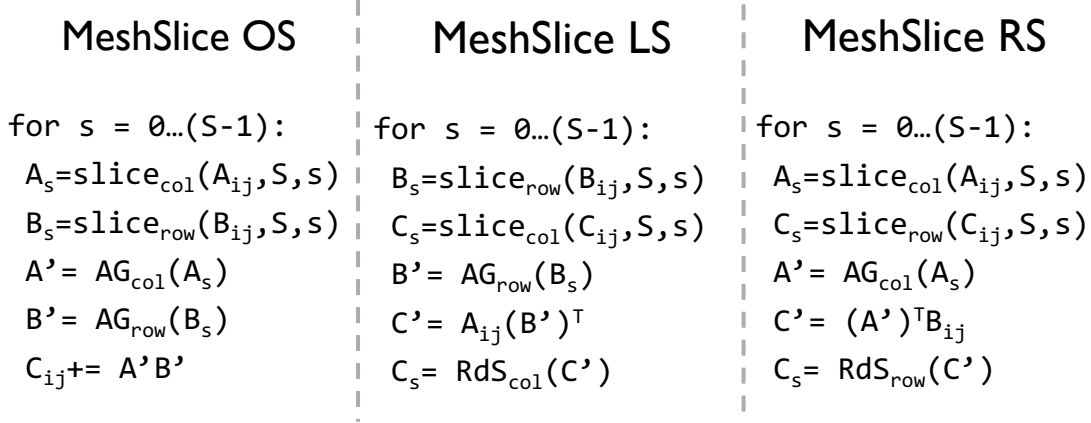


Figure 2.7: Pseudocodes of the MeshSlice 2D GeMM algorithm in the three dataflows.

We can apply a similar slicing method to LS and RS dataflows. For each iteration in the LS dataflow (Figure 2.7, center), B_{ij} and C_{ij} are sliced along their row and column dimensions, respectively. Then, the s -th sub-shards B_s in the chips in the same column are all-gathered to B' . Next, the partial multiplication result $C' = A_{ij}(B')^\top$ is computed. Finally, C' is reduce-scattered to the s -th sub-shards C_s in the chips in the same row. The RS dataflow (Figure 2.7, right) follows a similar flow.

Like Collective and SUMMA, MeshSlice can be applied to a mesh of any shape. Also, we can control the slice count S to adjust the granularity of communication. A small S (coarse granularity) results in a large non-overlapped prologue and epilogue during the software pipelining. A large S (fine granularity) reduces the size of the prologue and epilogue, but increases the total synchronization overhead by performing more communication operations. Given this trade-off, there are different optimal values of S for different 2D GeMM

configurations and hardware architectures.

The major challenge of MeshSlice is designing a correct and efficient slicing mechanism. This is not a trivial problem: most arbitrary slicings result in an incorrect computation. In the following, we describe how we implement the correct slicing mechanism.

Mathematical Description of the MeshSlice Algorithm

Assume that we are computing a 2D GeMM of $A \in \mathbb{R}^{M \times K}$ and $B \in \mathbb{R}^{K \times N}$ in OS dataflow on a mesh of shape $P_r \times P_c$. In each chip (i, j) , the computation result will be the shard $C_{ij} \in \mathbb{R}^{M/P_r \times N/P_c}$, which is the multiplication of $A_{i*} = AG_{col}(A_{ij}) \in \mathbb{R}^{M/P_r \times K}$ and $B_{*j} = AG_{row}(B_{ij}) \in \mathbb{R}^{K \times N/P_c}$. Figure 2.8 depicts A_{i*} and B_{*j} for a given i and j . Then, A_{i*} is broken down into K column vectors $\mathbf{a}_{i0}, \dots, \mathbf{a}_{i(K-1)} \in \mathbb{R}^{M/P_r}$, and B_{*j} is broken down into K row vectors $\mathbf{b}_{0j}, \dots, \mathbf{b}_{(K-1)j} \in \mathbb{R}^{1 \times N/P_c}$. The figure highlights and labels four of these vectors: \mathbf{a}_{i0} , $\mathbf{a}_{i(K-1)}$, \mathbf{b}_{0j} , and $\mathbf{b}_{(K-1)j}$. Finally, $C_{ij} = A_{i*}B_{*j}$ is equivalent to the sum of K outer products of the column and the row vectors as follows.

$$C_{ij} = \mathbf{a}_{i0}\mathbf{b}_{0j} + \dots + \mathbf{a}_{i(K-1)}\mathbf{b}_{(K-1)j} \quad (2.1)$$

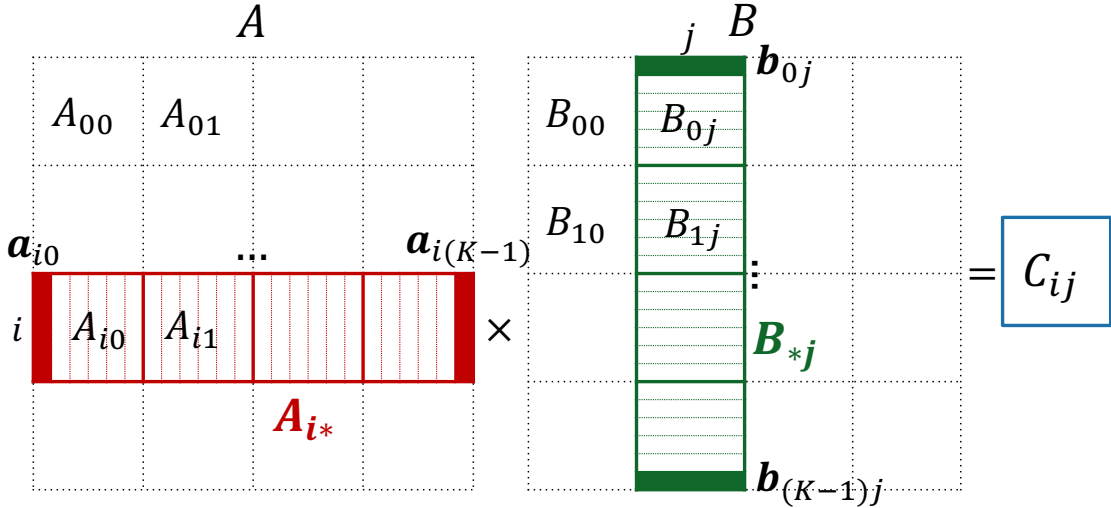


Figure 2.8: Generating an output shard in a 2D GeMM.

Our algorithm slices this computation in a loop with S iterations, where an iteration computes every S -th outer products. For $S=3$, Figure 2.9 shows the vectors accessed by the first iteration of the loop (in purple) and those accessed by the second iteration (in yellow). Algorithm 2.1 shows the algorithm. For instance, the first iteration accumulates

$\{\mathbf{a}_{i0}\mathbf{b}_{0j} + \mathbf{a}_{iS}\mathbf{b}_{Sj} + \mathbf{a}_{i2S}\mathbf{b}_{2Sj} + \dots\}$ to the output shard C_{ij} . The column vectors accessed by this iteration $\{\mathbf{a}_{i0}, \mathbf{a}_{iS}, \dots\}$ are gathered using AG from the shards $\{A_{i0}, A_{i1}, \dots\}$ in the chips at the i -th row. Likewise, the row vectors $\{\mathbf{b}_{0j}, \mathbf{b}_{Sj}, \dots\}$ are gathered from the shards $\{B_{0j}, B_{1j}, \dots\}$ in the chips at the j -th column.

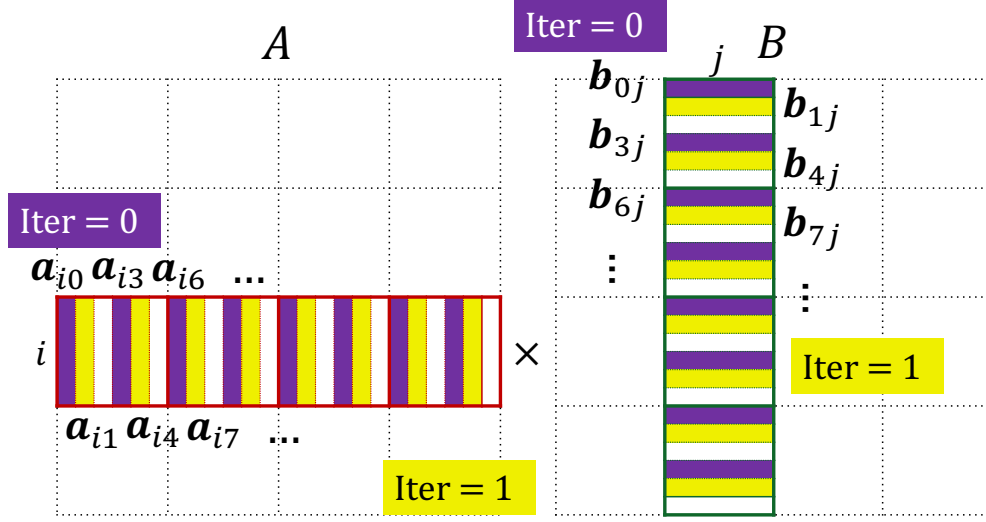


Figure 2.9: Slicing the shards with $S = 3$ in MeshSlice.

Algorithm 2.1 S -way sliced GeMM algorithm to compute C_{ij}

for $s = 0 \dots S - 1$ **do**

$$C_{ij} += \mathbf{a}_{is}\mathbf{b}_{sj} + \mathbf{a}_{i(s+S)}\mathbf{b}_{(s+S)j} + \mathbf{a}_{i(s+2S)}\mathbf{b}_{(s+2S)j} + \dots$$

Detailed Implementation of the MeshSlice Algorithm

In this section, we describe the MeshSlice algorithm presented in Figure 2.7 in detail, and show that its partial GeMM computation is identical to the sliced GeMM computation in Algorithm 2.1. To begin with, note that A_{ij} contains K/P_c column vectors of A_{i*} and B_{ij} contains K/P_r row vectors of B_{*j} as follows:

$$A_{ij} = [\mathbf{a}_{i(j \times K/P_c)}, \mathbf{a}_{i(j \times K/P_c + 1)}, \dots, \mathbf{a}_{i((j+1) \times K/P_c - 1)}] \quad (2.2)$$

$$B_{ij} = \begin{bmatrix} \mathbf{b}_{(i \times K/P_r)j} \\ \vdots \\ \mathbf{b}_{((i+1) \times K/P_r - 1)j} \end{bmatrix} \quad (2.3)$$

At the s -th iteration of the MeshSlice OS algorithm (Figure 2.7, left), applying $slice_{col}$ to A_{ij} collects every S -th column vectors in A_{ij} , and applying $slice_{row}$ to B_{ij} collects every S -th row vectors in B_{ij} . We call A_s and B_s the s -th sub-shards of A_{ij} and B_{ij} , respectively.

$$A_s = slice_{col}(A_{ij}, S, s) = [\mathbf{a}_{i(j \times K/P_c + s)}, \mathbf{a}_{i(j \times K/P_c + s + S)}, \dots] \quad (2.4)$$

$$B_s = slice_{row}(B_{ij}, S, s) = \begin{bmatrix} \mathbf{b}_{(i \times K/P_r + s)j} \\ \mathbf{b}_{(i \times K/P_r + s + S)j} \\ \vdots \end{bmatrix} \quad (2.5)$$

If we AllGather (AG_{col}) the A_s sub-shards from all the chips in the same row of the mesh, and AllGather (AG_{row}) the B_s sub-shards from all the chips in the same column, we obtain the following A' and B' matrices.

$$A' = AG_{col}(A_s) = [\mathbf{a}_{is}, \mathbf{a}_{i(s+S)}, \mathbf{a}_{i(s+2S)}, \dots] \quad (2.6)$$

$$B' = AG_{row}(B_s) = \begin{bmatrix} \mathbf{b}_{sj} \\ \mathbf{b}_{(s+S)j} \\ \mathbf{b}_{(s+2S)j} \\ \vdots \end{bmatrix} \quad (2.7)$$

These \mathbf{a} and \mathbf{b} vectors are those shown in Figure 2.9. Then, computing $C_{ij} += A'B'$ is mathematically identical to computing the s -th iteration of Algorithm 2.1.

Our slicing operation may result in non-contiguous memory accesses. For example, the $slice_{col}$ operation accesses column vectors $\mathbf{a}_{is}, \mathbf{a}_{i(s+S)}, \dots$, which are not contiguous in memory. This is inefficient in most memory subsystems. Therefore, we further optimize the slicing operations ($slice_{col}$ and $slice_{row}$) so that the $\mathbf{a} \in \mathbb{R}^{M/P_r}$ column vectors become matrices $\mathbf{a} \in \mathbb{R}^{M/P_r \times B}$, where B is an architecture-dependent block size (e.g., a cache line size). At the same time, the $\mathbf{b} \in \mathbb{R}^{1 \times N/P_c}$ row vectors become matrices $\mathbf{b} \in \mathbb{R}^{B \times N/P_c}$. This design ensures contiguous memory accesses.

As an example, the blocked column slicing algorithm $slice_{col}$ is shown in Algorithm 2.2, where R and C are the dimensions of a local shard. The block size B is determined by the hardware architecture. For instance, since a TPU accesses its memory via 2D 128×8 chunks [59], we set $B = 8$ for TPUs. The user can then choose any slice count S from the divisors of C/B .

Algorithm 2.2 Blocked column slicing algorithm

```
function slice_col( $X : \text{matrix} \langle R, C \rangle, S : \text{int}, s : \text{int}$ )  
     $B \leftarrow$  block size for efficient memory access  
    // Splits  $X$  by  $B$  contiguous columns  
     $X' \leftarrow X.\text{reshape}(\langle R, C/SB, S, B \rangle)$   
    // The  $s$ -th sub-shard has  $\langle R, C/S \rangle$  elements  
    return  $X'[:, :, s, :].\text{reshape}(\langle R, C/S \rangle)$ 
```

2.4 THE MESHSLICE LLM AUTOTUNER

The MeshSlice 2D GeMM algorithm has several parameters that determine its communication cost and efficiency. First, the shape of the mesh determines the traffic cost as discussed in Section 2.1.3. Next, there are numerous ways to partition the matrix into shards for the chips in the mesh (i.e., the sharding), and a different partitioning changes the 2D GeMM dataflow. Finally, we need to determine the slice count S of MeshSlice, which affects the synchronization overhead and the communication overlapping.

Finding an optimal configuration of these parameters is a hard problem, and usually relies on human inputs [23]. Manually finding the optimal parameters needs expert knowledge of the system architecture and expensive trial-and-error.

To address this problem, we design the *MeshSlice LLM autotuner*, which can find efficient parameter configurations for an LLM training. The inputs to the autotuner are the LLM architecture, the training hyperparameters (e.g., batch size and input sequence length), and the possible 2D mesh shapes of the cluster of chips. The autotuner runs in two phases. First, it determines efficient dataflows for the FC layers and the shardings of the LLM tensors. Next, it jointly optimizes the mesh shape of the cluster and the slice count S of each FC layer, using analytical cost models.

Phase 1: Dataflow and Sharding

The first optimization problem of 2D TP is choosing the right *sharding*. Given a mesh, a sharding of a tensor is a mapping from the mesh dimensions to the tensor dimensions that are to be partitioned. For example, given a 2D mesh and a 4D tensor, there are $4P2 = 12$ possible shardings, since we need to choose two tensor dimensions to split among the rows and columns of chips, respectively.

Automatically finding the optimal sharding is difficult for two reasons. First, there are many choices available. For example, consider a 2D GeMM of two 4D tensors. Since we have two input tensors and one output tensor, there are $(4P2)^3 = 1728$ possible sharding

combinations. Next, estimating the performance for each sharding combination is hard. A change in one sharding may induce a different dataflow, require a possible re-sharding due to transposition, and result in a different compute efficiency. To make matters worse, we cannot just consider the shardings of the forward computations, but also the shardings of the backpropagation computations during training.

Choosing a dataflow The MeshSlice LLM Autotuner approaches this problem in the opposite direction: the autotuner chooses the dataflow first and then determines the sharding of each tensor. Take a 2D GeMM $Y = XW$, which multiplies the input X and the weight W to compute the output Y . There are three 2D GeMM dataflows for the forward pass computation, as shown in the *Forward* column of Table 2.1. The autotuner chooses the dataflow that makes the largest matrix of the three remain stationary. For instance, if the output Y is the largest matrix, the choice is Y -stn.

Table 2.1: Three dataflows for the 2D GeMM $Y = XW$. Each dataflow makes Y stationary (Y -stn), X stationary (X -stn), or W stationary (W -stn).

Dataflow	Forward	Backward Data	Backward Weight
Y -stn	$Y = OS(X, W)$	$X' = LS(Y', W)$	$W' = RS(X, Y')$
X -stn	$Y = LS(X, W^\top)$	$X' = OS(Y', W^\top)$	$W'^\top = RS(Y', X)$
W -stn	$Y = RS(X^\top, W)$	$X'^\top = LS(W, Y')$	$W' = OS(X^\top, Y')$

To train a DNN layer, there are two passes of computation: forward pass to compute the outputs, and backward pass to compute the gradients for backpropagation. A forward pass of $Y = XW$ generates two backward pass computations, namely *backward data* and *backward weight*. The backward data computation computes the input gradient $X' = Y'W^\top$ as a multiplication of the output gradient Y' and the transpose of W . The backward weight computation computes the weight gradient $W' = X^\top Y'$ by multiplying the transpose of X and Y' . The compute and communication demands of each of the backward data and the backward weight computations are almost identical to those of the forward pass.

Given the forward pass dataflow, the autotuner chooses the dataflows for the backward pass computations from the same row of Table 2.1. This ensures the following properties. First, the largest matrix remains stationary in all three (forward and two backward) computations. Also, each matrix and its gradient matrix flow in the same direction in all three computations. Finally, none of the matrices needs to be transposed to compute the backward pass.

As an example, assume we choose the X -stn dataflow for the forward pass ($Y = LS(X, W^\top)$).

Hereby W^\top is statically transposed during its initialization. In the forward computation, Y flows horizontally, and W^\top flows vertically. The dataflow for backward data computation is $X' = OS(Y', W^\top)$. As with the forward pass, X' is stationary, Y' flows horizontally, and W^\top flows vertically. The dataflow for backward weight computation is $W'^\top = RS(Y', X)$, which makes X be stationary, Y' flow horizontally, and W'^\top flow vertically. Overall, *across all computations*, X and X' remain stationary as they are the largest matrices, Y and Y' flow horizontally, and W^\top and W'^\top flow vertically.

For each row of Table 2.1, we can transpose all the matrices and flip the dataflow directions of the two non-stationary matrices to obtain the corresponding *transposed* dataflow. For example, the transposed version of $Y = OS(X, W)$ is $Y^\top = OS(W^\top, X^\top)$. Therefore, for each layer, there are two dataflow choices (non-transposed and transposed) that make the largest matrix stationary. Finding the absolute optimal dataflow choices for an L -layer neural network is a search problem with 2^L possibilities. Hence, the MeshSlice LLM autotuner uses a simple heuristic: for each layer, choose the non-transposed dataflow as a default, unless the layer’s input X needs to be transposed to keep the non-transposed dataflow. In most LLMs, this heuristic eliminates transpositions between the layers.

Sharding Once the autotuner chooses the dataflows, the shardings of the three matrices are automatically determined. The matrix rows are sharded among the rows of chips, and the matrix columns are sharded among the columns of chips.

In practice, LLMs use 4-D tensors of shape (B, S, H, D) , where B is the batch size, S is the sequence length, H is the number of attention heads, and D is the per-head hidden dimension. In an FC layer, the 4D tensor is reshaped into a 2D matrix of $(B \times S, H \times D)$ dimensions. MeshSlice follows the simple principle of partitioning the two outer-most dimensions of a 2D matrix. Hence, the B dimension of the 4D tensor is sharded among the mesh rows, and the H dimension is sharded among the mesh columns.

Besides the FC layers, an LLM network has many other operations. The shardings of these other operations have minimal performance impact as they incur no communication traffic [16]. Therefore, once the shardings of the FC layer tensors are determined, we let other tensors follow the same shardings to avoid resharding traffic.

Phase 2: Mesh Shape and Slice Count

This phase configures the two remaining parameters of MeshSlice: the mesh shape and the slice count. To this end, we design analytical cost models that estimate the GeMM execution time for each configuration. Then, we use these cost models to co-optimize the

two parameters.

We construct an analytical cost model of the communication from offline measurements of the synchronization latency, network bandwidth, and communication operation launch overheads in a small ML accelerator cluster. The cost of a collective communication operation is defined as follows:

$$cost_{op} = t_{launch} + (P - 1) \times (t_{sync} + sizeof(shard)/bw) \quad (2.8)$$

Hereby t_{launch} is the overhead of operation launch, P is the number of chips in the row or column, t_{sync} is the synchronization latency, $sizeof(shard)$ is the size of a shard to be transferred, and bw is the measured bandwidth of a link. This linear model fits well for the AG/RdS communications on a row or column ring. This is because in an AG or RdS on a ring (Figure 2.5, right), the shard transfers are synchronized and there is no network contention.

To estimate the compute times, our analytical compute cost model divides the total FLOP count of the local GeMM by the effective FLOPS throughput of the ML accelerator. The effective FLOPS is measured by benchmarking a few GeMM operations on a single accelerator chip. This computation model is accurate enough for LLM training because most GeMMs in LLM training are large enough to fully saturate the compute throughput of ML accelerators. For better accuracy, one can measure the compute execution time in a single accelerator chip instead of using the analytical model.

For each FC layer, the autotuner breaks down the compute plus communication execution time of the MeshSlice algorithm into three parts: prologue, steady-state, and epilogue. The prologue and epilogue are the operations in the first and last loop iterations, respectively, that cannot be overlapped by software pipelining. For instance, in our OS algorithm of Figure 2.7, the two all-gather operations in the first iteration form the prologue, and the partial GeMM computation in the last iteration forms the epilogue. We assume that a communication in one direction can execute in parallel with a communication in the other direction. Hence, the prologue time is time of the longest of the two AG operations, and the steady-state time per iteration is the time of the longest of three operations: the two AG operations and the partial GeMM. The total estimated execution time of the S iterations is: prologue + $(S - 1) \times$ steady-state per iteration + epilogue.

Using the cost models, the autotuner co-optimizes the mesh shape of the cluster and the slice count S_i of each FC layer i using an exhaustive search. For each possible mesh shape, the autotuner tunes S_i for each FC layer i by searching through all possible S values. Since the optimal S_i values of the different FC layers are independent of each other, the autotuner

optimizes the S_i value one layer at a time. Finally, it picks the configuration with the shortest execution time. The search space for the mesh shape and the slice counts is small because there are only a few possible integer choices for them. Therefore, the autotuner finishes in a few seconds thanks to the small search space and the simple analytical cost models.

2.5 EVALUATION SETUP

2.5.1 Simulation Setup

Our target ML accelerator to evaluate is Google’s TPU [55] because TPUs can build a 2D torus cluster connected with ICI links. Hence, we have built an implementation of MeshSlice on Jax [60] that runs on TPU instances in Google Cloud. Hereby we use Jax’s *shard_map* feature to partition the computations into the mesh, and the *dynamic_slice* operation to implement MeshSlice’s blocked slicing algorithm. This implementation does not overlap computations with communications because Google only supports asynchronous (i.e., overlapped) communication for SendRecv operations and not for the AG and RdS operations needed by MeshSlice yet.

For this reason, for the most part, we evaluate our algorithm by simulating clusters of TPUs connected with 2D torus topologies of different sizes and shapes. Figure 2.10 shows the architecture of the simulated TPU, which models Google’s TPUv4 [25]. A node in the cluster consists of a TPU and a host. The TPU has two cores and a network interface controller (NIC) that share an HBM memory. Each core has a 64MB scratchpad memory and four 128×128 systolic arrays to compute matrix multiplications. The NIC is connected to a router with four ICI links that connect the TPUs in a 2D torus network. The NIC processes direct communications between the TPUs. It can directly read from and write to the HBM memory. The TPU cores and the NIC are controlled by the host and can execute in parallel. The only performance interference between the cores and the NIC comes from any contention for the shared HBM memory.

We customize the SST simulator [61] to simulate the cluster and network architecture. We modify the SST’s rdmaNic simulator to simulate the NIC. The modified rdmaNIC simulator implements one controller per each ICI link, so that the four ICI links can execute communication operations in parallel. The HBM memory is simulated using DRAMSim3 [62].

To model the TPU cores, we implement a custom accelerator in SST. We focus on accurately modeling their memory access behavior, to be able to capture the memory contention between the cores and the NIC. The simulated TPU takes a GeMM ($C = AB$) request from the host and breaks it down into tiled sub-matrix multiplications. An output tile C_t is

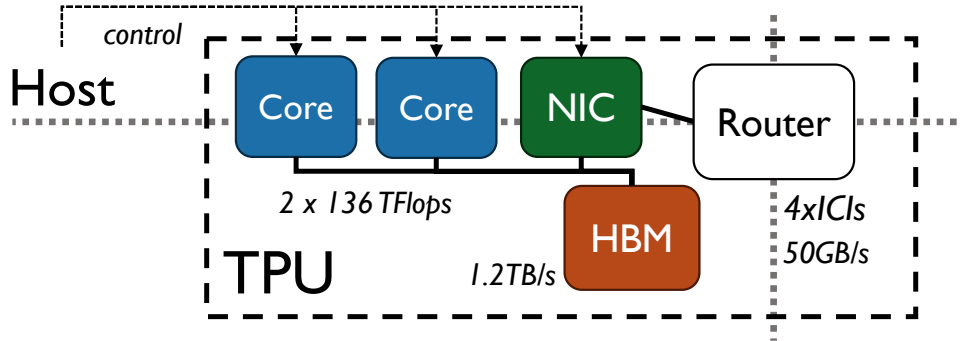


Figure 2.10: Architecture of the simulated TPU.

computed in a loop, where each iteration prefetches the input tiles A_t and B_t from HBM to the scratchpad memory, and computes $C_{t+} = A_t B_t$. We apply software pipelining in that the prefetches overlap with the multiplications. Once the loop finishes, C_t is written back to the HBM memory and the core transitions to the next tile.

We use the Google Cloud [63] to benchmark four real TPUv4 chips connected in a ring, to calibrate the parameters of our simulator (e.g., frequency, bandwidth, latency, and tile size). We profile the communication operations to configure the NIC and the router. The simulator is calibrated until the difference in benchmark execution time between the simulation and the real hardware is less than 10%.

2.5.2 2D GeMM Implementations

We evaluate the MeshSlice algorithm against four 2D distributed GeMM baselines: Cannon [21], SUMMA [22], Collective 2D GeMM (Collective) [58], and Wang’s algorithm (Wang) [24]. All 2D GeMM algorithms are executed in two parts: (i) prologue plus epilogue, which execute the operations that are not overlapped, and (ii) a single iteration of the steady state, where communications are overlapped with computations.

As depicted in Figure 2.6, for MeshSlice and SUMMA, the prologue plus epilogue are the parts of the first and last iterations that are not overlapped, and the steady state is the execution of the remaining loop iterations. In Cannon, the prologue is the skewing communication that shifts the matrix shards prior to the computation. There is no epilogue, and the steady state is the execution of the iterations, which overlap computations with SendRecv communications. For Collective, the prologue plus epilogue are the two collective communication operations, and the steady state is the execution of the GeMM without communication. For Wang, the prologue or the epilogue is the execution of a communication operation that is not overlapped, and the steady state is the execution of the iterations, which

overlap computations with SendRecv communications.

For each algorithm, the optimal mesh shape is different because the communication pattern is different. Hence, for fairness, we compare the performance with optimal mesh shapes for each algorithm.

Wang et al. [24] apply loop unrolling to their algorithm to have a smaller iteration count. This helps the computational efficiency by merging small GeMMs into larger GeMMs. Hence, we apply loop unrolling to SUMMA and Wang, as they have large iteration counts. We set the loop iteration counts of both algorithms to be the slice count of MeshSlice given by our LLM autotuner.

2.5.3 1D Distributed GeMM Baselines

While MeshSlice is a method for 2D TP, we also compare it against two 1D baselines. The first one is 1D TP following the Sequence Parallelism method [16], which is the most popular method to apply TP in LLMs. The second 1D baseline is Fully-Sharded Data Parallelism (FSDP) [14], which is a type of DP. FSDP is often considered an alternative to TP, as it also partitions the weight matrix into multiple shards and collects the shards right before the computation. It has a memory footprint similar to 1D TP.

The 1D baselines are simulated on a ring of TPU chips. Hence, each TPU is connected to only two ICI links. This halves the total bandwidth compared to a 2D mesh with the same number of chips. For both 1D TP and FSDP, we overlap communications with computations using Wang et al.’s method [24]—i.e., their AG or RdS communications are broken down into multiple SendRecv communications that are overlapped with partial GeMMs.

2.5.4 Target LLMs

We evaluate the training performance of two LLM models: OpenAI’s GPT-3 [7] and NVIDIA’s Megatron-NLG [8]. GPT-3 has 175B parameters and is the most popular LLM. Megatron-NLG has 530B parameters and requires a larger-scale distributed training.

LLMs follow the Transformer [12] DNN architecture, which consists of multiple stacks of identical neural network blocks. Each block consists of two sub-networks: multi-head attention and feed-forward. There are four FC layers in a block: two in the multi-head attention and two in the feed-forward network.

For the different 2D GeMM algorithms, only the FC layers have different implementations, while the other layers remain the same. Therefore, to compare the different GeMM algorithms, we only evaluate the FC layers. The other layers are benchmarked with TPUv4

chips in Google Cloud. They can be benchmarked with a single TPU because they do not incur communication cost: they are executed independently in each TPU chip [16]. The execution times of the FC layers and the other layers are combined to estimate the end-to-end performance of LLM training.

2.5.5 Building Analytical Cost Model

The communication cost model is a linear function with three parameters: bw , t_{sync} , and t_{launch} (Section 2.4). We benchmark the collective communication operations in 2-chip and 4-chip TPUv4 clusters with shard sizes ranging from 8KB to 512MB. t_{sync} is set by comparing the execution times with different numbers of chips. bw and t_{launch} are found with a linear regression on the execution times with different shard sizes.

The computation cost model only requires the effective compute throughput of the TPU, which can be measured by profiling a few GeMM operations on TPUv4.

The detailed implementation of the cost models in the MeshSlice LLM autotuner are presented in the source code repository¹, together with the TPU implementation of MeshSlice.

2.6 EVALUATION RESULTS

In this section, we evaluate the performance of LLM training using different distributed GeMM algorithms (Section 2.6.1), the autotuner and the cost models (Section 2.6.2), and MeshSlice running on a small real TPUv4 cluster (Section 2.6.3).

2.6.1 Distributed GeMM Algorithm Performance

Weak Scaling Performance

We compare the MeshSlice algorithm to four 2D baselines (Cannon, SUMMA, Collective, and Wang) and two 1D baselines (1D TP and FSDP). We start by evaluating the performance under weak scaling, which is the most common scenario in distributed DNN training. This is because adding more chips to a cluster results in more available memory, which enables an increase in the input batch size for faster training. We set the batch size to half the number of chips in the cluster, and the input data sequence length to 2048. These configurations are selected to follow the setup of Megatron-NLG. [8]

¹<https://github.com/hwnam831/meshslice>

We compare the performance of the algorithms in the Fully-Connected (FC) layers, combining all three training computations: forward, backward data, and backward weight. We report the FLOP utilization of each algorithm, which is computed as achieved GeMM compute throughput divided by the maximum compute throughput of the cluster (which is 272 TFLOPS per TPUv4). Because all the distributed GeMMs perform the same amount of compute, the FLOP utilization of an algorithm is proportional to its performance.

Figure 2.11 shows the FLOP utilization of the different algorithms for training GPT-3 (top) and Megatron-NLG (bottom). We see that MeshSlice is both the fastest method in all cases, and maintains good efficiency as the number of chips increases. For 256-way parallelism, MeshSlice is 13.8% and 26.0% faster than the state-of-the-art Wang algorithm in GPT-3 and Megatron, respectively. If we also include the performance of the non-FC layers of the LLM models, the *end-to-end speedups* of MeshSlice over Wang for 256-chips are 12.0% and 23.4% for GPT-3 and Megatron, respectively.

MeshSlice maintains high efficiency at 256-way parallelism. Going from 16-way to 256-way, MeshSlice only loses 16.8% and 5.8% of its efficiency for GPT-3 and Megatron-NLG, respectively.

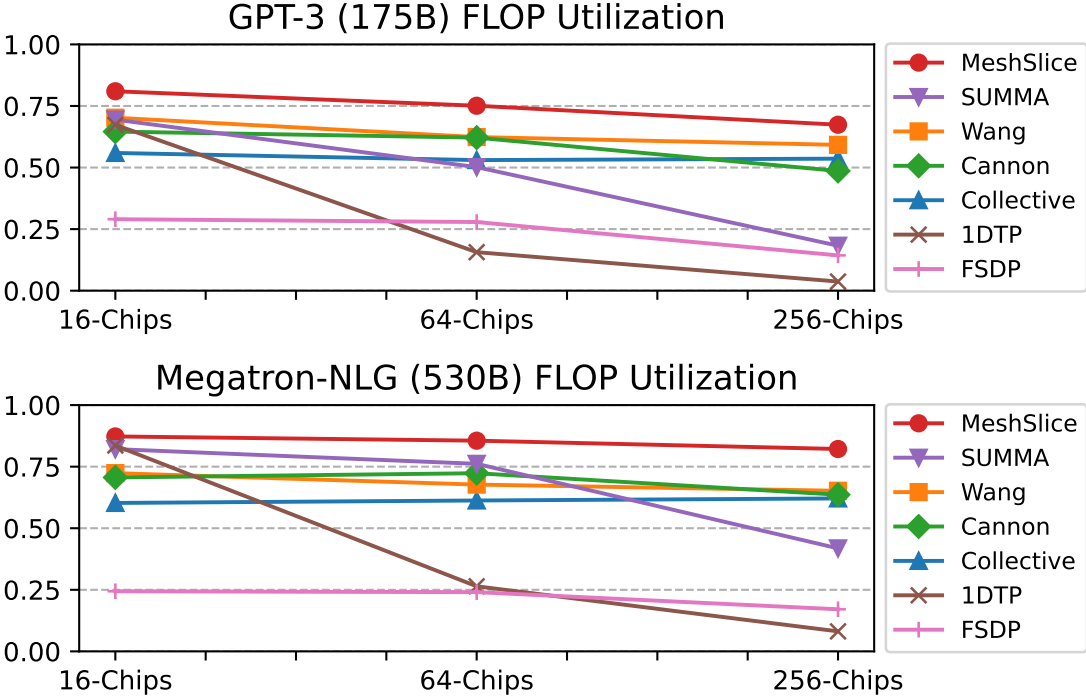


Figure 2.11: FLOP utilization of the FC layers with different distributed GeMM algorithms under weak scaling. The charts correspond to training GPT-3 (top) and Megatron (bottom).

Collective is always slower than MeshSlice because it can fall back to Collective by setting

$S = 1$ whenever it is more efficient to do so. Wang lies in between MeshSlice and Collective, as it partially overlaps one of the two collective communications in Collective. The other algorithms (i.e., SUMMA, Cannon, 1DTP, and FSDP) are inefficient in large clusters. They are even slower than Collective, which cannot overlap communications with computations.

Communication Cost Breakdown

To understand the inefficiencies of the algorithms, Figure 2.12 breaks down their total communication time (overlapped plus non-overlapped). For each algorithm, the figure shows the communication time relative to its GeMM computation time—which is almost the same in all the algorithms. The bars are broken down into three parts: time spent launching a communication operation (launch), transferring the shards (transfer), and synchronizing the chips (sync). In an algorithm, it is theoretically possible to hide all the communication time if the total relative time is less than 1.

Cannon has a relatively high communication time because it incurs a large traffic cost. Its extra traffic comes from two sources: the requirement for a square mesh shape and for skewing the matrix shards. The shortcoming of needing a square mesh shape becomes more pronounced at large cluster sizes. This is because, as we increase the cluster size, the weight matrix size remains constant but the batch size increases—which induces larger input and output matrices. As the matrix sizes become more imbalanced, the square mesh becomes more inefficient relative to the optimal mesh shape.

SUMMA has even higher communication time, due to its large synchronization overhead. As discussed in Section 2.1.3, SUMMA’s synchronization overhead grows quadratically with the number of rows or columns (whichever is larger). Its synchronization overhead becomes dominant in large meshes. Hence, SUMMA is efficient only in small clusters.

The high communication times in the 1D methods (1DTP and FSDP) show why we need 2D methods to scale DNNs efficiently. 1D methods are inefficient for two reasons. First, they can only utilize two ICI links to form a ring topology, rather than the four links of a 2D mesh. Second, 1D algorithms intrinsically have higher traffic than 2D algorithms, as discussed in Section 2.1.2. As a result, 1DTP and FSDP incur higher communication time than 2D GeMM algorithms, showing significantly lower performance in large accelerator clusters.

The figure shows that Collective has the least communication time, as it executes the smallest number of large collective communications. However, this communication time cannot be overlapped with computation. Wang and MeshSlice have slightly higher communication times than Collective. Wang has extra launch overhead to call multiple SendRecv opera-

tions, while MeshSlice adds extra synchronization overhead because each AG/RdS operation invokes more synchronizations than a SendRecv operation. Nonetheless, the communication time in MeshSlice can be mostly hidden by overlapping it with computation.

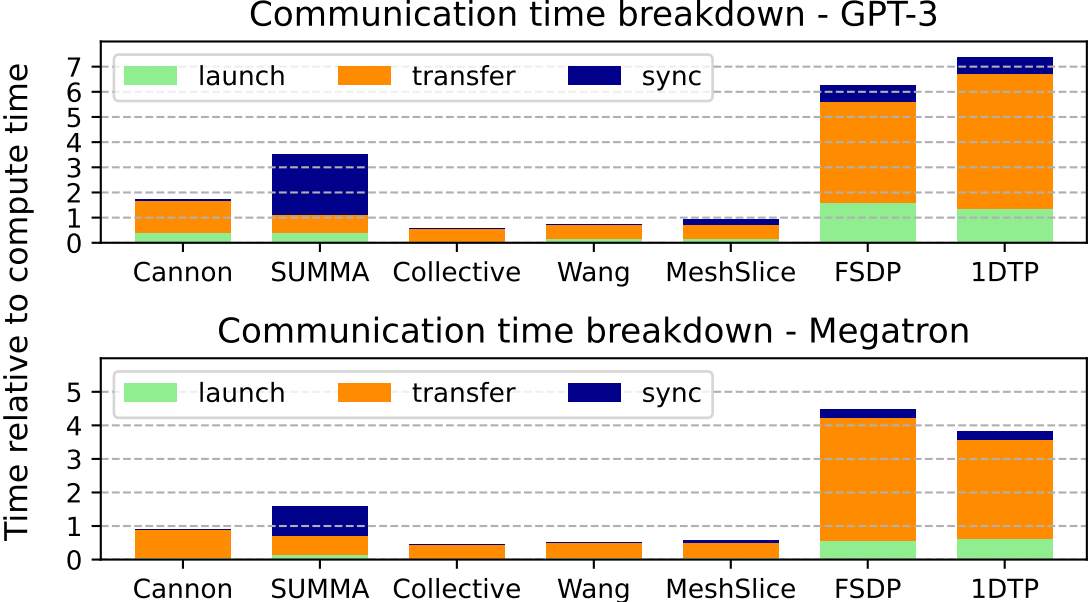


Figure 2.12: Breakdown of the communication time in the FC layers for the different algorithms relative to the algorithms’ own computation time. The charts correspond to training GPT-3 (top) and Megatron (bottom) in 256-chip clusters.

Strong Scaling Performance

Figure 2.13 compares the FLOP utilization of the different algorithms under strong scaling. In the experiments, the batch size is fixed to 32, which is the configuration for the 64-chip cluster in weak scaling. While strong scaling is not a realistic scenario in distributed DNN training, the results provide some insights. Note that FSDP cannot support strong scaling because DP assumes the batch size increases with the chip count.

The 16-chip results show a compute-bound scenario. Because there is not much communication cost to overlap, all the algorithms exhibit a relatively high efficiency. However, such a compute-bound scenario is becoming less common, as the compute power of ML accelerators is growing faster than the bandwidth of ICIs.

In the 256-chip results, the communication cost is now dominant. Hence, MeshSlice’s gain from communication overlapping diminishes, and MeshSlice shows a utilization similar to Collective and Wang. Still, MeshSlice has higher utilization than 1D TP and SUMMA,

which incur more traffic and synchronization overhead, respectively. Overall, MeshSlice is a safe choice regardless of whether it is compute-bound or communication-bound.

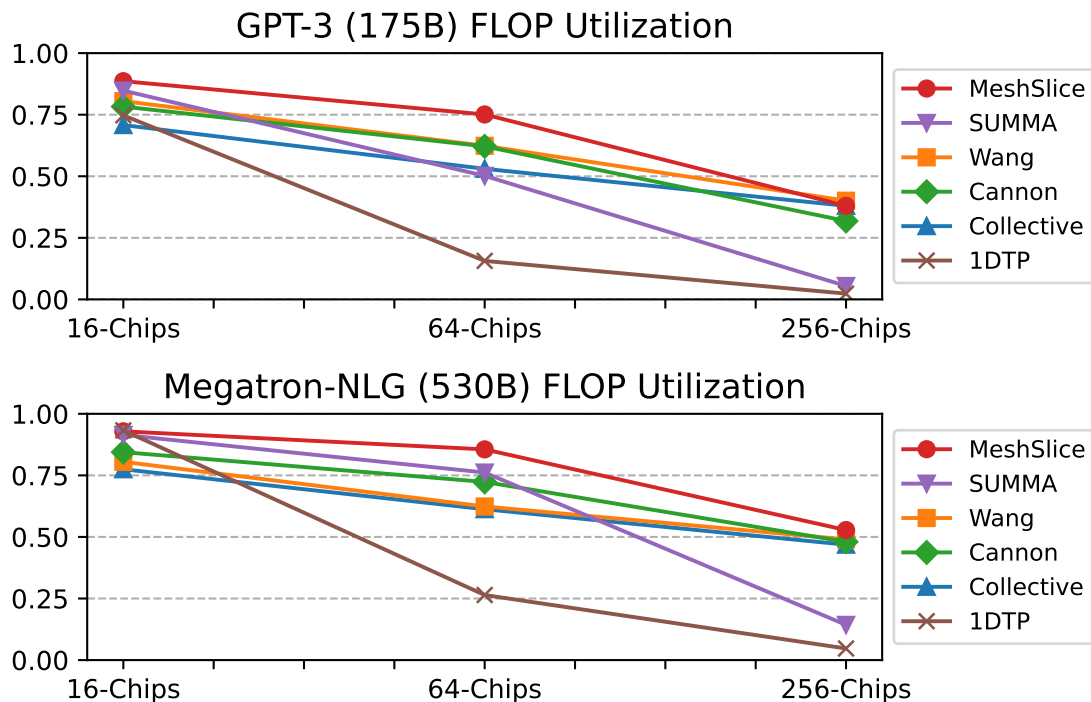


Figure 2.13: FLOP utilization of the FC layers with different distributed GeMM algorithms under strong scaling. The charts correspond to training GPT-3 (top) and Megatron (bottom).

Performance for Different Matrix Shapes.

During the forward and backward passes of the FC layers in LLM training, there are eight distinct GeMM operations with different M, N, K matrix shapes. Because we run GPT-3 and Megatron-NLG, this means we have a total of 16 GeMM variants. Figure 2.14 compares the FLOP utilization of these 16 GeMMs using the different 2D GeMM algorithms in a 256-chip cluster. We see that MeshSlice is consistently faster than the other algorithms in all 16 GeMMs. On average, MeshSlice is 27.8% and 19.1% faster than Collective and Wang, respectively. The speedups are higher in larger GeMMs, which take longer times to execute during LLM training.

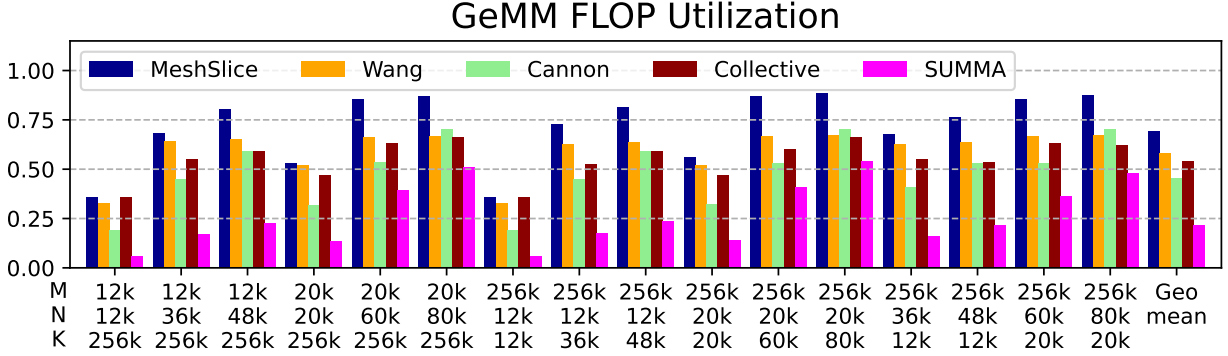


Figure 2.14: FLOP utilization of different 2D GeMM algorithms with different matrix shapes (M, N, K) in a 256-chip cluster.

2.6.2 LLM Autotuner and Cost Model

In this section, we validate the efficacy of the MeshSlice LLM autotuner. The autotuner optimizes three MeshSlice parameters: the dataflow, the mesh shape, and the slice count S . The shardings are automatically determined by the dataflows. We evaluate each parameter optimization in turn.

The first parameter configured by the autotuner is the dataflows of the FC layers. Without a dataflow optimization, the default approach is to use Y -stn dataflows from Table 2.1 for all FC layers. This is because the Y -stn dataflow does not transpose any of the matrices in the FC layers. Table 2.2 compares the FLOP utilizations in FC layer training without and with the MeshSlice dataflow optimization. The dataflow optimization brings 21.2% and 5.1% speedup in GPT-3 and Megatron, respectively. In Megatron, MeshSlice is already fast without the dataflow optimization because most of the communication cost can be hidden by overlapping it with the computation. However, GPT-3 performs a smaller amount of computation because the model is smaller. As a result, some of the extra communication cost caused by the unoptimized dataflow cannot be overlapped, bringing a significant slowdown over the optimized dataflow. Overall, the performance gains delivered by the dataflow optimization of the MeshSlice LLM autotuner are significant.

The MeshSlice LLM autotuner uses analytical cost models of communication and computation to find efficient configurations for the mesh shape and the slice count. The accuracy of the cost models is key to the autotuner effectiveness. Here, we compare the cost model estimations to the simulation results, to evaluate the accuracy of the analytical cost models. Note that what matters is that the models correctly estimate if one configuration delivers higher performance than another, not that they correctly estimate the actual performance of each configuration.

Table 2.2: FLOP utilizations in FC layer training without and with MeshSlice dataflow optimization in a 256-chip cluster.

LLM	Not optimized	Optimized	Speedup
GPT-3	55.6%	67.4%	21.2%
Megatron	78.2%	82.2%	5.1%

Figure 2.15 compares the FLOP utilization estimated by the autotuner’s communication and computation analytical cost models to the results obtained through simulations, for different mesh shapes of 256-chips. The autotuner’s cost models estimate the execution times of the FC layers. The simulated and estimated execution times are converted to FLOP utilizations for comparison. We see that the mesh shape has a large impact on performance: the optimal mesh shape can bring a 2.4x speedup over a non-optimal one in GPT-3. From the figure, we see that, for both LLM models, the cost models accurately identify the optimal shape.

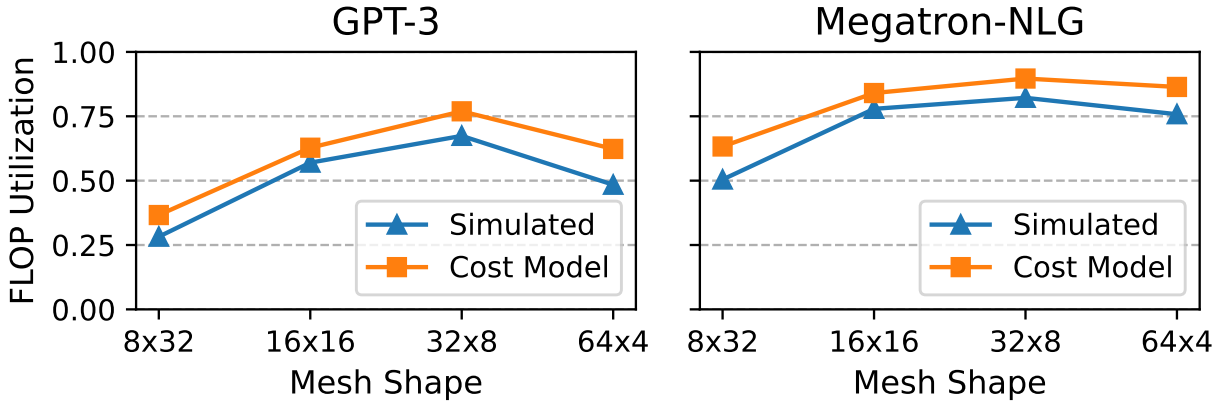


Figure 2.15: FLOP utilization estimated by the autotuner’s cost models and utilization obtained with simulations, for different mesh shapes of a 256-chip cluster.

Another hyperparameter that requires an accurate cost model is the Slice Count S . Figure 2.16 compares the FLOP utilization estimated by the autotuner’s communication and computation analytical cost models to the results obtained through simulations, in a 32×8 mesh with different S values. We see that the optimal slice counts found by the autotuner’s cost models are the same as the ones found by simulating the clusters. Overall, the MeshSlice LLM autotuner has simple but accurate analytical cost models that can find high-performance configurations of MeshSlice.

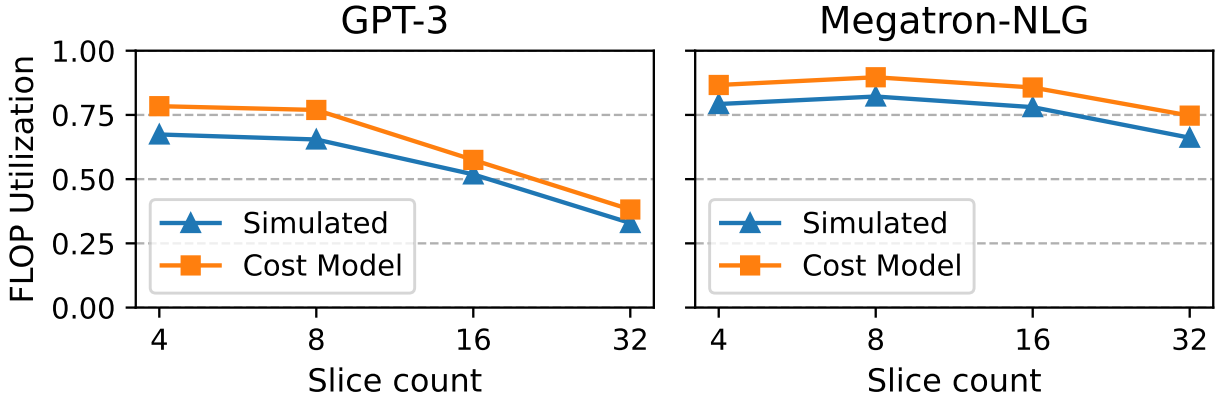


Figure 2.16: FLOP utilization estimated by the autotuner’s cost models and utilization obtained with simulations, for different Slice Counts S in a 32×8 mesh.

2.6.3 MeshSlice Performance on Real Hardware

In this section, we run MeshSlice on a real 4×4 TPUv4 cluster. We note that current TPUv4 clusters *do not allow* the overlap of AG/RdS operations with computations—although they support overlapping asynchronous SendRecv operations as used in Wang et al. [24]. As a result, MeshSlice is slower than Collective and about as fast as Wang. Nevertheless, we conduct these experiments to 1) measure MeshSlice’s intrinsic overheads and 2) validate the accuracy of our communication cost model.

Performance Comparison on a 4×4 TPU Mesh.

The first three columns of Table 2.3 show the FLOP utilization of the FC layers implemented with Collective, Wang, and MeshSlice on the 4×4 TPUv4 cluster. Note that all the FLOP utilizations are lower than those obtained with simulations in Figure 2.11. The reason is that Google Cloud’s 4×4 TPU clusters only utilize the *uni-directional bandwidth* of the bi-directional inter-node ICI links.

In this environment where MeshSlice cannot benefit from overlapping AG/RdS operations with computations, we see that MeshSlice adds $\approx 4.5\%$ execution time overhead over Collective. Out of this 4.5% extra overhead, it can be shown from the traces that only 1.3% comes from the MeshSlice slicing operations; the majority of the remaining overhead comes from the less efficient fine-grain partial GeMMs and fine-grain partial AG/RdS operations. Therefore, MeshSlice’s slicing mechanism is efficient and only adds very small performance overheads.

Wang is only marginally faster than Collective in GPT-3 and is slower than Collective in

Megatron. Wang’s speedup is lower than expected. This is because Google’s current Jax compiler optimizations create dependencies that prevent most of Wang’s communication operations from being overlapped with the computations.

Table 2.3: FC layer FLOP utilization of 2D GeMM algorithms in a real 4x4 TPUv4 cluster. MeshSlice-Overlap shows the *estimations* if AG/RdS were overlapped with computation.

LLM	Collective	Wang	Mesh-Slice	MeshSlice Overlap (Estim.)
GPT-3	47.4%	47.7%	45.5%	65.7%
Megatron	49.4%	46.4%	47.1%	65.6%

The last column of Table 2.3 (MeshSlice Overlap) shows the estimated FLOP utilization in MeshSlice if the AG/RdS operations were overlapped with GeMM computations. We see that, if AG/RdS operations were overlapped, we can expect 38.6% and 32.8% speedups of MeshSlice over Collective for GPT-3 and Megatron, respectively.

Validating the Accuracy of the Communication Cost Model.

Figure 2.17 validates the accuracy of our communication cost model using hardware measurements. The figure compares the estimated and measured communication times (overlapped plus non-overlapped) for different FC layers. We report the total communication time of one forward and backward pass per FC layer. We see that our communication cost model accurately estimates the communication times of the FC layers, with only 5.1% average error. The high accuracy of the communication cost model is expected because AG/RdS operations in a row or column suffer no network contention.

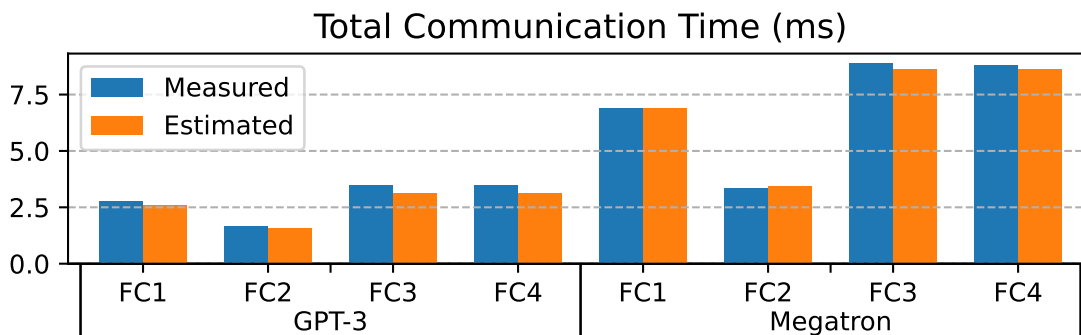


Figure 2.17: Estimated and measured total communication times of 8 different FC layers in MeshSlice.

2.7 DISCUSSION AND RELATED WORK

There are previous works that try to hide the communication cost of TP, but most of them target 1D TP. For example, Pati et al. [17] decompose a 1D TP GeMM into fine-grain computations and communications to overlap them. Centauri [18] applies communication partitioning and operation scheduling to a 3D training cluster that uses FSDP+PP+1DTP to efficiently overlap communications.

PrimePar [64] is the prior work most relevant to MeshSlice. It uses Cannon’s algorithm for 2D TP and adds an optimization algorithm that finds optimal sharding strategies on a given GPU topology. MeshSlice is a 2D GeMM algorithm that is more efficient and scalable than Cannon’s algorithm. Moreover, MeshSlice’s LLM autotuner differs from PrimePar’s optimization algorithm in four ways. First, MeshSlice focuses only on optimizing 2D TP, while PrimePar’s optimizer co-optimizes the partitioning using DP, PP, and TP together. Next, the MeshSlice LLM autotuner optimizes the cluster topology, while PrimePar works for a fixed cluster topology. Further, MeshSlice can choose from different 2D dataflows, while PrimePar only uses Cannon’s OS algorithm. Finally, MeshSlice allows configuring the communication granularity.

To further reduce the communication cost of 2D GeMM, algorithms have been proposed to compute a GeMM in a 3D cluster. The popular methods are 3D GeMM [56] and 2.5D GeMM [57]. 2.5D GeMM is the most popular method because 3D GeMM only works for a cubic 3D torus (i.e., a $P \times P \times P$ shape), while the 2.5D GeMM algorithm can work for any 3D torus with a $P \times P \times c$ shape. 2.5D GeMM makes c copies of the input matrices along the last dimension to reduce the communication cost. As an alternative, we can compute a GeMM in a 3D cluster by combining MeshSlice with DP, where DP also copies the weight matrices along the third dimension.

Because the 2.5D GeMM algorithm is based on Cannon’s algorithm, it suffers from the same limitations: it incurs high traffic due to skewing and the fact that it can only support a square shape for the base mesh (i.e., $P \times P$). As an example, assume that we build a 3D cluster of 1024 accelerators and use either 2.5D GeMM or MeshSlice plus DP to compute an FC layer of GPT-3 whose (M, N, K) is $1(1024K, 12K, 48K)$. In 2.5D GeMM, the only possible 3D torus shape is $16 \times 16 \times 4$, where the per-chip communication traffic becomes 1.6GB. On the other hand, with MeshSlice plus DP, we can choose the better shape of $32 \times 8 \times 4$, and only incur 336MB of per-chip communication traffic. Therefore, MeshSlice+DP has much less communication traffic than the 2.5D GeMM algorithm.

CHAPTER 3: POWERGRAD: GRADIENT-BASED HIERARCHICAL POWER MANAGEMENT OF POWER-LIMITED ML INFERENCE SYSTEMS

3.1 MOTIVATION

In this chapter, we design a power management framework for clusters running a variety of machine learning (ML) inference workloads. To gain insights, we start by characterizing the power and performance of four representative ML applications: language model (llama), image generation (stable-diffusion), image classification (resnet-50), and text-to-speech (VITS). The detailed setup is described in Section 3.3.2.

ML workload behavior varies substantially across models and across time. Figure 3.1 shows the power consumption and the instruction throughput (BIPS) of four ML applications across time. The figure shows that power and performance vary widely across models and time phases. For instance, *llama* shows two long and steady phases: the compute-bound encoding phase and the memory-bound decoding phase. On the other hand, *stable-diffusion* has short repeated iterations to refine the image multiple times. Each iteration has a high compute phase for DNN processing and a low compute phase for data pre/post-processing. In commercial AI platforms, there are even more model variations because the platforms serve the same models with different parameter sizes for different price models.

ML workload behavior depends on the user inputs. The computational and power demands of ML workloads demonstrate significant variability based on input and output characteristics [65]. For instance, a longer input prompt for a language model and a higher output resolution for an image generator increase the instruction throughput of the model computations. In addition, an AI model usually computes a *batch* of requests in parallel to enhance the computational efficiency. Figure 3.2 characterizes the power sensitivity of ML performance in two types of environments: *high* has inputs with larger batch sizes, longer prompt, and higher output resolutions; *low* has inputs with smaller batch sizes, shorter prompts and lower resolutions. As expected, the *high* performance is far more sensitive to power limits than the *low* performance, because when the inputs are smaller, the ML computation becomes more memory-bound due to the fixed cost of reading the model parameters. Therefore, it can be inferred that shifting the available power from *low* to *high* can improve the overall performance of the system. Moreover, from the user perspective, the latency of large-batch inputs is more important, since its processing time affects the response time of many requests.

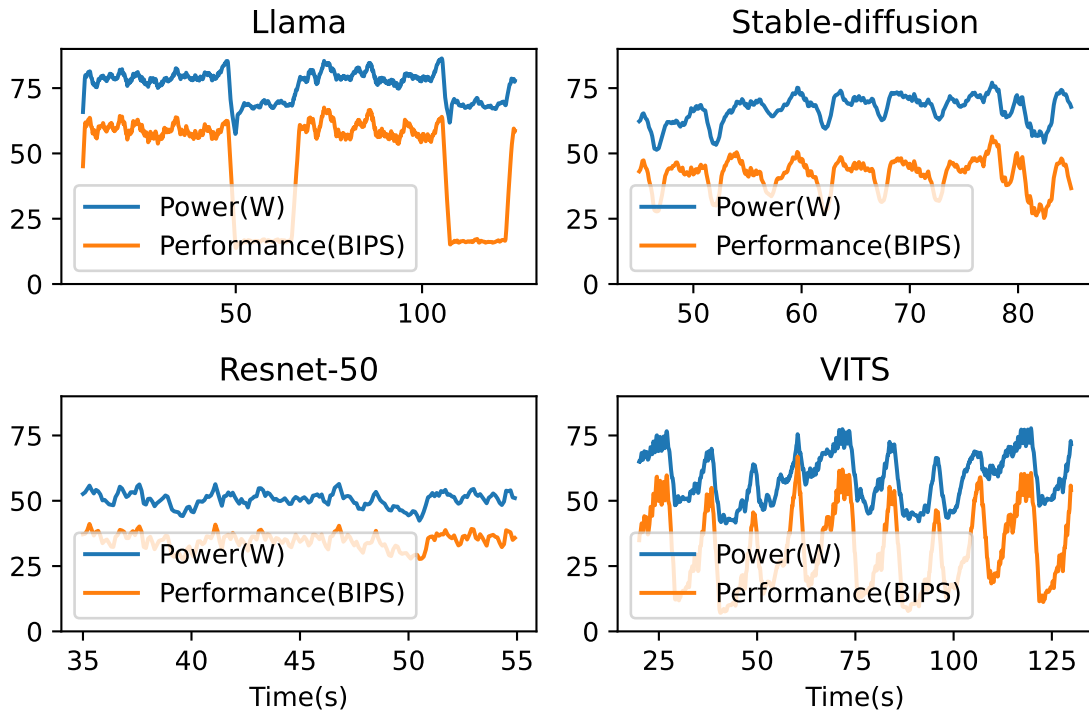


Figure 3.1: Power and performance patterns of four ML applications.

Power measurements are not sufficient to identify the performance behavior.

Consider *llama* in Figure 3.1. Even in the low-BIPS phase which is memory-bound, the power consumption is still relatively high (60W+). That is, in power-limited environments, memory-bound workloads are still likely to consume high power, making them indistinguishable from compute-bound workloads from the power consumption perspective. Therefore, we need a more intelligent method to characterize the workload behavior than measuring the power consumption.

3.2 THE POWERGRAD FRAMEWORK

3.2.1 Overview

To effectively manage power distribution of clusters running diverse ML workloads, we present a novel hierarchical power management system, namely *PowerGrad*. To understand what it does, consider Figure 3.3, which shows an example of a computer cluster. Nodes 1 and 2 have two processors each. The cluster is given a power budget ($PL_{cluster}$), which a cluster-level *Hierarchical Controller* in Node 3 divides into a per-node power limit (PL_i). Inside Nodes 1 and 2, a node-level *Local Controller* divides the power limit into a per-processor

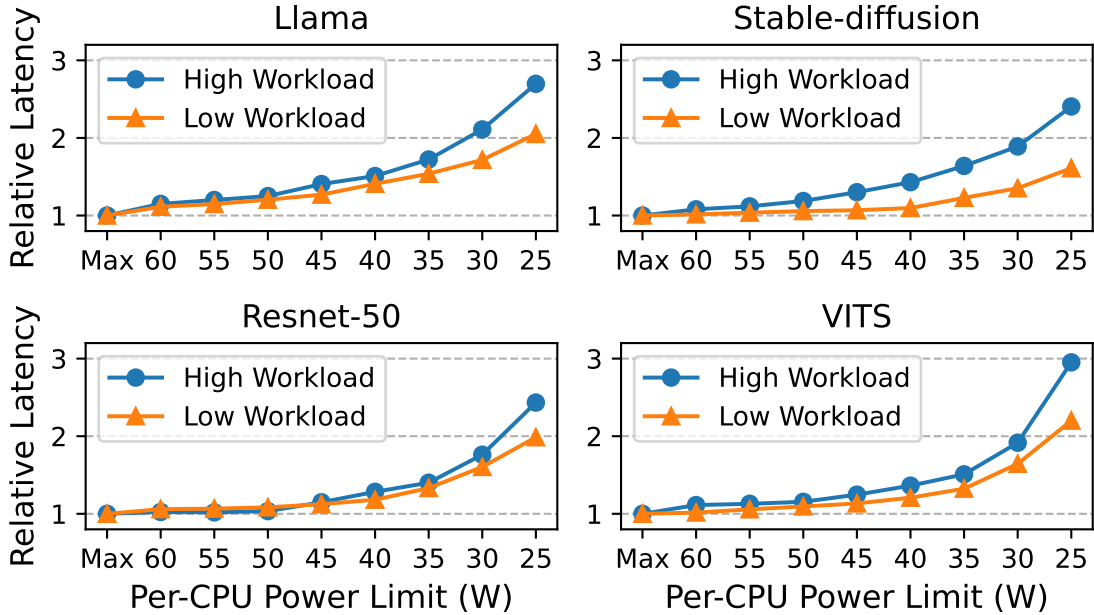


Figure 3.2: Latency changes of four ML models over different CPU power limits and input workload levels.

power limit ($PL_{i,j}$). The cluster power limit may change dynamically, as well as the power demands of the workloads running on the processors. Consequently, the hierarchical and local controllers need to adjust the power budgets dynamically.

Given an environment like this, the goal of PowerGrad is to find a power allocation that maximizes the performance of the system. Typically, an equal distribution of power is not optimal when the cluster is running heterogeneous workloads. In this case, to maximize the overall performance, the controller has to shift the power budget from applications that lose little performance when their power allocation is deducted to those that gain a lot of performance when their power allocation is increased.

PowerGrad attains this re-assignment by using a gradient-based optimization. The key idea of PowerGrad is dynamically estimating the *performance gradient* over power ($\frac{\partial perf}{\partial power}$). The performance gradient measures the performance sensitivity of each node to its power consumption. That is, a node running a compute-bound workload is likely to gain more performance per unit power increase, resulting in a higher gradient value. In contrast, a memory-bound node is likely to show a lower gradient value, as its performance is less sensitive to the CPU frequency. Therefore, shifting the power from a lower-gradient node to a higher-gradient one results in a net performance gain, since the latter gains more performance than the performance lost from the former.

Note that the performance gradient is not simply determined by whether node is compute-

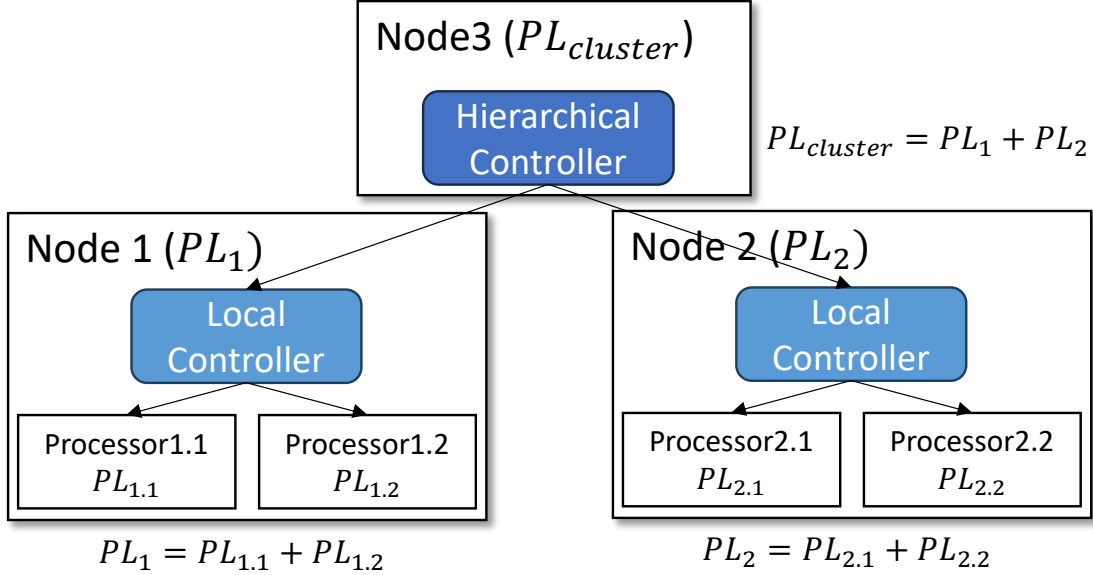


Figure 3.3: Hierarchical structure of PowerGrad.

bound or memory-bound. If the CPU frequency is high, even the compute-bound node can have a low performance gradient because it requires a lot of extra power to increase its performance.

PowerGrad requires *differentiable* models for performance and power to compute the gradients. To dynamically construct the power and performance models, PowerGrad uses runtime hardware performance counters, frequency, and voltage measurements. This data-driven approach does not require domain-specific knowledge of the running workload, and can be extended to any CPU architecture. Lack of knowledge about running workload is important in the ML inference workloads, as discussed in Section 1.4.

3.2.2 Components of the PowerGrad Framework

PowerGrad is a software framework that consists of three main components: Gradient Estimator, Local Controller, and Hierarchical Controllers. Its software architecture in a node is shown in Figure 3.4. In this section, we describe the flow of PowerGrad control in turn.

Gradient Estimator. At each timestep, the Gradient Estimator collects the runtime hardware counters (①) to build polynomial power and performance models. The coefficients of the polynomials depend on dynamic performance counters (e.g. cache-misses, instruction-count), frequency, and voltage. Then, the Gradient Estimator differentiates the per-core power and performance models to estimate the performance gradient, $\frac{\partial BIPS}{\partial P}$ (②). The

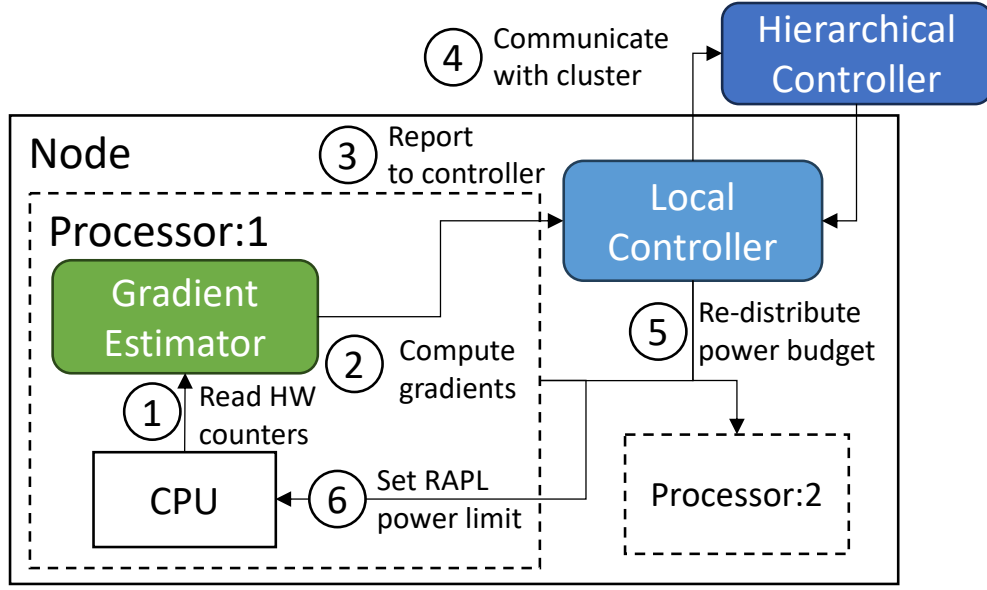


Figure 3.4: PowerGrad software architecture in a node.

power/performance measurements and the estimated gradient is reported to the node-level Local Controller (③).

Local Controller. The Local Controller is responsible for interacting with the upper-level Hierarchical Controller (④) and re-distributing the given node-level power budget to the processors (⑤). The local controller uses the gradients to actively re-distribute the node-level power budget to maximize the overall performance. The reallocated power budget is then enforced to the CPU processors via the Intel RAPL interface [66] (⑥).

Hierarchical Controller. The Hierarchical Controller at each control level leverages the power, performance, and gradient values reported from its child controllers to re-distribute the power budgets using the gradient-based optimization algorithm. This power re-distribution is done asynchronously with the child controllers, enabling fast-paced controls in the lower levels of the hierarchy. The hierarchical structure can scale recursively — the controller can report its aggregated power, performance, and gradient values to its parent Hierarchical Controller, which also asynchronously re-distributes the power budgets of its child controllers.

All of the PowerGrad components are implemented as user-level software. Inside a node, the modules are Java threads communicating and synchronizing through shared memory. We wake up the Gradient Estimator and the Local Controller in a short control period (=100ms), which perform steps ① to ⑥, and then go to sleep. The Hierarchical Controller is a Python process that communicates with the other controllers through network sock-

ets. Collecting information from all the node-level controllers and then providing responses takes time due to communication overheads. Hence, in our design, we run the cluster-level controller asynchronously in a longer control period (4 to 120 seconds).

3.2.3 Building Power and Performance Models from Runtime Hardware Measurements

PowerGrad uses the performance gradients over power P to make power allocation decisions. We measure the performance of a CPU as billion instructions per second (BIPS), which is a software-independent metric that can be measured by hardware performance counters. Therefore, we need to build differentiable power and performance models to compute $\frac{\partial BIPS}{\partial P}$ values.

Inspired by PPEP [67], the Gradient Estimator dynamically builds system power and performance models. It collects CPU runtime measurements at every timestep t , and builds the power model $P^{(t)}(V)$ as a polynomial function of voltage V , and the online CPI model $CPI^{(t)}(f)$ as a linear function of CPU frequency f . The coefficients of the power and CPI models are dynamically determined by the current frequency $f^{(t)}$ and runtime performance counters $\mathbf{E}^{(t)}$ measured from the CPU at every timestep. Finally, the models can estimate the power and performance behavior of the CPU when we apply a new voltage V and frequency f to the CPU.

$$P^{(t)}(V) = P_{idle}^{(t)}(V) + P_{active}^{(t)}(V, \mathbf{E}^{(t)}) \quad (3.1)$$

$$CPI^{(t)}(f) = CCPI^{(t)} + MCPI^{(t)} * f/f^{(t)} \quad (3.2)$$

In the power model, $P_{idle}^{(t)}$ is the idle power model which does not depend on the core activity and $P_{active}^{(t)}$ is the active power model that depends on the core activity (characterized by performance counters $\mathbf{E}^{(t)}$). The CPI is broken down to two pieces: memory CPI (MCPI) for memory operations whose performance does not scale with the core frequency and core CPI (CCPI) for instructions whose performance linearly scales with the frequency.

The idle power model $P_{idle}^{(t)}$ is a third-order polynomial of V , where the regression coefficients a_i are fitted off-line using idle power traces.

$$P_{idle}^{(t)}(V) = a_3V^3 + a_2V^2 + a_1V + a_0 \quad (3.3)$$

The active power model P_{active} is a function of voltage and performance counters $\mathbf{E}^{(t)}$ as

follows.

$$P_{active}^{(t)}(V, \mathbf{E}^{(t)}) = \sum_i w_i E_i^{(t)} * (V^\alpha + V) \quad (3.4)$$

Hereby w_i are regression coefficients that are also fitted *off-line* from active power data traces collected at a fixed V . α is the scaling factor characterized from the CPU behavior in different voltages. The performance event counts are divided by time to calculate $E_i^{(t)}$ whose unit is billion events per second, such as billion instructions per second (BIPS) and billion active cycles per second (BCPS) for instruction count and active cycle count, respectively.

For the performance model, the CPI is calculated by dividing active cycles to the instruction count, and the MCPI is calculated by dividing the memory stalls (ldm-stalls) with the instruction count. CCPI is CPI subtracted by MCPI.

$$BCPS = f * util \quad CPI = \frac{BCPS}{BIPS} \quad (3.5)$$

$$MCPI = \frac{ldm_stalls}{BIPS} \quad CCPI = CPI - MCPI \quad (3.6)$$

Hereby *util* is the core CPU utilization. The unit of f is GHz and *ldm-stalls* is billion memory stalls per second.

3.2.4 Estimating Performance Gradients

Figure 3.5 illustrates how the Gradient Estimator uses the hardware measurements to compute $\frac{\partial BIPS}{\partial P}$. At every timestep t , the measured performance counters $E^{(t)}$ with the regression coefficients (a_i in Eqn (3.3) and w_i in Eqn (3.4)) are combined to generate the power and performance models, $P^{(t)}(V)$ and $CPI^{(t)}(f)$ that take voltage V and frequency f as independent variables, respectively. Note that the regression coefficients are pre-determined off-line so that the coefficients of $P^{(t)}$ and $CPI^{(t)}$ only change with the runtime measurements of $\mathbf{E}^{(t)}$.

One of the major technical challenges of PowerGrad is differentiating the power and performance models to compute the performance gradients. This is because the hardware measurements have intertwined dependencies — voltage, frequency, and performance counters are correlated with each other. Therefore, we need good approximations of the relationships between the hardware metrics to accurately estimate the gradients.

To this end, we make three mathematical assumptions based on computer architecture knowledge. First, we assume that the core voltage is a quadratic function of frequency, which

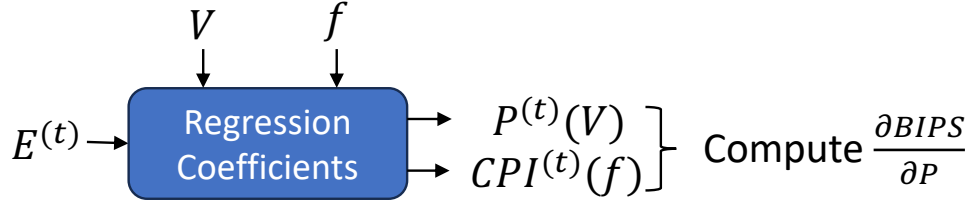


Figure 3.5: Workflow of how the Gradient Estimator computes the performance gradients from the hardware measurements.

is a common approximation of the V-f curve of a CPU. Second, in different frequencies, we assume that the performance counters $E_i^{(t)}$ are linearly proportional to $BIPS$. In other words, we assume that $E_i^{(t)} = e_i^{(t)} * BIPS$, where $e_i^{(t)}$ remain constant among different frequencies. This assumption is straightforward since performance events like branch misses and executed microops are mostly proportional to the total executed instructions. Finally, we assume that the CPU core utilization ($util$) is a function of frequency f as follows.

$$util(f) = \frac{util^{(t)}}{util^{(t)} + (1 - util^{(t)}) * f/f^{(t)}} \quad (3.7)$$

Hereby $util(f)$ is the new utilization as a function of changing frequency f , and $util^{(t)}$ and $f^{(t)}$ are the current utilization and frequency. This model is based on the assumption that non-idle times are inversely proportional to the frequency while the idle times remain constant. Differentiating $util(f)$ at $f = f^{(t)}$ gives the following.

$$\frac{\partial util}{\partial f}(f^{(t)}) = \frac{util^{(t)}(1 - util^{(t)})}{f^{(t)}} \quad (3.8)$$

Given the three assumptions, we begin differentiating the power model by breaking it into two parts: idle power and active power.

$$\frac{\partial BIPS}{\partial P} = \left(\frac{\partial P}{\partial BIPS}\right)^{-1} = \left(\frac{\partial P_{active}}{\partial BIPS} + \frac{\partial P_{idle}}{\partial BIPS}\right)^{-1} \quad (3.9)$$

Based on the second assumption, $P_{active} = \sum_i w_i E_i (V^\alpha + V)$ can be expressed as $\sum_i w_i e_i * BIPS * (V^\alpha + V)$, whose gradient can be computed as follows.

$$\frac{\partial P_{active}}{\partial BIPS} = \sum_i w_i e_i (V^\alpha + V) + \sum_i w_i E_i (\alpha V^{\alpha-1} + 1) \frac{\partial V}{\partial BIPS} \quad (3.10)$$

Since both V and CPI are function of f , we can compute $\frac{\partial V}{\partial BIPS}$ using the chain rule regarding f .

$$\frac{\partial BIPS}{\partial f}(f^{(t)}) = \frac{util * CCPI}{CPI^2} + \frac{util(1 - util)}{CPI} \quad (3.11)$$

$$\frac{\partial V}{\partial BIPS} = \frac{\partial V}{\partial f} \left(\frac{\partial BIPS}{\partial f} \right)^{-1} \quad (3.12)$$

Since we assume a quadratic V-F curve, $\frac{\partial V}{\partial f}$ can be obtained by differentiating the fitted quadratic function.

By re-using the computed $\frac{\partial V}{\partial BIPS}$, we can also compute the idle power gradient by differentiating the idle power polynomial.

$$\frac{\partial P_{idle}}{\partial BIPS} = \frac{\partial P_{idle}}{\partial V} \frac{\partial V}{\partial BIPS} = (3a_3V^2 + 2a_2V + a_1) \frac{\partial V}{\partial BIPS} \quad (3.13)$$

Hereby a_i are the regression coefficients in Eqn (3.3).

Since we collect hardware counters per every core, the Gradient Estimator builds *per-core* performance and power models. Meanwhile, we want to optimize the efficiency of an entire computer cluster which is hierarchically organized as multiple nodes, where each node has one or more multi-core CPU sockets. Therefore, we need to combine per-core perf/power measurements $BIPS_i, P_i$ and gradients $\frac{\partial BIPS_i}{\partial P_i}$ to maximize the system-wise $BIPS$ where $BIPS = \sum BIPS_i$ and $P = \sum P_i$. Because the power consumption of one core is independent of the performance of another core, the chain rule of differentiation allows the simple aggregation rule to compute the performance gradient $\mathcal{G} = \frac{\partial BIPS}{\partial P}$.

$$\mathcal{G} = \frac{\partial BIPS}{\partial P} = \sum_i \frac{\partial BIPS}{\partial P_i} \frac{\partial P_i}{\partial P} = \sum_i \frac{\partial BIPS_i}{\partial P_i} \frac{P_i}{P} \quad (3.14)$$

3.2.5 PowerGrad Local Controller

PowerGrad's Local Controller performs a gradient-based optimization to find a processor power distribution that maximizes the total instruction throughput (BIPS). Hence, the Local Controller makes the decision using the gradient \mathcal{G} from Eqn (3.14). Specifically, at every control period, the PowerGrad Local Controller re-distributes the node-level power budget to its CPU processors with Algorithm 3.1. For the Local Controller, its children are CPU sockets, and the parent is the Hierarchical Controller.

The Local Controller takes the current per-CPU power consumptions P , current frequencies f , and the gradient estimations \mathcal{G} as inputs given from the per-CPU Gradient Estimators.

Algorithm 3.1 PowerGrad power allocation algorithm

```
1: //  $\mathcal{G}$ : performance gradients,  $f$ : average CPU frequency
2: //  $P$ : power consumption,  $PL$ : power limit
3: //  $lr$ : learning rate,  $\alpha$ : decrement rate for unused budget
4: function ALLOCATE_POWER( $\mathcal{G}, f, P, PL, lr, \alpha$ )
5:   communicate(parent,  $\mathcal{G}, f, P$ ) // Report to the parent controller
6:   global  $PL_{node}$  // The limit is asynchronously set by the parent
7:    $PL_{total} \leftarrow 0$ ;  $PL' \leftarrow PL$ ;
8:   for  $c \in children$  do // Apply gradient ascent optimization
9:      $PL'[c] \leftarrow PL[c] + lr \times \mathcal{G}[c] - \alpha(PL[c] - P[c])$ 
10:     $PL_{total} \leftarrow PL_{total} + PL'[c]$ 
11:   for  $c \in children$  do // Adjust all power budgets equally
12:      $PL'[c] \leftarrow PL'[c] - (PL_{total} - PL_{node})/N_{children}$ 
13:   for  $c \in children$  do // Ensure the minimum frequency
14:     if  $PL'[c] < PL[c] + 1$  and  $f[c] < min\_freq$  then
15:        $PL'[c] \leftarrow PL[c] + 1$ 
16:       Re-adjust other CPU power budgets accordingly
17:     break;
return  $PL'$ 
```

To begin with, it reports its states to the parent controller (line 5). The parent controller also re-distributes the cluster power budget $PL_{cluster}$ to the nodes in a different timescale and updates the node-level power limit PL_{node} asynchronously.

Next, the first loop at line 8 applies the *gradient ascent* to determine the power limit of CPU processors. Each CPU socket's power limit is incremented proportionally to its performance gradient $\mathcal{G}[c]$. Hereby lr is the learning rate which determines the speed of iterative optimization. Higher lr enables a faster optimization, but increases the risk of unstable oscillations. Note that it is possible that the power demand of the processor is not high enough to consume the current power limit. Hence, we also adjust the power limit by $\alpha(PL[c] - P[c])$ to try to close the power gap between the limit and the actual consumption. Hereby $\alpha \in (0, 1)$ is the correction factor. Eventually, the power limit and the power demand will converge. A high α keeps the power limit closer to the actual power, but may result in slower response when the system increases its power demand. The hyperparameters lr and α are tuned by benchmarking the target system.

Then, the second loop at line 11 ensures that the sum of children power limit PL_{total} does not exceed PL_{node} . This is done by equally adjusting the difference $(PL_{total} - PL_{node})$ to all children. When the initial power limit sum is bigger than PL_{node} , $(PL_{total} - PL_{node})$ becomes positive and all power limits $PL'[c]$ are equally subtracted by $(PL_{total} - PL_{node})/N_{children}$. This equal adjustment preserves the relative difference between the new power limits com-

puted by gradient ascent at line 9. Hence, the equal adjustment results in moving the power budget from children that have a low \mathcal{G} to those that have a high \mathcal{G} . In other words, it shifts power budget from processors whose performance is less sensitive to power to processors whose performance is more power-sensitive.

Additionally, the last loop at line 13 prevents a child from starvation when its power limit is too low to maintain a set minimum frequency. If the controller finds a child running at a very low frequency, it force-increments the power limit of that child and re-adjusts the other power limits by an equal amount. This is a safeguard logic to ensure the stability of the gradient-based optimization when the learning rate lr is high.

Finally, the algorithm returns the new power limits PL' to be applied to the CPU sockets. The Local Controller enforces the chip power limit through Intel RAPL. RAPL controls the V - f state of the chip to ensure that the running average power stays below the set power limit. Because RAPL only supports per-chip power measurement and control, a CPU processor chip is a leaf in our hierarchy. The overall algorithm repeats at every control period with new runtime measurement inputs from the Gradient Estimator.

3.2.6 Adding More Levels to the Hierarchy

PowerGrad’s control algorithm can recursively scale to any levels of the hierarchy. For each new level that we add, the Hierarchical Controller for that level follows the same power allocation algorithm in Algorithm 3.1. In this case, the first step is for each child to *aggregate* its measurements and gradient estimations from its own sub-components. The power measurements P are summed, the frequency measurements f are averaged, and the gradient estimations \mathcal{G} are aggregated using Eqn (3.14). Then, the aggregated values are reported to the parent Hierarchical Controller as the inputs for the power allocation algorithm (Algorithm 3.1).

We can add any number of hierarchical levels using these principles as illustrated in Figure 3.6. The default hierarchical structure of PowerGrad is shown in Figure 3.6a. It represents a two-level hierarchy — a node-level controller determines the power of the CPUs in the node and the cluster-level Hierarchical Controller distributes the power budget to the nodes. In addition, we can consider centralized (PG-central) and multi-level (PG-multi) variants of PowerGrad. PG-central (Figure 3.6b) has a single cluster-level controller that directly allocates the power limits of all CPUs in the cluster. On the other hand, PG-multi (Figure 3.6c) divides the cluster into multiple sub-clusters each of which has its own sub-cluster controller.

Note that the speed of the controls in the lower levels of the hierarchy is unaffected by the extra levels of the hierarchy. This is because the operations at different levels are

asynchronous with each other. As we move up the hierarchy, however, communication is more expensive, as it involves using network sockets and suffering long network link latencies. As a result, higher-level controllers run less frequently. For example, in PG-multi, we run the node-level controllers every 100ms, sub-cluster controllers every second, and the cluster controller every 8 seconds.

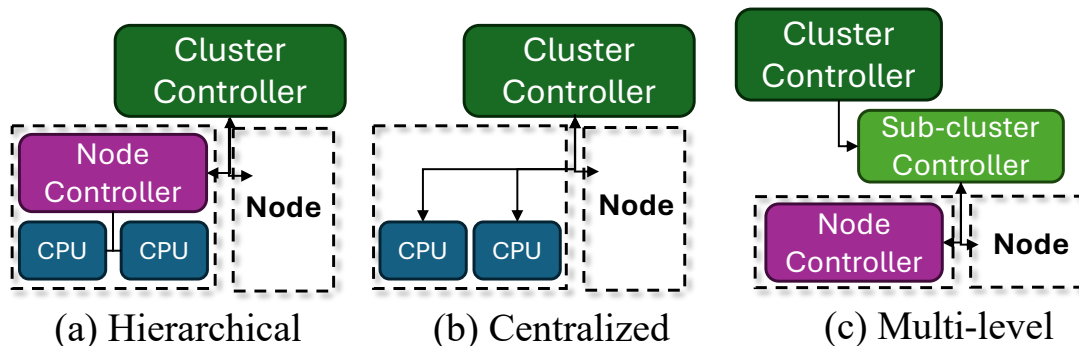


Figure 3.6: Different levels of PowerGrad control.

3.3 EXPERIMENTAL SETUP

3.3.1 System Setup

We conduct experiments on CPU cluster environments in Cloudlab [68]. Each node (c220g2 of Cloudlab) has two 10-core Intel Haswell processors (E5-2660 v3). Each processor power limit is enforced with Intel RAPL interface.

We use 17 nodes in total: one controller node and 16 inference nodes with 2 processors each. This forms a two-level hierarchy in Figure 3.6a, where the cluster controller is activated at every 8 seconds and the node controllers run at every 100ms. We also implement the central (PG-central) and multi-level (PG-multi) variants. In PG-central, the Local Controller is disabled and the Cluster Controller directly determines the power limits of $16 \times 2 = 32$ CPUs. In PG-multi, the cluster is broken down to four sub-clusters, where each sub-cluster runs only one type of ML model.

3.3.2 ML Inference Models

In this work, we benchmark the power management systems using four machine learning models listed in Table 3.1. The first model is LLama-3.1-8b [11] (Llama), one of the most

Table 3.1: ML inference applications used.

Service type	Model name	DNN structure
Language model	Llama-3.1-8b	Transformer
Image generation	Stable-diffusion	Transformer+CNN
Text-to-speech	VITS	Transformer+1D CNN
Image classification	Resnet-50	CNN

popular open-source large language model. Llama follows the autoregressive Transformer [12] architecture which has a highly parallel encoding phase which encodes the input prompts and a sequential decoding phase to generate the output tokens. Its performance behavior is determined by the input prompt length, batch size, and the number of output tokens. We choose the smallest 8-billion parameter model, which is capable of generating 2 to 3 tokens per second in our 10-core Intel Haswell CPU.

The second model is Stable Diffusion [69] (SD), which is the most popular model used for image generation. SD combines convolution neural network (CNN) [70] and Transformer architectures to encode both image and text data together and generate the output image in multiple iterations. Its performance behavior largely depends on the resolution of the output image. We choose input resolutions ranging from 128×128 to 512×512 .

Next is a popular text-to-speech model, namely VITS [71]. VITS combines Transformer and 1-dimensional CNN architectures to generate the voice output from the text input. Its performance mostly depends on the input length.

Finally, we run image classification using Resnet-50 [72] (Resnet), one of the most popular CNN image classifier models. Resnet is relatively lighter and faster than the other three models. Resnet’s performance behavior only depends on the batch size, as it only consumes fixed-resolution (224×224) images.

All models are run in a containerized environment where each container is assigned to a CPU processor. Llama, SD, and VITS are loaded from Huggingface’s model library [73], while Resnet is loaded from Mxnet’s model zoo [74].

To run the four ML models, we build a cluster where each node is assigned a single ML model. In a node, there are two containerized copies of the model that are assigned to the two CPU processors. The requests are categorized by the workload level (high, med, low) and partitioned into the CPU processors based on the category. The *high* inputs use large batch sizes, high output resolution, or long input prompts, while the *low* inputs use the opposite. This is a typical scheduling pattern in ML inference clusters, as requests with similar input patterns are often *batched* and processed within the same node.

For each processor, we generate 10-minute request arrival patterns using the Poisson

process. The generated arrival patterns are re-used for all evaluations for fair comparison of the request response times. In each experiment, we measure the per-model average response time and the tail (95th percentile) latency for the arrived requests. Because the ML models process batched requests, we treat a request of batch size N as N individual requests when we calculate the average and tail latencies.

3.3.3 Training Regression Coefficients

To train the regression coefficients (a_i in Eqn (3.3) and w_i in Eqn (3.4)), we collect hardware counter traces by running the Parsec 3.0 benchmarks [75] on our target CPU nodes. Note that we do not train the coefficients using the target ML workloads for better generalization. We mostly follow the original PPEP workflow [67] to train the regression coefficients. However, there are two major differences because of architectural differences between the AMD CPU used in the original PPEP work and the Intel CPU we use.

First, the number of maximum performance counters and the list of counters are different from those of AMD. We collect six performance counters: *instruction-count*, *cycle-count* (non-idle cycles), *uops.executed* (uops), *cycle_activitiy.ldm_stalls_pending* (memory stalls), *cache-misses*, and *branch-misses*. In addition, the type of executed instruction plays a big role in the CPU energy consumption, but the available counters do not distinguish the types of instructions being executed. Therefore, the power model coefficients are multiplied by the dynamic adjustment factor γ at runtime. γ is the running average of the ratio between the measured power consumption and the predicted power consumption. The dynamic adjustment factor also applies to the gradient estimation, improving the gradient estimation accuracy as well.

3.3.4 Baselines

We compare PowerGrad to two baselines of SLURM [33] and DPS [34]. These baselines are software transparent as they only rely on power measurements, thus applicable to dynamic ML workloads. However, since these baselines are *centralized*, their algorithms are implemented in the Python process in the controller node, along with the cluster-level Hierarchical Controller. Therefore, like PG-central, they skip the Local Controller and directly allocate the power budgets to the 32 CPUs. They are implemented alongside with the Hierarchical Controller, running at the same 8-second control period.

3.4 POWERGRAD EVALUATION

In this section, we evaluate each component of PowerGrad. First, we measure PowerGrad Gradient Estimator’s gradient values to verify that the estimated gradients effectively reflect the power-performance behavior of the ML workloads. Next, we evaluate the efficacy of the PowerGrad Local Controller by running a single-node evaluation only with the Local Controller activated. Finally, we compare the whole PowerGrad framework against the baseline power management systems in the cluster environment running the ML inference workloads.

3.4.1 Verifying the Gradient Estimations

The efficacy of PowerGrad depends on the accuracy of the estimated performance gradient ($\partial BIPS/\partial P$) values. However, a complete verification of the accuracy of the estimated gradients is impossible because the workload dynamics rapidly change for a variety of reasons. Instead, we can compare the power-performance relationship estimated by the gradient values against the real trace data to qualitatively evaluate the Gradient Estimator.

Figure 3.7 shows the comparisons between benchmarked power-performance values and estimated power-performance curves from the gradients. Hereby we run four different ML inference applications (Llama, SD, VITS, Resnet) in two different input workloads (high and low). During the executions, we constantly change the CPU frequency via a random walk of 100Mhz steps so that the power-performance behaviors of the applications change dynamically. Then, we collect the data points (blue dots) from the steady-state portion of each execution. For each data point, the Gradient Estimator estimates the performance gradient value $\partial BIPS/\partial P$. Using the gradient values, we first apply linear regression to the linear $\partial BIPS/\partial P$ -power function. Next, we integrate the linear function to draw the estimated quadratic *BIPS*-to-power trend curve (red curves in Figure 3.7).

The results show that the estimated trend curves match reasonably well with the performance-power behavior of the collected data points. The curves effectively reflect whether the workload is compute-bound or memory-bound. For instance, the trend curve has a steep slope for the compute bound *high* workload of Llama, while it shows a gentle slope for the memory bound *low* workload of Llama. Note that the Gradient Estimator is not specifically trained for ML workloads — the underlying regression coefficients are trained only using the Parsec benchmark traces. That is, the Gradient Estimator can effectively characterize the power-performance behavior of various unseen workloads, which is a desirable attribute for controlling widely dynamic ML inference workloads.

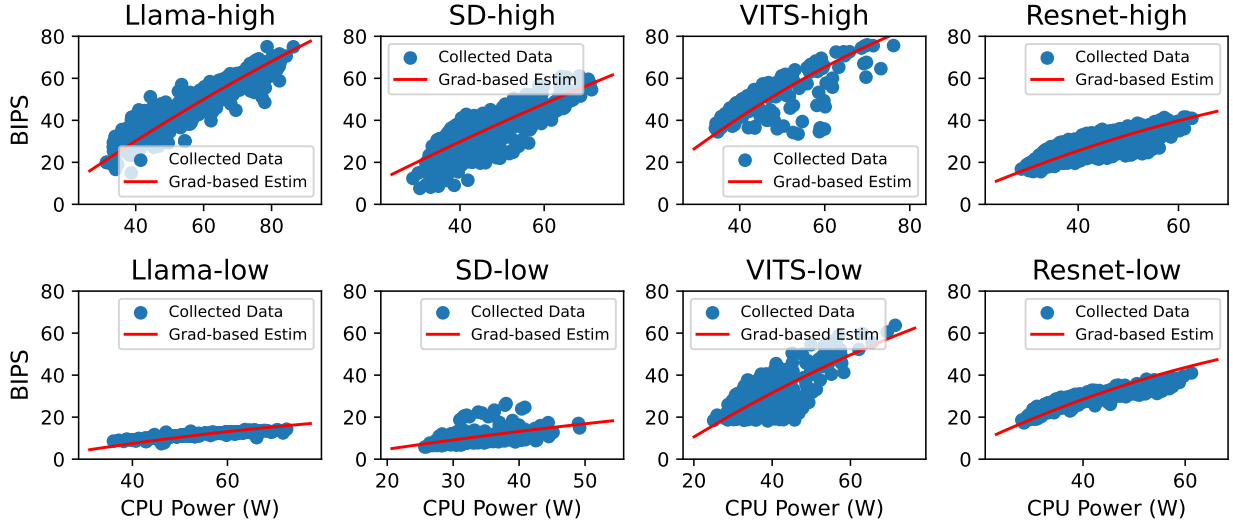


Figure 3.7: Comparison between collected power-performance traces (blue dots) and power-performance curves estimated from the gradients (red line) in the steady state of different ML applications.

3.4.2 Local Controller Evaluation

In the second experiment, we evaluate the efficacy of PowerGrad’s Local Controller in a single node running ML inference workloads. As described in Section 3.3.2, each model is copied to two CPUs in a node and each CPU runs different inputs (‘high’ or ‘low’) arriving in different traces. Hereby only the Local Controller and the Gradient Estimator are enabled, re-distributing the CPU power limits at every 100ms control period.

Figure 3.8 compares the Local Controller to the equal power distribution (Fair). We applied different node-level power limits (75 to 55W), which are split into half per CPU (37.5 to 22.5W) in Fair power distribution. In terms of the average performance, using the PowerGrad Local Controller can achieve 16.1% lower geomean response time over Fair. The difference can be as wide as 37.5% in Llama. The tail (P95) latency speedup is even more significant. The Local Controller achieves 28.8% shorter tail latency over Fair. The difference can be as wide as 51.1% in Llama. The results show that active power re-distribution between the CPU sockets can bring meaningful speedup, even without considering the other nodes in the cluster. The difference is larger for tail latencies because resource allocation plays a critical role when requests are congested and starving for compute resources.

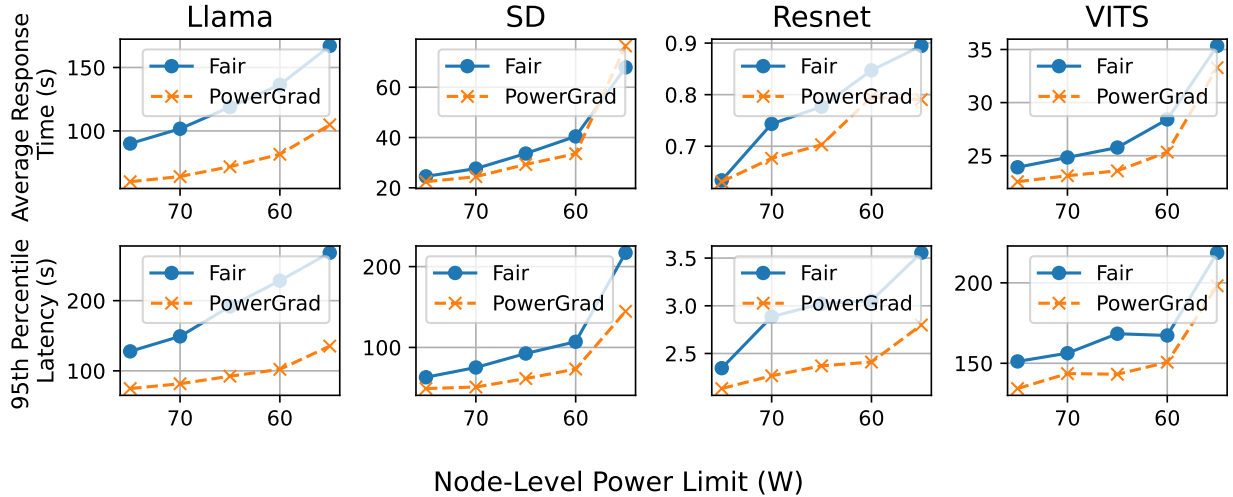


Figure 3.8: Comparing average and 95th percentile (tail latency) response times of four ML applications with equal power distribution (Fair) or with PowerGrad Local Controller, for different node power limits.

3.4.3 Cluster-level Evaluation

Finally, we evaluate PowerGrad as a power management framework compared to the baselines (DPS [34] and SLURM [33]). Here, we control a cluster of 16 ML inference nodes, each of which is running one of the four ML models in Table 3.1. We measure the per-application average and tail response times in the cluster, then take the geometric mean of them to compare the performance under the power management methods.

Figure 3.9 compares the average and tail response times of the four ML applications in the cluster. The response times are shown relative to the equal power distribution (Fair). For the average response time, PowerGrad is the fastest and the gap becomes wider when the power is more limited. On average, PowerGrad’s response time is 24.9% and 20.3% lower than those of Fair and DPS, respectively. When the power limit is as low as 55W per node, PowerGrad reduces the average latency by 25.7% over DPS. The trend is similar when we compare the tail (95th percentile) latency. PowerGrad’s tail latency is 27.3% shorter than DPS, 20.4% shorter than SLURM, and 33% shorter than Fair. When the power limit is 55W per node, PowerGrad reduces tail latency by 26.3% over DPS.

We also compare centralized PowerGrad (PG-central in Figure 3.9) to the others to evaluate the efficacy of PowerGrad without the hierarchical structure. PG-central is still better than the other baselines but not as fast as the hierarchical PowerGrad. The results indicate that both intelligent power distribution based on gradients and hierarchical structure are important to the success of PowerGrad.

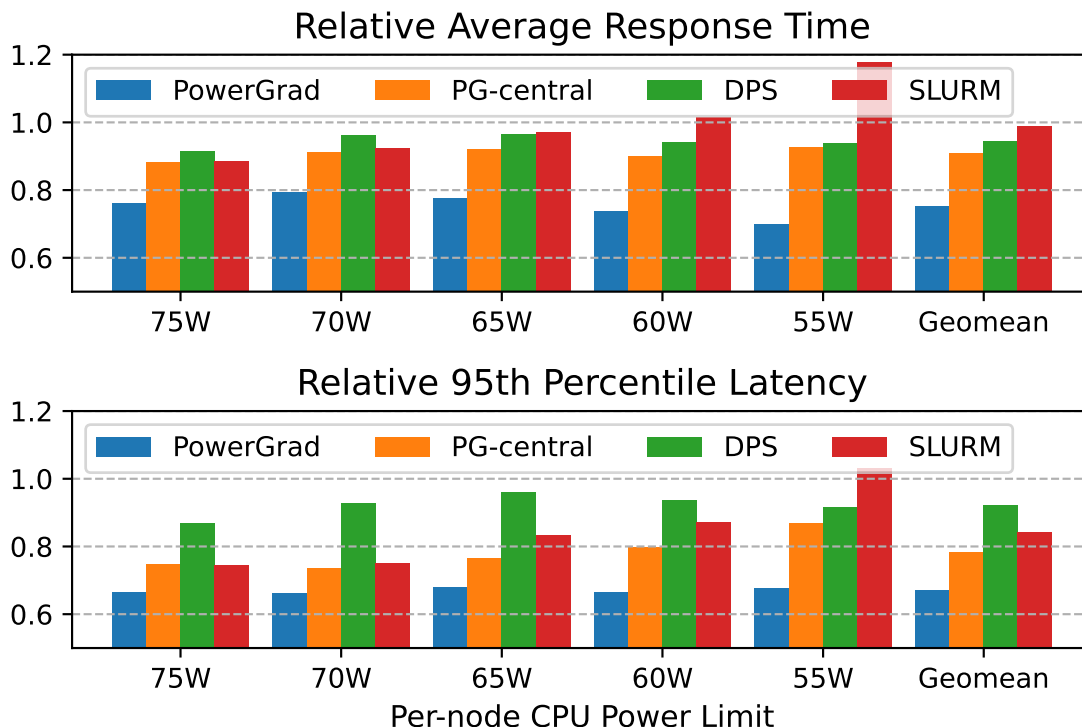


Figure 3.9: Average and tail latency (relative to Fair) of a 16-node cluster running four ML models controlled by PowerGrad or baselines for different node power limits.

Note that in larger datacenters with thousands of the nodes, the control period of the cluster management may become significantly longer than 8 seconds due to communication costs. For instance, the default control period of SLURM is 60 seconds, which is 7.5 times longer than 8 seconds. To estimate the behavior of the power management methods in such environments, we change the control period of the cluster-level controller from 4 seconds to 120 seconds to evaluate how scalable they are.

Figure 3.10 compares the average and tail response times of the power management methods with different control periods. We compare them in the fixed 60W per-node power limit because it can be observed from Figure 3.8 that 60W is usually the tipping point for rapid system behavior changes. We see that the centralized methods (SLURM and DPS) are more sensitive to the control speed. With longer control periods, their performance becomes closer to that of Fair, even showing worse performance with very long control periods. On the other hand, PowerGrad is fairly robust to the control frequency change, showing relatively steady efficacy up to a 32-second control period.

We also compare multi-level PowerGrad (PG-multi shown in Figure 3.6c) to the others, with four sub-cluster controllers running at fixed 1-second control period. We observe that

adding more levels to the hierarchy makes PowerGrad more resistant to the control frequency change. Hence, the hierarchical structure of PowerGrad helps it scale to very large datacenters where the top-level controller requires a very long time to communicate to the nodes.

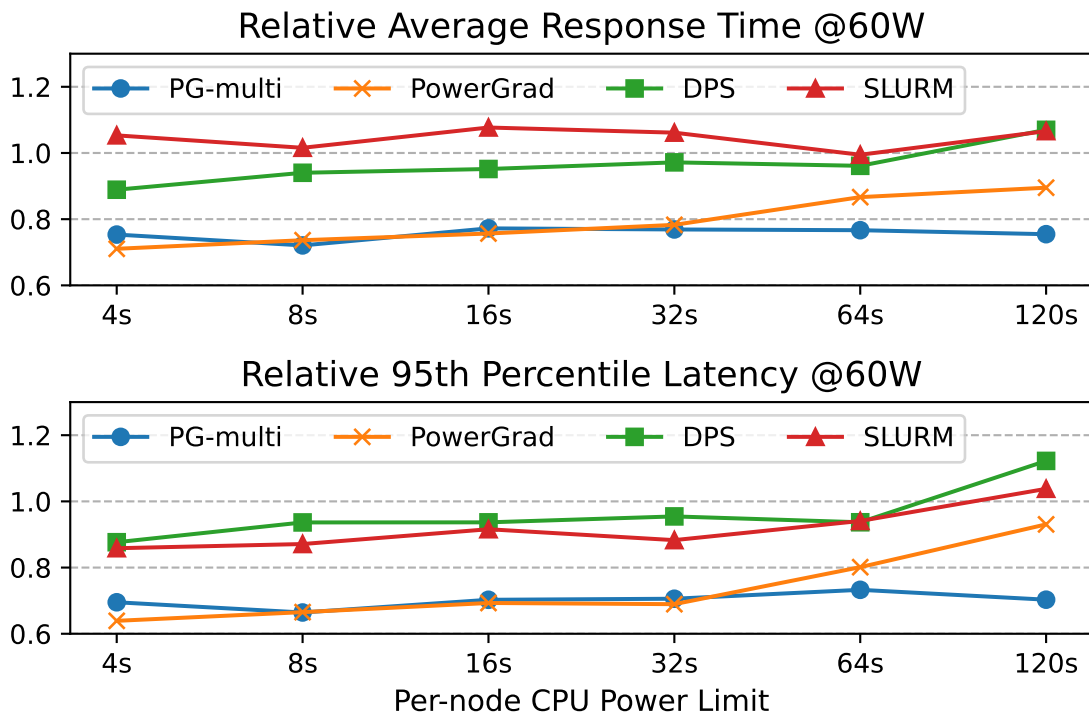


Figure 3.10: Average and tail performance (relative to Fair) of the 16-node ML cluster at fixed power limit (60W) controlled by PowerGrad and baselines for different control periods.

3.5 RELATED WORK

Distributed computer systems often adopt power *overprovisioning* — building clusters that can draw more power than the available power supply. Such overprovisioned clusters require *power management systems* that limit the system power consumption. Power management systems allocate power budgets to the child nodes and let hardware mechanisms (e.g. Intel RAPL [66]) to enforce the power limits through dynamic voltage-frequency scaling (DVFS).

Researchers have proposed a variety of power management systems for power overprovisioned clusters. However, most of the methods require prior information about the application workload — making them not suitable to manage ML inference workloads. For instance, Dynamo [30] and CapMaestro [29] are hierarchical power-capping solutions that rely on application profiling and priority information. They assume that only one type of application

runs per node, whose priority and quality-of-service is known ahead of time. Then, they shift power from low-priority applications to high-priority applications. Another example is PoDD [32], which is a design that finds the optimal power distribution between *coupled* (e.g., producer-consumer) applications using online profiling. PoDD is also not applicable to ML inference clusters because it is designed specifically for coupled applications, while ML inference clusters serve multiple independent ML applications.

To manage the dynamic behavior of ML inference workloads, we need to use a power management method that relies on software-agnostic hardware measurements. The most popular approach is to use *power consumption* as a metric to make the power management decision. SLURM [33] is a widely-used cluster management system with a built-in heuristic power manager. The algorithm works by re-distributing the power that a node does not use, e.g., because of low demand, to other nodes to potentially improve their throughput. However, SLURM’s power management algorithm does not take into account the temporal power consumption patterns and power/performance characteristics of the workload. DPS [34] improves over SLURM by prioritizing nodes using their *power dynamics* – the recent history of power consumption.

However, in power-limited environments, power consumption does not provide information on whether the node is currently compute-bound or memory-bound. This is because both compute-bound and memory-bound nodes are likely to consume up to their power limits when the total limit is low. Therefore, we need a more intelligent metric that identifies compute-bound and memory-bound nodes to maximize the system performance in power-limited environments.

Moreover, SLURM and DPS are *centralized* control algorithms that need to interact with all nodes in the cluster to make power management decisions. Such centralized control methods are not scalable — as it will take a very long time just to communicate with all the nodes in a large cluster. For instance, the default control period SLURM’s power management algorithm is 60 seconds, which is too slow to effectively handle rapidly changing ML workloads.

To our knowledge, there is no existing power management system that is 1) hierarchically structured, 2) software agnostic, and 3) intelligent enough to identify workload characteristics. PowerGrad is a novel power management framework that satisfies all three requirements.

3.6 DISCUSSION

We conclude this chapter by considering a few issues.

Applying PowerGrad to GPUs. The key idea of PowerGrad is estimating the performance gradients using runtime hardware counters. This idea is not specific to CPUs and can be applied to GPUs as well. However, it is not yet possible to apply PowerGrad to existing GPUs because they do not support runtime measurements of hardware counters. Once GPUs and ML accelerators implement interfaces to monitor hardware counters at runtime, it will be possible to apply PowerGrad to GPU/accelerator clusters.

Applying PowerGrad beyond ML inference applications. Since PowerGrad only relies on software-agnostic hardware measurements, it can be easily applied to any type of application. Also note that the Gradient Estimator is not specifically built for ML applications, as the PPEP coefficients are trained using Parsec benchmarks. This work focused on ML inference because: 1) ML inference is one of the most power demanding applications in contemporary datacenters and 2) existing methods serve well for traditional non-ML workloads. There are other types of services that share similar characteristics with ML inference, such as microservices and function-as-a-service (FaaS). Applying PowerGrad to such applications would be an interesting future work.

Applying PowerGrad beyond the target CPU architecture. The performance gradient ($\partial BIPS/\partial P$) is a universal metric that can be applied to any processor architecture. Hence, PowerGrad is applicable to *heterogeneous clusters* that use a mixture of processors. The only requirement is to build a Gradient Estimator for each processor architecture. Then, the Local and Hierarchical Controllers can allocate power between the processors using the gradient estimation, without any controller modifications.

Software overhead of PowerGrad. The algorithms of PowerGrad controllers are relatively simple and the regression coefficients are trained off-line with Parsec benchmarks. Most of the software overhead of PowerGrad comes from the Gradient Estimator when collecting the hardware performance counters. This is because of expensive system calls to access the hardware monitoring interface. Still, the Gradient Estimator takes less than 1ms to execute at every 100ms control period. As a result, PowerGrad consumes less than 1% of the CPU processing power.

One straightforward way to reduce the overhead is more integration with the hardware. For example, BlitzCoin incorporated the power management algorithm into the SOC architecture for faster and more accurate control [76]. Likewise, PowerGrad can be integrated into hardware in a simple and modular way: incorporating the Gradient Estimator into the hardware. A hardware-implemented Gradient Estimator can improve both its speed and its

accuracy, because it can access detailed hardware information at much less cost. We only need a software interface to report the performance gradients to make it compatible with the rest of the framework. With this support, it would be possible to reduce the control period and enable faster PowerGrad controls.

CHAPTER 4: FRIENDLYFOE: ADVERSARIAL MACHINE LEARNING AS A PRACTICAL ARCHITECTURAL DEFENSE AGAINST SIDE CHANNEL ATTACKS

4.1 BACKGROUND

4.1.1 Side Channel Analysis with ML

The goal of side channel analysis is to recover sensitive information from a victim’s execution. Such analysis has been successfully applied to exfiltrate information from structures like caches [77, 78], branch predictors [79], and interconnects [37], as well as physical signals of computers like power [45, 80] and electromagnetic (EM) emanations [81].

Recently, attackers have used ML classifiers for signal analysis, leveraging their pattern recognition capabilities [38, 82, 83, 84, 85]. ML-based side channel analyses have recovered sensitive data from encryption modules [83], caches [86], power measurements [35, 84, 87, 88, 89], EM emissions [90] and on-chip interconnects [37]. These classifiers, particularly DNNs, can automatically identify information-carrying patterns, overcoming simple defenses like noise addition or signal misalignment [38, 40]. Hence, there is an urgent need to develop countermeasures that are effective against even the strongest ML-based attackers.

4.1.2 Adversarial Machine Learning (AML)

Recent studies have found that it is possible to generate *adversarial examples* [91] against an ML classifier which, with very small perturbations (imperceptible to the human eye), bring drastic changes to the classification outcomes. For example, adding an imperceptible noise to an image of a panda can cause a DNN to misclassify the image as that of a gibbon [91].

Figure 4.1a shows how Adversarial Machine Learning (AML) works. Given an input sample (e.g., an image), a *Generator* adds a small perturbation that causes a *Classifier* to misclassify the image to a wrong label. If both the generator and classifier are DNNs, they can be trained *adversarially* [46]—i.e., the generator is trained to create better perturbations, while the classifier is trained to predict the correct label despite the perturbation. In the best case, after the training, the generator’s perturbations are *transferable* to other classifiers [92]. This means that the generator can induce misclassification in other classifiers that have a different structure and parameters than the original classifier.

A Generative Adversarial Network (GAN) [49] is a structure that trains a generator adversarially against another neural network called *Discriminator*. The generator creates samples

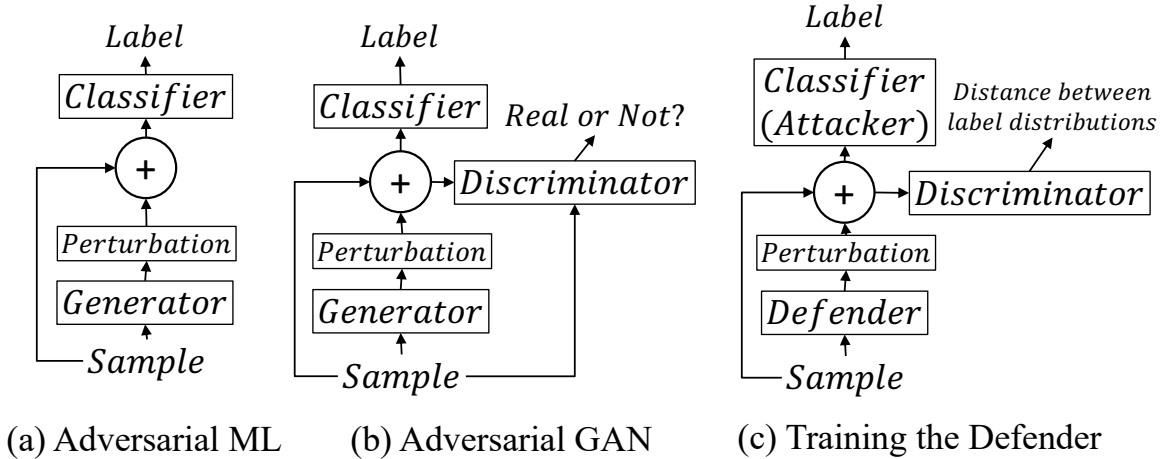


Figure 4.1: From adversarial ML (AML) to training a FriendlyFoe Defender. In our design, we call the generator *Defender*.

in a certain domain (e.g., images) and the discriminator tries to distinguish whether a sample is from the generator (fake) or from the real dataset (real). A common practice is for the discriminator to estimate the differences between the fake and real sample distributions using the Wasserstein distance [93]. By GAN training, the generator learns to create realistic outputs that can deceive the discriminator.

The Adversarial GAN (AdvGAN) structure introduced by Xiao et al. [46] combines the previous two concepts for maximum generator effectiveness. This structure, shown in Figure 4.1b, includes a generator, a classifier, and a discriminator network that are trained adversarially. The classifier, generator, and discriminator can be any type of DNN, where the choice is often based on the data type. For images, a Convolutional Neural Network (CNN) [94] is often used, while to detect sequential patterns, a recurrent neural network (RNN) such as a Long Short-Term Memory (LSTM) [95] or a Gated Recurrent Unit (GRU) [96] is used.

4.2 FRIENDLYFOE

This chapter shows that on-the-fly AML can be a generic, practical, and effective architectural technique to obfuscate side channel signals. Our approach, called *FriendlyFoe*, uses a DNN called FriendlyFoe *Defender* to obfuscate a signal coming out of an architecture side channel. Such signal is composed of a set of samples $\{x_1, x_2, \dots\}$. Successful obfuscation involves adding a perturbation (i.e., noise) p to each of the samples such that, with the resulting signal $\{x_1 + p_1, x_2 + p_2, \dots\}$, even advanced ML classifiers that have been trained

using noisy signals are unable to extract information from the signal.

FriendlyFoe is a general technique. It can be applied to a side channel if the side channel leaks information through a signal measurable over time, an attacker applies pattern recognition to decode the secret in the signal, and there exists a method for the Defender to actuate on the signal. In this work, we demonstrate FriendlyFoe in two side channels: memory contention and system power. There are many other side channels that could be used, such as network traffic that enables website fingerprinting [97] or PCIe latency that allows keystroke detection [98].

4.2.1 How to Apply FriendlyFoe

FriendlyFoe can be applied in several ways. Table 4.1 lists some of the attributes that determine how to apply FriendlyFoe, and whether they are affected by the side channel, the victim application, or the system constraints. We consider each in turn.

Table 4.1: Attributes that affect how to apply FriendlyFoe.

Attribute		Affected by		
		Side Channel	Victim Appl.	System
General	Signal recorded	✓		
	Actuation on signal	✓		✓
	Area, power, latency			✓
Signal	Sampling period	✓		
	Length (in # samples)	✓	✓	
Other	Actuation grain size	✓		✓
	Actuation timing	✓		
	Number of classes	✓	✓	

General Environment. Important attributes that determine how to apply FriendlyFoe are the type of recorded signal, the actuation capability to perturb the signal, and the hardware cost allowed for the FriendlyFoe module (area, power, latency).

Signals. Two attributes of the signal that affect a FriendlyFoe Defender design are the sampling period (the time between each sample), and the signal length in number of samples. The sampling period is determined by the side channel, and impacts the expected response time of the Defender. Typically, the Defender takes the set of most recent samples and generates the noise to add to a few upcoming samples before the attacker can observe the next sample. Consequently, if the period is short, the Defender has to make decisions

quickly. The signal length is a characteristic of the victim application and the side channel. It determines the capacity of the Defender: long signals often require large DNNs to manage the long-term pattern, increasing the hardware cost.

Other Attributes. As shown in Table 4.1, there are other attributes that determine how to apply FriendlyFoe. For example, one may have to sub-sample the signal, adding noise to only one out of N samples. We call this approach *coarse-grain* actuation. Additionally, if responding before the next sample is outright impossible, one may generate noise to be applied to a future sample, delaying the actuation. We call this attribute actuation timing. Finally, some signals may carry a binary secret, while other signals carry a multi-label secret.

4.2.2 Defender Architecture and Training

We want a Defender that minimizes both the information leakage and the perturbation level on the side channel signal. Our general design for the Defender is shown in Figure 4.2. It has two modules: a Gaussian noise shaper that takes as inputs the mean μ and standard deviation σ , and a neural network that periodically generates new μ and σ pairs for the noise shaper. Given a μ and σ pair, the noise shaper generates N samples from the resulting Gaussian distribution that will become the next N samples of the *Target* signal.

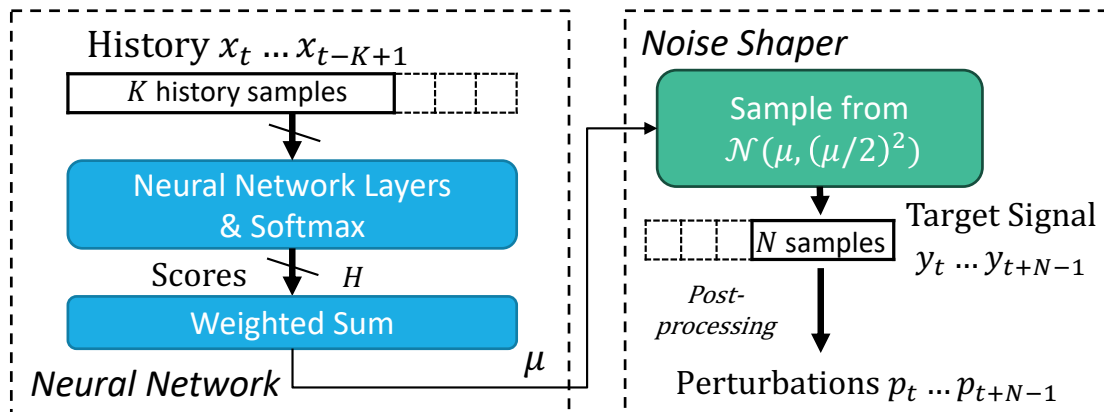


Figure 4.2: FriendlyFoe Defender architecture.

The noise shaper generates target signal values and not just perturbation values because generating perturbations using a Gaussian distribution leads to a signal that is easily compromised. Also, this target signal is not the final signal. If y_t is the value of the target signal at time t and x_t is the measured value of the system at time t , then a post-processing step computes the perturbation value p_t that must be added to the measured signal value to obtain the final signal value as $p_t = y_t - x_t$. If a side channel signal cannot allow negative

perturbations (e.g., a timing side channel), only positive or zero perturbations are added to the measured signal value.

After the N samples are generated, the neural network determines a new μ and σ pair. In our experiments, we set $\sigma = \mu/2$ because we empirically find that it is hard for neural networks to learn to generate good σ values. The likely reason for this is that the effect of the standard deviation is zero on average. Therefore, it becomes hard to learn good values of the standard deviation in large-batch DNN training. Overall, the neural network only learns to generate sequences of μ values that minimize both the information leakage and the perturbation level of the side channel signal.

To pick a μ value, the neural network takes the history of K past samples of the signal. This vector is passed through a series of hidden neural network layers, followed by a softmax layer. Assume that the softmax layer has H outputs $w_1 \dots w_H$. The neural network computes μ as $A \sum_{i=1}^H (\frac{i}{H} w_i)$, where A is a configurable hyperparameter to adjust the amplitude. The μ value obtained from the neural network is then applied to the noise shaper to generate the next N target signals. This process repeats every N samples.

The designer chooses the values of the subsampling window size N , the number of past samples read K , and the number of hidden neurons H to meet the appropriate area, power, and latency constraints. A smaller N enables finer control of the noise pattern, allowing the pattern to change more frequently. Increasing H increases the capacity of the model to handle more patterns. A higher K gives more visibility to the network to protect long-term patterns. However, these changes also increase the compute costs.

We recommend setting N first, based on the relative speed of the Defender module and the side channel signal. For example, if the Defender module is implemented on an FPGA, whose frequency is lower than the frequency of the memory system we are trying to defend, then N is adjusted to be at least the ratio of the two frequencies. Then, the designer can configure H and K to match the target implementation cost. Finally, to choose neural network layers, we recommend simple feed-forward layers when the signal patterns are short and we desire a low hardware cost. For long patterns, we may need to use RNNs [95, 96], 1-D CNNs [99], or attention-based Transformers [12].

To train the Defender, we propose the GAN structure shown in Figure 4.1c, inspired by Adversarial GAN [46]. In this design, the generator is the Defender and the classifier is the attacker. The Defender adds a perturbation to a signal and passes the noisy signal to the discriminator and classifier networks. The classifier tries to make a correct guess. The discriminator tries to estimate the Wasserstein distances [93] between the different label distributions, acting as another type of classifier. The networks are trained adversarially: the Defender is trained to perturb signal so that the classifier produces a wrong label; the

classifier is trained to identify the correct label; and the discriminator is trained to recognize the difference between the different labels. The classifier and discriminator can be any type of DNN. They can be large and sophisticated, as they are used only for the training phase; only the Defender is deployed to the system.

4.2.3 Resisting Adaptive Attacks

The Defender is trained using a particular classifier and input data set. The operation is shown in Figure 4.3a, where classifier and data set are called *Classifier 1* and *Train 1*, respectively. However, the Defender must also be effective against a variety of other classifiers that the attacker may use at runtime. Specifically, the attacker may use a *deployed* Defender to train a potentially stronger classifier (*Classifier 2*) with a new input data set (*Train 2*) (Figure 4.3b). The attacker’s goal is to become more effective against the Defender. Then, the attacker will use the trained *Classifier 2* to attack deployment runs with real (i.e., *Test*) input data sets and perform accurate predictions of the secret data (Figure 4.3c).

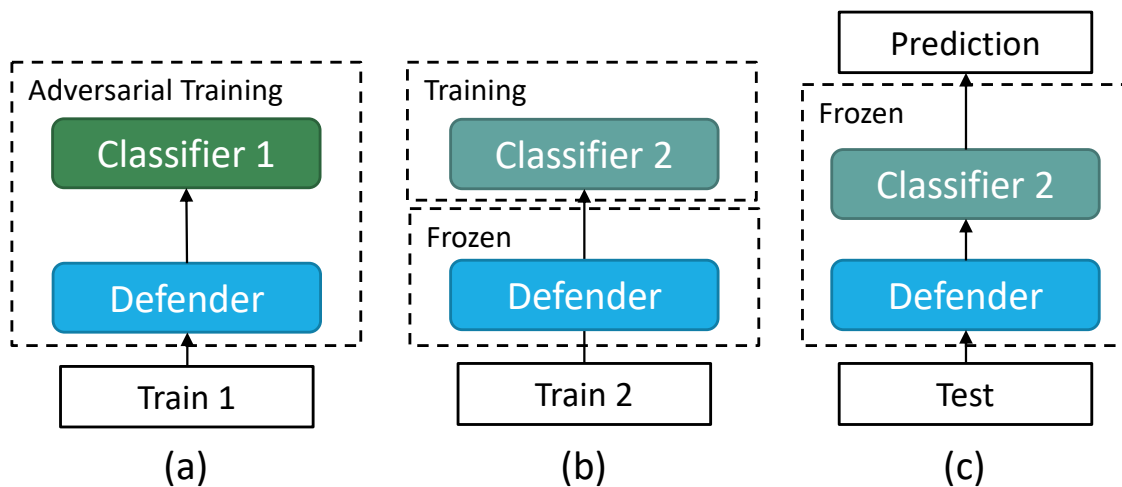


Figure 4.3: Training and deploying a FriendlyFoe Defender.

A Defender needs to be effective against classifiers which it has not been trained for. This ability is called *transferability*. In Figure 4.3, while the Defender has been trained only against Classifier 1, it must be effective against Classifier 2. For example, Classifier 1 may be a CNN and Classifier 2 a newly-trained CNN, or a different ML model such as an RNN or a Support Vector Machine (SVM) [100].

4.3 TWO TARGET SIDE CHANNEL ATTACKS

To show the effectiveness of FriendlyFoe, we examine attacks using two different side channels: memory contention and system power. Their characteristics are shown in Table 4.2. In this section, we outline the two use cases and, in Section 4.4, describe our proposed FriendlyFoe workflow, emphasizing the differences between the two use cases.

Table 4.2: Two FriendlyFoe use cases.

Attribute	Use Case 1	Use Case 2
Side channel	Memory contentn	System power
Perturbation	Introduce stalls	Add compute threads & idle time. Change freq.
Victim application	Crypto	PARSEC
Implementation	Hardware	Software
Time between samples	A few ns	20 ms
Signal length	28–75 samples	500 samples
Actuation timing	Now	Later
Classes	Binary	Multi-label

4.3.1 Threat Model

Memory-contention side channel. We instantiate an attack based on the work of Paccagnella et al. [37]. We consider a chip multiprocessor where the victim runs on one core and the attacker on another. The victim executes code whose memory accesses depend on secret bits; the attacker repeatedly accesses the main memory and measures the latency of its own accesses. The latency observed by the attacker’s accesses is affected by main-memory contention induced by the victim. Depending on the pattern of memory access latencies observed, the attacker can deduce the secret bits.

A good target for this attack is the loop inside a crypto application where, in each iteration, the memory accesses depend on one bit of an encryption key. Following the work of Paccagnella et al., we assume that the administrator has configured the system to clean the victim’s cache footprint on context switches, and that the attacker can interrupt the victim using preemptive scheduling techniques. In this case, one can time the attacker to interrupt the victim at the beginning of each iteration. Then, we simulate cache cleaning by calling `clflush` on the victim’s cache lines, which evicts them from all cache levels. Then, as the victim executes the loop iteration with a clean cache and is forced to make main-memory

accesses, the attacker accesses main memory and measures the load latency. With this design, the attacker can deduce the value of the bit in the iteration from the memory access latency measurements.

In our implementation, we run two crypto applications: the RSA [101] and EDDSA [102] algorithms. The trace of memory access latencies seen by the attacker during one iteration is the side channel signal, and the secret bit of the iteration is the target label. This is a binary classification problem.

We do not consider cache attacks. Caches can be secured by existing methods. For example, the last-level cache (LLC) can be partitioned into different security domains, so that the attacker cannot observe the victim’s LLC access. Also, simultaneous multithreading can be disabled, so that private caches are not shared by multiple threads. Main memory remains shared and vulnerable.

Application power side channel. This attack is based on the work of Pothukuchi et al. [45]. It considers a chip multiprocessor running one of several victim applications. The attacker periodically measures the chip power consumption using the Intel RAPL interface [66]. Based on the measurements over time, the attacker tries to deduce which application is running. We run PARSEC 3.0 [75] applications. We assume the attacker knows when a new application starts its execution.

4.3.2 Defender for Memory Side Channel

The envisioned Defender is in a hardware unit in the memory controller (MC) that records the time when a load arrives at the MC and the time when the main memory produces the requested data. To obfuscate the latency of the attacker load access, the Defender may stall the returning data for a certain time period. We assume that every main-memory load transaction must pass through the MC and that its access latency is accurately measured in cycles.

It would appear that the time between samples is the time between loads from a core arriving at the MC. However, since there may be multiple attacker cores, the Defender has to process every single load arriving at the MC—irrespective of the source core. As a result, as shown in Table 4.2, the time between samples can be a few ns. The signal length is 28–75 samples per iteration of the algorithm. The Defender dynamically generates a delay to be added to the current memory access—not to a future one.

4.3.3 Defender for Power Side Channel

The envisioned Defender is a software process running in privileged mode in one or more cores. We assume that the OS and hardware power and performance measurements are all trusted. The Defender measures the power consumption using RAPL, and spawns compute-intensive threads, adds idle time, or changes the frequency to distort power. The set-up is based on the one described in Maya [45]. Maya used a randomized mask generator to distort power; in this work, we replace it with a FriendlyFoe Defender.

As shown in Table 4.2, the sampling period is 20 ms, the default period of Maya. For the applications considered, the signal length is about 500 samples. Because the Defender makes a decision only after measuring the current power, it can only affect future power samples. Finally, in this attack, the classifier picks one of multiple labels, which corresponds to the application run.

4.4 FRIENDLYFOE WORKFLOW

In this section, we describe the workflow for FriendlyFoe, which we apply to the two use cases.

4.4.1 Designing the Defender Network

We design the Defender network based on the architecture of Section 4.2.2. For the memory side channel, we assume an FPGA implementation. We set the subsampling window N to 8, as determined by the speed of FPGA DNN accelerators (200Mhz [103]) and the frequency of DDR4-3200 memory (1600Mhz). We set the history length K and the hidden neuron size H so that the area and power of the Defender are less than 1% of the area and TDP, respectively, of the multicore chip. The actual values used are $K=16$ and $H=16$. The Defender’s perturbation values must be positive numbers of cycles to add to the latency of memory accesses. Hence, negative values become zero. Note that we could use a custom circuit to implement a Defender with a smaller N and a higher frequency. However, such a Defender would consume more power and area.

The Defender computes new target samples at every $N = 8$ memory controller accesses. To support this actuation frequency, the Defender is implemented in hardware and has relatively few hidden neurons. To reduce the hardware cost, we quantize the network parameters (weights and biases) to 16-bit floating point (FP16) numbers. While 8-bit integer (INT8) is the standard for quantization, we find that an FP16 model with fewer neurons has better

security than an INT8 model with more neurons. Further, our Defender has two layers of 16 neurons each. As a result, it has 528 parameters in total: 512 for the two 16×16 weight matrices of the two layers, and 16 for the bias addition in between.

The Defender for the power side channel measures and actuates on power every 20 ms. Hence, we use $N = 1$ because we have plenty of time between samples. In addition, since the response time is not very critical, we implement it in software and use $K = 32$ and $H = 64$. This makes the network compute latency less than 1 ms, which is small compared to the 20ms sampling period. The outputs can be positive or negative numbers, which correspond to increasing or decreasing the power consumption, respectively. This Defender does not need any quantization since the latency is already low even with an FP32 software implementation. It has 6208 parameters in total: 6144 for 32×64 and 64×64 weight matrices, and 64 for the bias addition in between.

A summary of the Defender configurations is shown in Table 4.3. The implementations are detailed in Sections 4.4.4 and 4.4.4.

Table 4.3: Defender configurations for two side channels.

Parameter	Memory contention side channel	Application power side channel
Window size N	8	1
History length K	16	32
Hidden neurons H	16	64
Parameter precision	FP16	FP32
Neural layer type	Feed-forward	Feed-forward
Parameter count	528	6208

4.4.2 Training the Defender

Once we have designed the Defender and collected labeled signals from the side channel, we train the Defender using a GAN network. As shown in Figure 4.1c, training is performed with three networks: the Defender, the classifier, and the discriminator.

Figure 4.4 shows how we train the networks. The Defender takes K samples of history as input and produces the perturbation values to be applied to N future samples. Specifically, at step t , the Defender reads the most recent K samples $\{x_{t-K+1}, \dots, x_t\}$ and produces perturbations for the next window, which has N samples, i.e., p_t, \dots, p_{t+N-1} . At the beginning of the next signal window, i.e., at time $t + N$, the Defender is fed K samples once

more $\{x_{t+N-K+1}, \dots, x_{t+N}\}$ and produces perturbations for the next window after that, i.e., $p_{t+N}, \dots, p_{t+2N-1}$. After processing all the samples in x in this way, the entire signal and generated perturbations are added—eliminating negative perturbations in the memory side channel—and passed to the classifier and discriminator. Then, the classifier and discriminator produce their predictions, and the three networks compute their loss functions and proceed to backpropagation. Then, the whole process repeats for another input signal.

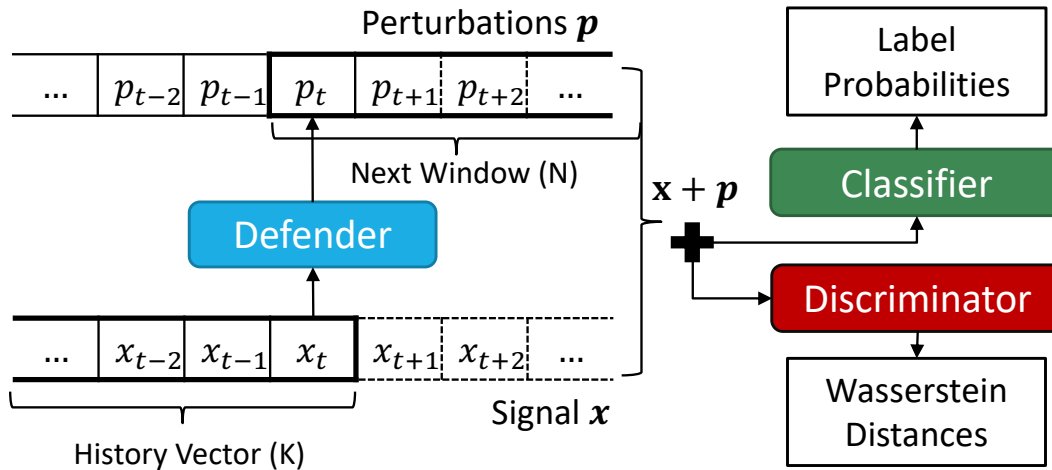


Figure 4.4: Training the networks in the GAN structure.

The three networks are adversarially trained simultaneously, with different loss functions. The classifier is trained by the cross-entropy loss between its prediction probabilities and the true label. The discriminator is optimized by Wasserstein loss [93], which tries to maximize the Wasserstein distances between distributions.

The Defender is trained with three losses: (1) the Kullback-Leibler divergence [104] between the classifier output probabilities and the uniform probability, (2) the negated Wasserstein loss of the discriminator, and (3) the L2-norm of the perturbation. The first loss optimizes the Defender to confuse the classifier, while the second loss optimizes it for perturbed outputs that are indistinguishable to the discriminator. Finally, the L2-norm loss minimizes the perturbation level to lower the overhead caused by the Defender’s perturbations. The loss can be hinged [105] by a constant to adjust the allowed perturbation level.

4.4.3 Assessing Defender Transferability

Recall from Section 4.2.3 that, to be useful, a Defender trained with a certain classifier must be transferable to other classifiers. Consequently, after we train a Defender, we select

various types of DNN networks for the classifier and, for each of them, perform the two steps shown in figures 4.3b and c. Then, we report the highest classification accuracy attained by any classifier. This is the worst case for the Defender.

Theoretically, it is impossible to verify the minimal security guarantees provided by a Defender because one cannot evaluate it against every possible classifier. However, this transferability analysis for a variety of classifier architectures often provides empirical assurance that the Defender is transferable.

4.4.4 Architecting the Overall System

While the previous steps of the workflow are common for different uses of FriendlyFoe, the next step, which involves architecting the whole system, is use-specific. Depending on the side channel addressed, applications used, or systems targeted, the implementation is different. For this reason, in this section, we consider the implementations of our two FriendlyFoe examples separately.

Defender for the Memory Contention Side Channel

We implement the Defender in hardware in the memory controller (MC). As shown in Figure 4.5, when a load request arriving at the MC from the network is sent to the DRAM module, the hardware stores its core ID ($coreid_{in}$), its address ($addr_{in}$), and the current time (t_{in}) in an Input Buffer. When the memory returns a response at t_{out} with an address ($addr_{out}$) and core ID ($coreid_{out}$), two operations occur in parallel: 1) the response is deposited in a Delay Buffer where it is delayed by a precomputed perturbation value, and 2) the response checks the Input Buffer for a matching entry and, on finding it, the hardware computes $\delta = t_{out} - t_{in}$ and pushes δ to a K -entry FIFO History Vector for that core. Every time that the History Vector for a core has taken in N entries, it sends its K entries to the Defender, which computes the perturbation values (p_1, \dots, p_N) to delay the next N responses to that core. From then on, the next N responses to that core arriving at the Delay Buffer are delayed by p_1, \dots, p_N . During this time, new entries continue to be pushed into the History Vector. As before, after N insertions, the History Vector sends its K entries to the Defender, starting the process again.

Most of the hardware cost of the Defender comes from computing matrix multiplications. Since the Defender has two layers of 16 neurons each, it needs hardware for 512 FP16 FMAs (Fused Multiply Adds). We estimate their cost based on Johnson’s analysis [106] for TSMC 28nm, scaled down [107] to 7nm. Assuming a single MC with dual-channel DDR4-3200

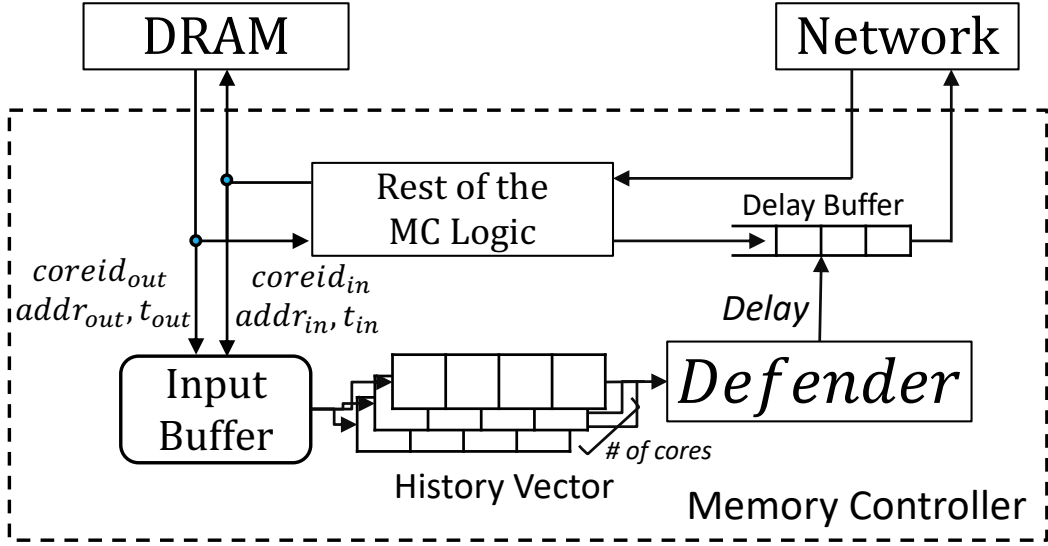


Figure 4.5: Defender for the memory contention side channel.

memory, we conservatively estimate that our added hardware consumes a maximum dynamic power of 0.11W and uses an area of $0.4mm^2$. These are less than 1% of the TDP (105W) and area ($74mm^2$) of a contemporary 7nm desktop processor like the AMD 5800x [108]. We estimate that the critical path of the Defender can be pipelined in about ten stages: two stages for each of the two feed-forward layers (one for an FMA and one for a bias addition), three stages for softmax, and three stages for the final perturbation computation. In each stage, many operations are performed in parallel. Given an MC frequency of 1.6GHz, the Defender operation takes less than 7ns. Most of this latency can be overlapped with the memory controller’s routing and flow-control logic—shown in Figure 4.5 as “rest of the MC logic”. Since the Defender can process requests at the same rate as the MC, the memory throughput remains the same.

Defender for the System Power Side Channel

We implement the Defender in software as a privileged process running on one of the cores. We build on the framework of Maya [45], a technique used to obfuscate power signals. Figure 4.6a shows the Maya framework. Maya consists of two parts. The Mask Generator creates randomized target power shapes; the Robust Controller uses control theory techniques to control the system inputs (e.g., frequency) so that the system produces the target power shapes. As shown in Figure 4.6b, our Defender is plugged into the Maya framework, by replacing a Mask Generator with a FriendlyFoe Defender.

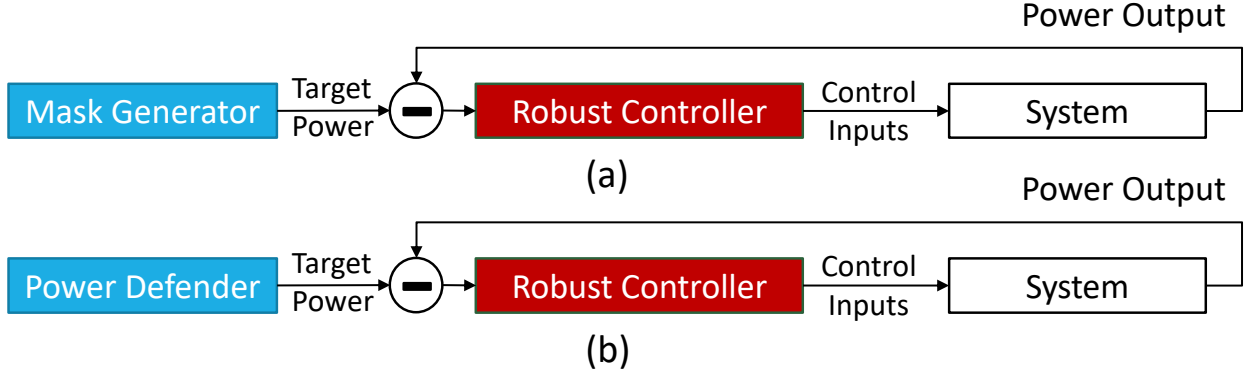


Figure 4.6: Maya [45] framework (a) and FriendlyFoe power side channel Defender plugged into the same framework (b).

4.4.5 Defending Other Victim Applications

A Defender is often effective for other victim applications beyond those for which it has been trained. We evaluate this case in Section 4.6.3.

However, in the general case, defending other types of victim applications may require the system designer to collect data for the new applications, train a Defender using the data, and deploy the new Defender. Note that the Defender architecture does not change; only the weight and bias parameters need to be updated based on the applications. Therefore, as we change the victim, we re-use the Defender module and re-program its parameters.

We can support this form of reconfigurability by placing the Defender parameters in registers or special memories, and providing a way for the software to update them across victim applications. For the system power side channel Defender, we can easily update the parameters since we deploy it via software. On the other hand, for the memory side channel Defender, we need a software-hardware interface to update the parameters.

This interface to reconfigure the Defender can use model-specific registers (MSR), microcode, or firmware. For example, if an MSR interface is used, the software can write to an MSR the ID of the parameter it wants to change and its new value. For our Defender, one MSR is sufficient to perform a full reconfiguration. This is because the Defender has 528 parameters (weights and biases) as shown in Table 4.3, and each of them is an FP16 value. Hence, the MSR takes 10 bits to specify the parameter ID and 16 bits to specify the new value. If one MSR write takes about $1\mu\text{s}$, reprogramming all the parameters of the Defender takes less than 1 ms.

With this support, one can also disable the Defender, if needed, by writing zeros to all parameters. Thus, to selectively invoke the Defender only for a security-critical application,

one can write the application-specific Defender parameters at the beginning of the application and clear them at program exit.

4.5 EXPERIMENTAL METHODOLOGY

To evaluate FriendlyFoe for our two side channels, we proceed in two steps. First, we run the victim and the attacker on a real computer, collect traces, and train a FriendlyFoe Defender. Second, we evaluate the security provided by the FriendlyFoe Defender and its execution and/or power overhead.

For the memory contention side channel Defender, we evaluate the security provided by applying the perturbations to the traces; we evaluate its performance overhead on benign applications through simulations of the architecture. Since we deploy the power side channel Defender on a real system, we evaluate its security and overheads with real system measurements.

4.5.1 Memory Contention Side Channel

We base our attack code on the ring side channel code of Paccagnella et al. [37]. We collect signal traces from a desktop with an Intel i5-7400 processor when running one victim crypto application and an attacker that repeatedly measures its memory access latency. We have two victim applications: the RSA decryption algorithm in *libgcrypt* 1.5.2, and the EDDSA algorithm in the *libgcrypt* 1.6.3 [109].

The execution traces consist of the latencies of the attacker accesses. The measurements are segmented into multiple signals. Each signal corresponds to one loop iteration of the victim application, which processes a single secret bit. Each signal contains 28 and 75 samples for RSA and EDDSA, respectively. The sample values are normalized by subtracting the median and dividing by the standard deviation of all the samples in all the signals. Each signal is labeled with the secret bit it corresponds to. For each application, we collect 12000 signals and split them into a 6:3:1 ratio to create Train 1, Train 2, and Test datasets (Figure 4.3). We use PyTorch [110] to train and evaluate the different ML networks.

We then apply four obfuscation methods: FriendlyFoe Defender, *Gaussian*, *Gaussian Sinusoid*, and *Padding to Constant*. The last three are techniques from previous work [45]. They add perturbations to create target signals of gaussian shape, gaussian + sinusoidal shape, and constant value, respectively. Recall that when the Defender produces a negative perturbation for a sample, we set it to zero.

We also evaluate whether the defense is robust against interference from concurrent threads. Hence, we conduct experiments in noisy environments for both attack and defense. Specifically, while collecting execution traces, we run random PARSEC [75] applications on the two remaining cores, filling up the four cores of i5-7400. The Defender is trained using traces from clean environments, but it is tested on the traces collected from this noisy environment.

We evaluate the performance impact of the FriendlyFoe Defender on benign applications in two environments: single- and multi-threaded. For this, we configure the Gem5 [111] simulator to model the hardware that generated the signals and run SPEC 2017 applications [112]. We compare the execution overhead induced by the FriendlyFoe Defender, Padding to Constant, and the state-of-the-art DAGguise defense [43].

To model the DAGguise defense, we mostly follow DAGguise’s setup. Specifically, in single-threaded evaluations, we run one of DAGguise’s victim applications (DocDist) and one SPEC application, for a total of two busy cores. In multi-threaded evaluations, we run four DAGguise victims (2 DocDist and 2 DNA) with four SPEC applications, for a total of 8 busy cores. We use this setup because the DAGguise defense is highly specific to the victim application that is running. Then, for fairness, we must also run the same DAGguise applications when evaluating the FriendlyFoe Defender. Since the Defender is not entirely specific to the victim and, instead, has transferability to defend other victim applications, we deploy the Defender trained to protect the RSA victim. In our experiments, we run the simulations up to one billion instructions per thread and compare the performance of the SPEC application threads.

4.5.2 System Power Side Channel

For this attack, we use the code base from Maya [113]. We collect power traces on a desktop with an Intel Xeon W-2245 processor using RAPL [66]. As victim applications, we use 6 PARSEC (blackscholes, bodytrack, canneal, freqmine, vips, streamcluster) and 4 Splash2x (radiosity, volrend, water_nsquared, and water_spatial) applications from the PARSEC 3.0 suite [75] with simlarge inputs.

The traces of the victim applications collected in an unprotected environment are the Train 1 dataset (Figure 4.3), which we use to train the FriendlyFoe Defender. Train 1 has 6000 signals. Then, we collect traces with two defense environments: a *Gaussian Sinusoid* mask generator from Maya [45], and FriendlyFoe Defender. We collect 4000 signals for each defense method. These signals are split 3:1 to the Train 2 and Test datasets. All signals are 500-samples long to record the longest workload (i.e., 10 seconds). The power values

are normalized by subtracting the idle power (30W) and dividing by the power range (160-30=130W). Each signal is labeled with the label of its dataset. Hence, the attack becomes a 10-label classification problem. The networks are trained using PyTorch and deployed to the same machine using the Maya code base and the C++ API of PyTorch.

To evaluate inter-application transferability, we split the 10 applications into two sets: the 6 original PARSEC and the 4 Splash2x applications. We train one Defender on one set and the other on the other set, and evaluate whether the Defenders can protect applications on which they have not been trained.

4.5.3 Machine Learning Models

The neural network in the Defender consists of a 2-layer MLP, a softmax layer, and a weighted sum (Figure 4.2). For the classifier in Figure 4.4, we use a 16-layer 1-D CNN, which has the best attack accuracy. For the discriminator, we use an MLP, producing one real number for binary discrimination and N numbers for the N -class cases. We consider seven classifiers to evaluate network transferability (i.e., Classifier 2 in Figure 4.3). Five of them are DNNs implemented using PyTorch: the original 16-layer 1-D CNN (CNN), a similar CNN with 25 layers (CNN-D for deep), a 16-layer CNN with double hidden neurons (CNN-W for wide), a GRU-based network (RNN), and an attention-based network (Att). The other two are non-DNNs implemented with scikit-learn [114]: an SVM and a K-Nearest Neighbor (KNN).

4.6 MEMORY CONTENTION SIDE CHANNEL DEFENSE EVALUATION

Since the two side channels are different, we start by evaluating the defense for the memory contention side channel.

4.6.1 Security Provided

We take the unperturbed *Test* signals (Figure 4.3c) for the RSA and EDDSA victims, perturb them with our defenses, and then use the 7 different classifiers described in Section 4.5.3 (trained with Train 2) to see if they can break the defenses. We consider the 4 defenses of Section 4.5.1 (FriendlyFoe, Gaussian, Gaussian Sinusoid and Padding to Constant) and, for each, vary the level of noise—i.e., the average added memory latency. This is accomplished by adjusting the amplitude A of the noise generators.

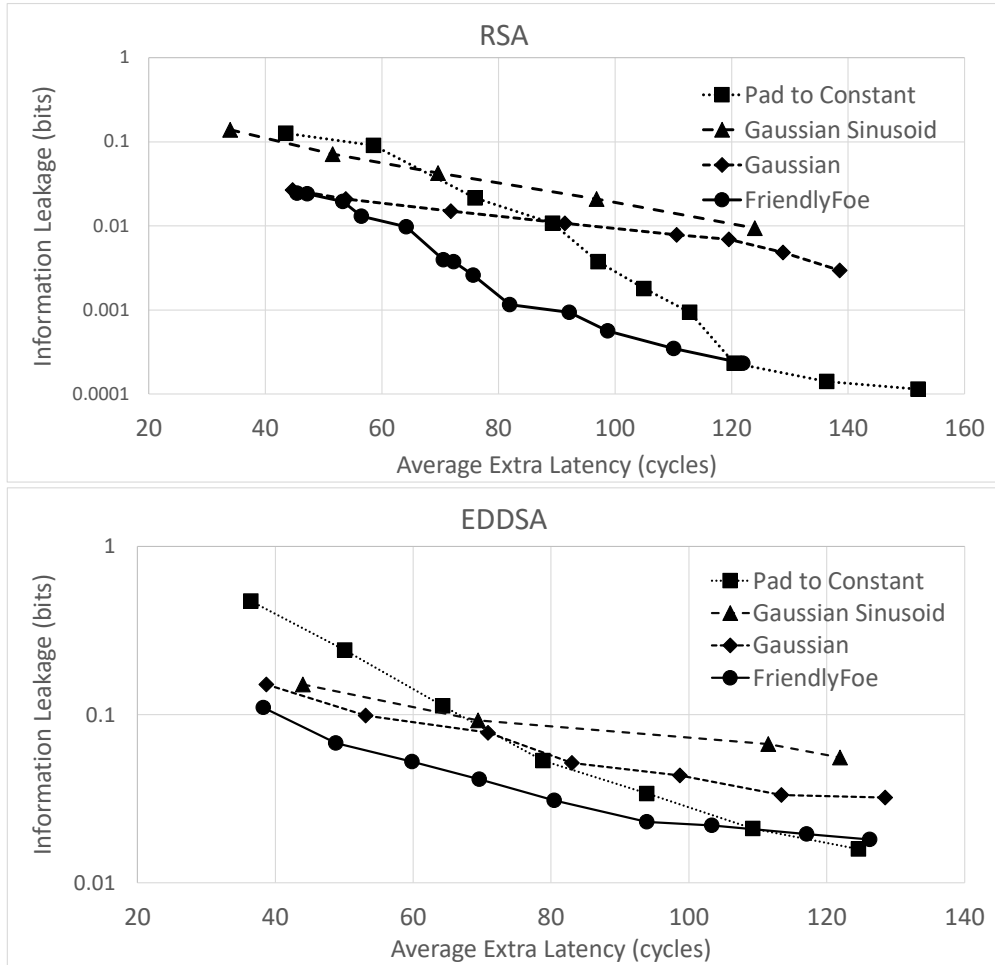


Figure 4.7: Trade-off between security and overhead of the defense methods for RSA (Left) and EDDSA (Right). The x-axis is the average extra latency a defense adds, while the y-axis is the information leakage with the best attacker in bits.

Figure 4.7 shows the information leakage in bits for RSA and EDDSA with different types of defenses and levels of noise—showing only the best attack results of the 7 classifiers. Information leakage is the mutual information [115] between the secret bits and the attacker’s predictions. If one defense has M times less leakage than another, the attacker must do M times more repetitions to obtain the same information as for the other. As the attack tries to retrieve a single bit, the leakage is 1 bit when the accuracy is 100%, and the leakage is zero when the accuracy is 50%.

Consider RSA (Figure 4.7 left). The simplest defense, Pad to Constant, is effective only when the average noise added is large (120-160 cycles). For fewer cycles added to the memory requests, e.g., 40 to 110 cycles, Pad to Constant leaks more information than FriendlyFoe. For example, for an information leakage of 0.0026 bits, Pad to Constant induces about 38%

more latency than FriendlyFoe (76 cycles for FriendlyFoe vs 105 cycles for Pad to Constant). Alternatively, for a fixed extra latency overhead of 76 cycles, Pad to Constant leaks 8.2x more information (0.0214 bits) than FriendlyFoe (0.0026 bits). The randomized defenses (Gaussian and Gaussian Sinusoid) are also ineffective. The main reason is that part of their perturbation samples end up being zero because the target sample values were lower than the measurement values. On the other hand, the neural network in the FriendlyFoe Defender dynamically adjusts the noise level to reduce both the information leakage and the execution overhead. Hence, it is the best design.

Next, consider EDDSA (Figure 4.7 right). Compared to RSA, the leakage is higher in all cases. This is expected, as the EDDSA signals are longer than the RSA signals (75 vs 28 samples), delivering more information per signal. Nonetheless, FriendlyFoe remains a better choice over other defenses. For example, for an average extra latency of 80 cycles, the leakage with the FriendlyFoe Defender is 58% of that with the next best scheme.

4.6.2 Transferability to Other Classifiers

Our FriendlyFoe Defender was trained with a CNN classifier and an MLP discriminator. We now assess the transferability of the Defender to the other classifiers. Table 4.4 shows the classification accuracy of the 7 different attacker classifiers of Section 4.5.3 for both the RSA and EDDSA victims. Recall that, in binary classification, an accuracy of 0.5 is zero leakage, while 1.0 is full 1-bit leakage. The highest accuracies are in bold, which represent the worst-case information leakage with the Defender.

Table 4.4: Classification accuracy of 7 attacker classifiers as they attack our Defender trained with the CNN classifier.

Victim Appl.	Classifier Accuracy						
	Att	RNN	CNN	CNN-D	CNN-W	SVM	KNN
RSA	0.508	0.505	0.533	0.530	0.537	0.519	0.506
EDDSA	0.560	0.549	0.603	0.601	0.598	0.501	0.499

The range of accuracy numbers in the table is 0.499-0.603. Compared to the CNN classifier, the other networks do not show higher classification accuracy, even though the Defender was not trained with them. This means that the perturbations generated after training our FriendlyFoe Defender with a CNN classifier are transferable to various other types of ML attackers. We also see that CNNs with more layers (CNN-D) or neurons (CNN-W) do not show noticeably higher accuracy than the original CNN. This observation suggests that using a bigger DNN is unlikely to break the defense.

4.6.3 Different Victim Application

In this section, we examine the *inter-application* transferability of the FriendlyFoe Defender—i.e., the security of one victim when the Defender is trained for another victim. We consider Defenders trained for the RSA or the EDDSA victims. The resulting Defenders have the same model structure; they only differ in the neural network weights and biases.

Table 4.5 shows the attacker accuracies of the most effective classifiers against our FriendlyFoe Defender trained on one victim application (*Train*) but used to defend another victim (*Target*). We also show the average added cycles per sample. The Defenders are configured to have similar noise levels by setting the same noise amplitude A .

Table 4.5: Attacker accuracies against a FriendlyFoe Defender trained on one victim (Train) but used to defend another victim (Target), along with their average added cycles.

Target Victim	Train Victim			
	RSA		EDDSA	
	Attack Accur.	Added Cycles	Attack Accur.	Added Cycles
RSA	0.537	76.0	0.542	77.3
EDDSA	0.603	81.9	0.603	80.5

In the table, we compare the data within the same row. For example, for the EDDSA row, we target EDDSA with either a Defender trained with RSA or with EDDSA. We see that, in both cases, the accuracy is the same (0.603). Further, the number of cycles only increases by 1.4 as we go from an EDDSA-trained Defender to an RSA-trained one. When targeting RSA, the attacker accuracy only increases slightly from 0.537 to 0.542 as we go from an RSA-trained Defender to an EDDSA-trained Defender, and the number of cycles only increases by 1.3. Therefore, for these applications, the FriendlyFoe Defender shows inter-application transferability. One can avoid the effort of collecting data and re-training the Defender for the new application. Note that inter-application transferability is not guaranteed and it may not apply to some victim applications. In such cases, we must re-train a new Defender and deploy the Defender using the methods discussed in Section 4.4.5.

4.6.4 Effect of Environmental Noise

Since our target side channel is shared among all cores, other applications may inject noise to the channel. This noise alters the shapes of the signals observed by both attackers and the Defender. Table 4.6 shows the attacker accuracies against the FriendlyFoe Defender trained

on a clean environment and used to defend a victim in a clean or noisy environment. The average added cycles is also shown. The noise is injected by PARSEC applications in the background (see Section 4.5.1).

Table 4.6: Attacker accuracies against a FriendlyFoe Defender trained on a clean environment and used to defend a clean or noisy environment, along with the average added cycles.

Target Victim	Target Environment			
	Clean		Noisy	
	Attack Accur.	Added Cycles	Attack Accur.	Added Cycles
RSA	0.537	76.0	0.533	79.8
EDDSA	0.603	80.5	0.539	81.6

We see that the attacker accuracy is lower in the noisy environment than in the clean one. The reason is that interference provides extra obfuscation to block information leakage. The number of added cycles in the noisy environment is only slightly higher.

4.6.5 Performance Overhead

We evaluate the performance overhead of FriendlyFoe on benign applications in two environments discussed in Section 4.5.1: single- and multi-threaded workloads. We compare three defenses: FriendlyFoe, DAGguise [43], and Pad to Constant. Due to space limitations, we only show the RSA evaluation; the EDDSA evaluation shows similar results. Recall from Table 4.5 that, with FriendlyFoe, the attacker accuracy is 0.537 and the added cycles are 76.0. We configure Pad to Constant to have a similar level of leakage as the FriendlyFoe Defender. It was discussed in Section 4.6.1 that this corresponds to a design that adds 105 extra cycles. Indeed, in Figure 4.7, the configuration in the FriendlyFoe Defender line with 76.0 cycles and the one in the Pad to Constant line with 105 cycles are at the same Y coordinate. The DAGguise configuration is the default one in their code base [116].

Figure 4.8 shows the execution overhead of the defense schemes on benign single-threaded workloads, namely running one SPEC application (plus the victim application). In this environment, minimizing memory latency is important for performance. DAGguise adds memory contention, which only indirectly increases the latency of the benign application. FriendlyFoe and, especially, Pad to Constant, directly add latency to the benign application. For these reasons, on average, DAGguise only adds 6.4% overhead, while FriendlyFoe and Pad to Constant add 8.1% and 11.1%, respectively.

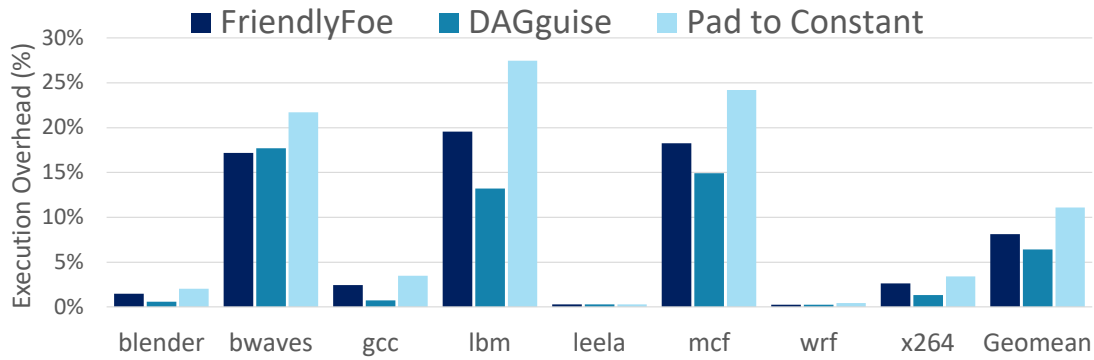


Figure 4.8: Execution overhead of defense schemes on benign applications in a single-threaded environment.

Figure 4.9 shows the execution overhead of the defenses on benign multi-threaded workloads. In this case, there are four instances of the same SPEC application running (plus four victims). In this environment, the memory throughput is critical to their performance. FriendlyFoe and Pad to Constant do not add extra memory contention. However, DAGguise’s fake memory requests introduce substantial traffic, which results in memory contention and higher latencies. On average, FriendlyFoe only adds 2.7% overhead, while Pad to Constant and DAGguise add 7.4% and 20.7%, respectively.

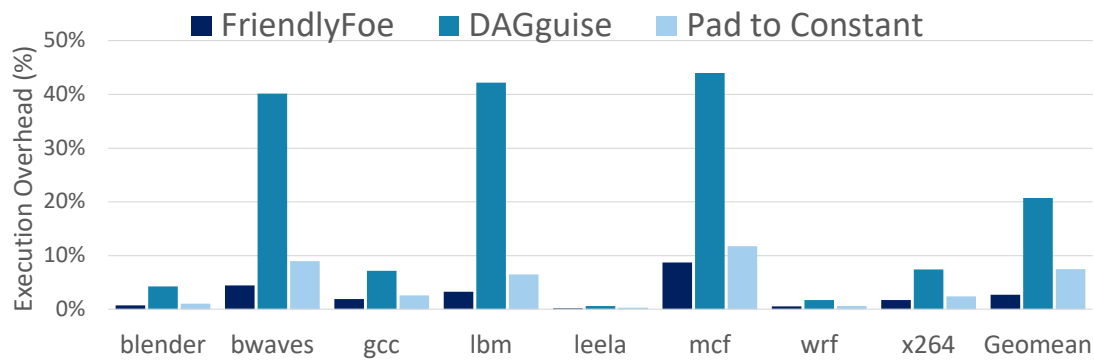


Figure 4.9: Execution overhead of defense schemes on benign applications in a multi-threaded environment.

Overall, FriendlyFoe is the most competitive scheme: it has the lowest overhead in multi-threaded environments (which are the most common ones) and modest overhead in single-threaded ones. Compared to Pad to constant, its execution overhead is 27% and 64% lower for single- and multi-threaded workloads, respectively.

4.7 POWER SIDE CHANNEL DEFENSE EVALUATION

We now evaluate the defense for the power side channel.

4.7.1 Security and Power

To evaluate the security versus power tradeoff, we compare three mask generators under the framework of Figure 4.6: None (i.e., no mask generator and therefore no defense), Maya (i.e., the state of the art gaussian sinusoid mask proposed in the Maya paper [45]), and FriendlyFoe Defender. For each of them, Table 4.7 shows the accuracy of different classifiers as they classify 10 PARSEC 3.0 applications. Since there are 10 applications, an attack accuracy of 10% corresponds to a perfect defense. The table only shows a few representative classifiers, including an MLP, which is used in the Maya paper [45], and a CNN, which consistently shows the highest accuracy. The table also shows the average application power consumption and the average application energy ($power \times time_{exec}$). The latter is relative to None, to compare the efficiency of the systems.

Table 4.7: Accuracy of different classifiers for 10 PARSEC 3.0 applications under three different defense schemes. The table also shows the application power and relative energy.

Defense	Classifier Accuracy			Avg. Appl.	Rel. Appl.
	SVM	MLP	CNN	Power (W)	Energy
None	0.729	0.653	0.917	85.9	1
Maya	0.113	0.114	0.235	95.4	1.89
FriendlyFoe	0.105	0.113	0.169	91.7	1.69

Table 4.7 shows that FriendlyFoe provides better security than the state-of-the-art approach represented by Maya. Although Maya is effective against the simpler ML networks (SVM and MLP), it is less effective against the more powerful CNN model. This is because Maya’s gaussian sinusoid simply adds random noise and does not take advantage of the application’s dynamic behavior. In contrast, our FriendlyFoe Defender shapes the noise based on the application’s behavior to maximize obfuscation.

Compared to Maya, and against the strongest attacker (i.e., CNN), which is the one that matters, FriendlyFoe reduces the attacker accuracy from 0.235 to 0.169. Converting accuracy to information leakage, it can be shown that this corresponds to leaking 0.11 and 0.03 bits per observation, respectively. Consequently, FriendlyFoe reduces the leakage of Maya by $3.7\times$. Since this is a 10-class classification problem, each label has $\log_2(10) = 3.32$ bits of information. If we assume that the attacker can force repeated re-execution

of the victim, the attacker under Maya requires $3.32/0.11 = 30.2$ repeated observations on average to retrieve the application label. The attacker under FriendlyFoe requires $3.32/0.03 = 110.7$ observations on average. In many cases, however, the attacker cannot force repeated re-execution of the victim.

The controller in both FriendlyFoe and Maya (Figure 4.6) distorts the system power by modulating CPU frequencies, injecting CPU idle times, and adding dummy compute threads. However, compared to Maya, FriendlyFoe is optimized to minimize the distortion of the power signal while still obfuscating it. Less distortion implies less inefficiency added to the system. As a result, applications consume less energy with FriendlyFoe than with Maya. This is shown in Table 4.7, where FriendlyFoe and Maya consume 69% and 89% more energy than None, respectively. In other words, FriendlyFoe reduces the energy overhead of Maya by 22.5%.

The impact of FriendlyFoe on the applications’ average power and average execution time relative to Maya depends on application behavior. The data, however, shows that, on average, applications consume less average power and take less time to execute with FriendlyFoe than with Maya. Table 4.7 shows that FriendlyFoe and Maya increase the average power consumption over None by 5.8W and 9.5W, respectively.

Figure 4.10 shows the execution time of the applications when the system is defended with FriendlyFoe and with Maya, relative to the system without protection. We see that, while there is variation across applications, applications tend to have longer execution times with Maya than with FriendlyFoe. On average and compared to None, applications take 70% longer with Maya and 58% longer with FriendlyFoe.

4.7.2 Inter-application Transferability

To verify if our FriendlyFoe Defender is transferable to other victim applications, we split the 10 PARSEC 3.0 applications in two sets (6 original PARSEC and 4 Splash2x) and train two Defenders separately on the two sets. Then, we measure the CNN attacker accuracy when targeting one set using the Defender trained with either the same set or the other set. Table 4.8 shows the attacker accuracies. Comparing the numbers within the same row, we see that the attacker accuracy on a set of applications increases only slightly if the Defender has been trained on another set of applications. It can be shown that, in both cases, the application power differences are very small. Consequently, we conclude that our defense has inter-application transferability.

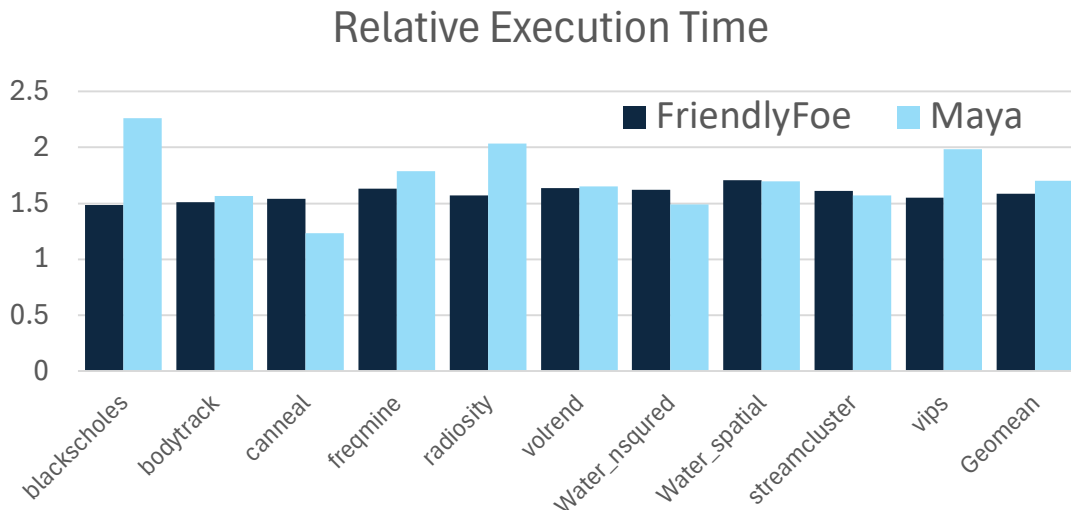


Figure 4.10: PARSEC 3.0 application execution times on a system defended with FriendlyFoe and Maya, relative to the system without protection.

Table 4.8: CNN attacker accuracies against a FriendlyFoe Defender trained on one set of applications (Train) and used to defend either the same set or another set (Target).

Target	Train Victim	
	PARSEC	Splash2x
PARSEC	0.191	0.195
Splash2x	0.289	0.279

4.8 RELATED WORK

There are two prior works that use AML as a countermeasure against ML analysis of signals [47, 48]. These schemes need to know the full trace of a signal to produce the perturbations for the signal. Such a trace is only available after the system execution. By the time these schemes start post-processing the signal, the information has already leaked to an attacker. To protect real systems, we need a dynamic defense like our proposed scheme, which uses AML to produce the perturbations on the fly.

In addition, these prior works have limitations that our technique solves. Specifically, Picek et al. [47] showed that adding small adversarial modifications can mislead ML-based power side channel attackers. However, they only use classifiers trained with *unprotected* signals, which is unrealistic. Our methodology targets attackers that are trained with signals *already perturbed* by the Defender. The work by Rahman et al. [48] showed the use of AML against website fingerprinting attacks through traffic measurement. Their defense is not very

effective, in that attackers can still achieve 38–42% accuracy for a 100-class classification.

There are two works where ML is used not to obfuscate a signal, but to help other techniques to do so. Gu et al. [117] designed a countermeasure against a power side channel attack by combining the one-pixel attack [118] with the insertion of noise instructions by the compiler. This is a compiler-based method that requires access to the source code of the victim application, while ours does not. Rijdsijk et al. [119] applied reinforcement learning (RL) to help counter ML-based attacks. The work assumes multiple existing countermeasures and uses RL to choose a right method combination dynamically. FriendlyFoe directly obfuscates the signal and does not depend on multiple existing solutions.

There are many non-ML techniques to obfuscate information in particular side channels. Two relevant ones are Maya [45] and DAGguise [43]. Maya [45] re-shapes power signals into gaussian sinusoid signals with the help of control theory; FriendlyFoe has been shown to improve Maya by replacing the signal generator.

DAGguise [43] is a defense technique that re-shapes the main memory accesses of a victim to obfuscate its behavior. Specifically, it may delay memory responses to the victim or create extra memory accesses for the victim. FriendlyFoe is different in multiple ways. First and foremost, FriendlyFoe is a general defense for various side channels (including power), while DAGguise is applicable only to contention side channels. Second, FriendlyFoe uses ML methods, while DAGguise does not. Third, focusing on the memory contention side channel, FriendlyFoe distorts the attacker’s access latencies (and therefore its measurements); DAGguise distorts the victim’s access patterns by delaying the victim’s accesses and adding fake requests. Finally, as shown in Section 4.6.5, FriendlyFoe incurs substantially less performance overhead than DAGguise in multi-threaded workloads and only slightly more than DAGguise in the less common single-threaded workloads.

CHAPTER 5: CONCLUSION AND FUTURE WORK

5.1 CONCLUSIONS

This dissertation explored two directions of research: computer architecture for machine learning computations, and machine learning for computer architecture. For the first direction, this work proposed two techniques: MeshSlice and PowerGrad. MeshSlice is a novel 2D GeMM method for efficient tensor parallelism in distributed DNN training. PowerGrad is a hierarchical power management framework for machine learning inference clusters. For the other direction, this dissertation proposed FriendlyFoe, which uses adversarial machine learning to secure computer hardware from side channel attacks.

MeshSlice solved the inefficiencies of previous 2D GeMM algorithms: it supports multiple mesh shapes, uses efficient AG/RdS primitives, and efficiently overlaps communication with computation in both dimensions. We also proposed the MeshSlice LLM autotuner, which picks an efficient 2D GeMM dataflow and uses a cost model to co-optimize the accelerator mesh shape and the communication granularity. In our evaluation, we showed that, in a simulated cluster of 256 TPUv4s, MeshSlice trains the GPT-3 and Megatron-NLG models 12.0% and 23.4% faster end-to-end, respectively, than the state-of-the-art.

PowerGrad addressed the challenge of maximizing ML inference cluster energy efficiency in environments that are power-limited (such as those with renewable energy sources where power availability is lower than the power demand) and where workload profiling is hardly possible. The main idea of PowerGrad is to identify, at runtime, how much the performance of each workload benefits from extra power, and hierarchically shift power from workloads that benefit the least to those that benefit the most. Specifically, PowerGrad dynamically computes the derivative of each compute unit’s performance over power (i.e., the performance gradient), and shifts power from lower-gradient units to higher-gradient ones.

This work demonstrated PowerGrad on a 16-node cluster with two processors per node running four different ML applications in power-limited setups. The results showed that PowerGrad is very effective. On average, PowerGrad delivers 20.3% and 20.4% shorter average and tail latencies, respectively, than the best baseline. When the power budget is as low as 55W per node, PowerGrad showed 25.7% and 26.3% shorter average and tail latencies than the baseline. Additionally, PowerGrad’s hierarchical structure makes it more scalable by adding more levels in the hierarchy. Multi-level PowerGrad was resistant to very long control periods, which is likely to happen in very large datacenters with thousands of nodes.

Finally, FriendlyFoe showed the practicality, efficacy, and generality of on-the-fly AML

as an architectural defense to obfuscate signals from architectural side channels. FriendlyFoe presents a workflow to design, implement, train, and deploy transferable FriendlyFoe Defenders for different environments. We successfully applied FriendlyFoe to thwart two very different side channel attacks: one based on memory contention and one on system power. The first example used a hardware Defender module with ns-level response time that, for the same level of security as a Pad-to-Constant scheme, has 27% and 64% lower performance overhead for single- and multi-threaded workloads, respectively. The second example used a software Defender with ms-level response time that reduces leakage by $3.7\times$ over a state-of-the-art scheme while reducing the energy overhead by 22.5%.

5.2 FUTURE WORK

There are many avenues for future work. We outline some of them here.

Multi-dimensional parallelism. The efficient 2D tensor parallelism enabled by MeshSlice can be exploited to build multi-dimensional DNN training clusters. As current LLM training clusters usually leverage a 3D parallelism (DP+PP+TP), using MeshSlice will naturally extend the cluster to 4D (DP+PP+2DTP) parallelism. It is possible to add even more dimensions to the training clusters. For example, five-dimensional clusters can be possible if we add expert parallelism (EP) of mixture-of-experts (MoE) models [120] or context parallelism (CP) when the training context is very long [11].

2D TP for other applications. While MeshSlice is designed for DNN training workloads and hardware, the algorithm is a generic 2D GeMM algorithm that can be applied to any application that performs GeMM computations. In addition, we can consider applying MeshSlice ideas to other DNN layers that are not simple GeMM computations. One example is a convolution layer, which can be implemented as a GeMM operation [121]. MeshSlice could also be used for GNNs [122, 123]: there has been work to perform 2D distributed sparse GeMMs for GNNs [124], and MeshSlice could be applied to optimize them. Also, MeshSlice can also be applied to inference. Indeed, the Wang algorithm has been used in an LLM inference cluster [125]. For inference, MeshSlice and its LLM autotuner may need to be modified, since inference computations are more likely to be memory bound. Lastly, the way MeshSlice implements a 2D distributed algorithm can be transferred to other matrix operations. For instance, 2.5D GeMM [57] also implements an efficient LU factorization algorithm using similar communication patterns.

Distributed GeMM in multi-chiplet architecture. Distributed GeMM algorithms are not only applied to distributed systems, but can also be applied to multi-core and multi-chiplet architectures. Performing efficient distributed GeMM in a multi-chiplet accelerator architecture is an interesting architectural research topic. Doing so will require a careful hardware-software co-design, as we need a new distributed GeMM algorithm that targets the hardware architecture.

Hierarchical power management for heterogeneous clusters. The performance gradient ($\partial BIPS/\partial P$) is a metric that can be applied to any type of processor architectures, including GPUs and accelerators. Hence, PowerGrad is applicable to *heterogeneous clusters* using a mixture of different processors. The only requirement is to build a Gradient Estimator for each processor architecture. Then, the Local and Hierarchical Controllers can reassign power between the processors using the gradient estimation, without any controller modifications. However, it is not yet possible to apply PowerGrad to existing GPUs and accelerators because they do not support dynamic measurements of runtime hardware counters. Once GPUs and accelerators implement interfaces to monitor hardware counters at runtime, it will be possible to apply PowerGrad to clusters with GPU or accelerator nodes.

Side channel defense using low-cost ML hardware. The FriendlyFoe Defender is designed to reduce the hardware and software implementation cost while using standard feed-forward neural network layers. Recent advances modify the network structures to lower the hardware cost for ML model computations. For instance, there are various methods that leverage lookup tables (LUT) to build hardware efficient ML accelerator designs [126, 127]. Applying such techniques may enable a FriendlyFoe Defender with less hardware cost, or a more powerful Defender with a similar hardware cost.

REFERENCES

- [1] Huggingface, “Large Language Models: A New Moore’s Law?” 2021. [Online]. Available: <https://huggingface.co/blog/large-language-models>
- [2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altschmidt, S. Altman, S. Anadkat et al., “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [3] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican et al., “Gemini: a family of highly capable multimodal models,” *arXiv preprint arXiv:2312.11805*, 2023.
- [4] A. Lewkowycz, A. Andreassen, D. Dohan, E. Dyer, H. Michalewski, V. Ramasesh, A. Slone, C. Anil, I. Schlag, T. Gutman-Solo et al., “Solving quantitative reasoning problems with language models,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 3843–3857, 2022.
- [5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman et al., “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [6] OpenAI, “Hello GPT-4o,” 2024. [Online]. Available: <https://openai.com/index/hello-gpt-4o/>
- [7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell et al., “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [8] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhume, G. Zerveas, V. 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.
- [9] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann et al., “Palm: Scaling language modeling with pathways,” *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.
- [10] R. Rojas and R. Rojas, “The backpropagation algorithm,” *Neural networks: a systematic introduction*, pp. 149–182, 1996.
- [11] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan et al., “The Llama 3 herd of models,” *arXiv preprint arXiv:2407.21783*, 2024.

- [12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [13] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, “Project Adam: Building an efficient and scalable deep learning training system,” in *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, 2014, pp. 571–582.
- [14] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer et al., “Pytorch FSDP: experiences on scaling fully sharded data parallel,” *arXiv preprint arXiv:2304.11277*, 2023.
- [15] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu et al., “Gpipe: Efficient training of giant neural networks using pipeline parallelism,” *Advances in neural information processing systems*, vol. 32, 2019.
- [16] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, “Reducing activation recomputation in large transformer models,” *Proceedings of Machine Learning and Systems*, vol. 5, pp. 341–353, 2023.
- [17] S. Pati, S. Aga, M. Islam, N. Jayasena, and M. D. Sinclair, “T3: Transparent tracking & triggering for fine-grained overlap of compute & collectives,” *arXiv preprint arXiv:2401.16677*, 2024.
- [18] C. Chen, X. Li, Q. Zhu, J. Duan, P. Sun, X. Zhang, and C. Yang, “Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 178–191.
- [19] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, “DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3505–3506.
- [20] J. Duan, S. Zhang, Z. Wang, L. Jiang, W. Qu, Q. Hu, G. Wang, Q. Weng, H. Yan, X. Zhang et al., “Efficient training of large language models on distributed infrastructures: A survey,” *arXiv preprint arXiv:2407.20018*, 2024.
- [21] L. E. Cannon, *A cellular computer to implement the Kalman filter algorithm*. PhD dissertation, Montana State University, 1969.
- [22] R. A. Van De Geijn and J. Watts, “SUMMA: Scalable universal matrix multiplication algorithm,” *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.
- [23] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni et al., “GSPMD: General and scalable parallelization for ML computation graphs,” *arXiv preprint arXiv:2105.04663*, 2021.

- [24] S. Wang, J. Wei, A. Sabne, A. Davis, B. Ilbeyi, B. Hechtman, D. Chen, K. S. Murthy, M. Maggioni, Q. Zhang et al., “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*, 2022, pp. 93–106.
- [25] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles et al., “TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–14.
- [26] SemiAnalysis, “AI Datacenter Energy Dilemma – Race for AI Datacenter Space,” 2024. [Online]. Available: <https://semianalysis.com/2024/03/13/ai-datacenter-energy-dilemma-race/>
- [27] J. Park, T. Stavrinou, S. Peter, and T. Anderson, “EMPower: The Case for a Cloud Power Control Plane,” *UW FOCI Whitepaper*, 2023.
- [28] J. Xing, B. Acun, A. Sundarrajan, D. Brooks, M. Chakkaravarthy, N. Avila, C.-J. Wu, and B. C. Lee, “Carbon Responder: Coordinating Demand Response for the Datacenter Fleet,” *arXiv preprint arXiv:2311.08589*, 2023.
- [29] Y. Li, C. Lefurgy, K. Rajamani, M. Allen-Ware, G. J. Silva, D. D. Heimsoth, S. Ghose, and O. Mutlu, “CapMaestro: Exploiting Power Redundancy, Data Center-Wide Priorities, and Stranded Power for Boosting Data Center Performance,” *IBM Research Report RC25680*, 2018.
- [30] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song, “Dynamo: Facebook’s data center-wide power management system,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 469–480, 2016.
- [31] W. Whiteside, S. Funk, A. Marathe, and B. Rountree, “Pann: Power allocation via neural networks dynamic bounded-power allocation in high performance computing,” in *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, 2017, pp. 1–7.
- [32] H. Zhang and H. Hoffmann, “PoDD: Power-capping dependent distributed applications,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–23.
- [33] SchedMD, “Slurm Power Management Guide,” 2018. [Online]. Available: https://slurm.schedmd.com/power_mgmt.html
- [34] J. Ding and H. Hoffmann, “DPS: Adaptive Power Management for Overprovisioned Systems,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–14.

- [35] H. Maghrebi, T. Portigliatti, and E. Prouff, “Breaking cryptographic implementations using deep learning techniques,” in *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 2016, pp. 3–26.
- [36] P. Lifshits, R. Forte, Y. Hoshen, M. Halpern, M. Philipose, M. Tiwari, and M. Silberstein, “Power to peep-all: Inference attacks by malicious batteries on mobile devices,” *Proceedings on Privacy Enhancing Technologies*, 2018.
- [37] R. Paccagnella, L. Luo, and C. W. Fletcher, “Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical,” *USENIX Security Symposium*, 2021.
- [38] G. Perin, B. Ege, and J. van Woudenberg, “Lowering the bar: Deep learning for side channel analysis,” *BlackHat USA, Las Vegas, NV, USA, Tech. Rep*, 2018.
- [39] D. Das, A. Golder, J. Danial, S. Ghosh, A. Raychowdhury, and S. Sen, “X-DeepSCA: Cross-Device Deep Learning Side Channel Attack,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.
- [40] E. Cagli, C. Dumas, and E. Prouff, “Convolutional neural networks with data augmentation against jitter-based countermeasures,” in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 45–68.
- [41] L. Lerman, R. Poussier, G. Bontempi, O. Markowitch, and F.-X. Standaert, “Template attacks vs. machine learning revisited (and the curse of dimensionality in side-channel analysis),” in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2015, pp. 20–33.
- [42] B. Indupriya, V. C. Jadala, and D. L. Parameswari, “A deep learning based solution for data disproportion problem in side channel attacks using intelligent sensors,” *Measurement: Sensors*, vol. 33, p. 101137, 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2665917424001132>
- [43] P. W. Deutsch, Y. Yang, T. Bourgeat, J. Drean, J. S. Emer, and M. Yan, “DAGguise: Mitigating Memory Timing Side Channels,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2022. [Online]. Available: <https://doi.org/10.1145/3503222.3507747> pp. 329–343.
- [44] Y. Zhou, S. Wagh, P. Mittal, and D. Wentzlaff, “Camouflage: Memory Traffic Shaping to Mitigate Timing Attacks,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 337–348.
- [45] R. P. Pothukuchi, S. Y. Pothukuchi, P. G. Voulgaris, A. Schwing, and J. Torrellas, “Maya: Using formal control to obfuscate power side channels,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 888–901.

- [46] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song, “Generating adversarial examples with adversarial networks,” *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, 2018.
- [47] S. Picek, D. Jap, and S. Bhasin, “Poster: When Adversary Becomes the Guardian—Towards Side-channel Security With Adversarial Attacks,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2673–2675.
- [48] M. S. Rahman, M. Imani, N. Mathews, and M. Wright, “Mockingbird: Defending Against Deep-Learning-Based Website Fingerprinting Attacks With Adversarial Traces,” *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1594–1609, 2021.
- [49] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” *Advances in neural information processing systems*, vol. 27, 2014.
- [50] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale et al., “Llama 2: Open foundation and fine-tuned chat models,” *arXiv preprint arXiv:2307.09288*, 2023.
- [51] NVIDIA, “NVIDIA Collective Communication Library (NCCL) Documentation,” 2024. [Online]. Available: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html>
- [52] NVIDIA, “DGX H100: AI for Enterprise,” 2023. [Online]. Available: <https://www.nvidia.com/en-gb/data-center/dgx-h100/>
- [53] A. S. Zekri and S. G. Sedukhin, “The general matrix multiply-add operation on 2d torus,” in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 8–pp.
- [54] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, “A systematic methodology for characterizing scalability of DNN accelerators using scale-sim,” in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 58–68.
- [55] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers et al., “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.
- [56] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, “A three-dimensional approach to parallel matrix multiplication,” *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

- [57] E. Solomonik and J. Demmel, “Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms,” in *European Conference on Parallel Processing*. Springer, 2011, pp. 90–109.
- [58] M. D. Schatz, R. A. Van de Geijn, and J. Poulson, “Parallel matrix multiplication: A systematic journey,” *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C748–C781, 2016.
- [59] Google, “Cloud TPU performance guide,” 2024. [Online]. Available: <https://cloud.google.com/tpu/docs/performance-guide>
- [60] R. Frostig, M. J. Johnson, and C. Leary, “Compiling machine learning programs via high-level tracing,” *Systems for Machine Learning*, vol. 4, no. 9, 2018.
- [61] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis et al., “The structural simulation toolkit,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.
- [62] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, “DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator,” *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.
- [63] Google, “Cloud Tensor Processing Units (TPUs),” 2025. [Online]. Available: <https://cloud.google.com/tpu>
- [64] H. Wang, L. Wang, H. Xu, Y. Wang, Y. Li, and Y. Han, “PrimePar: Efficient Spatial-temporal Tensor Partitioning for Large Transformer Model Training,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 801–817.
- [65] J. Stojkovic, C. Zhang, Í. Goiri, J. Torrellas, and E. Choukse, “DynamoLLM: Designing LLM inference clusters for performance and energy efficiency,” in *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2025.
- [66] S. Pandravadra, “Running Average Power Limit – RAPL,” 2014. [Online]. Available: <https://01.org/blogs/2014/running-average-power-limit--rapl>
- [67] B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang, “PPEP: Online performance, power, and energy prediction framework and DVFS space exploration,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 445–457.
- [68] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb et al., “The design and operation of CloudLab,” in *2019 USENIX annual technical conference (USENIX ATC 19)*, 2019, pp. 1–14.

- [69] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, “High-resolution image synthesis with latent diffusion models,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 10 684–10 695.
- [70] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.
- [71] J. Kim, J. Kong, and J. Son, “Conditional variational autoencoder with adversarial learning for end-to-end text-to-speech,” in *International Conference on Machine Learning*. PMLR, 2021, pp. 5530–5540.
- [72] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [73] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz et al., “Transformers: State-of-the-art natural language processing,” in *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, 2020, pp. 38–45.
- [74] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems,” *arXiv preprint arXiv:1512.01274*, 2015.
- [75] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *International Conference on Parallel Architectures and Compilation Techniques*, 2008.
- [76] M. Cochet, K. Swaminathan, E. Loscalzo, J. Zuckerman, M. C. Dos Santos, D. Giri, A. Buyuktosunoglu, T. Jia, D. Brooks, G.-Y. Wei et al., “BlitzCoin: Fully Decentralized hardware power management for accelerator-rich SoCs,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 801–817.
- [77] Y. Zhang, “Cache Side Channels: State of the Art and Research Opportunities,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17. Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3133956.3136064> pp. 2617–2619.
- [78] O. Aciğmez, “Yet another microarchitectural attack: Exploiting I-cache,” in *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.
- [79] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, “BranchScope: A New Side-Channel Attack on Directional Branch Predictor,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’18. Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3173162.3173204> pp. 693–707.

- [80] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, “Introduction to differential power analysis,” *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, 2011. [Online]. Available: <https://doi.org/10.1007/s13389-011-0006-y>
- [81] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, “ECDSA key extraction from mobile devices via nonintrusive physical side channels,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1626–1638.
- [82] S. Jin, S. Kim, H. Kim, and S. Hong, “Recent advances in deep learning-based side-channel analysis,” *ETRI Journal*, vol. 42, no. 2, pp. 292–304, 2020.
- [83] B. Hettwer, S. Gehrler, and T. Güneysu, “Applications of machine learning techniques in side-channel attacks: A survey,” *Journal of Cryptographic Engineering*, pp. 1–28, 2019.
- [84] G. Hospodar, B. Gierlichs, E. De Mulder, I. Verbauwhede, and J. Vandewalle, “Machine learning in side-channel analysis: A first study,” *Journal of Cryptographic Engineering*, vol. 1, no. 4, p. 293, 2011.
- [85] R. Vinaykumar, K. Soman, M. Alazab, S. Sriram, and K. Simran, “A comprehensive tutorial and survey of applications of deep learning for cyber security,” *TechRxiv*, 2020. [Online]. Available: https://www.techrxiv.org/articles/preprint/A_Comprehensive_Tutorial_and_Survey_of_Applications_of_Deep_Learning_for_Cyber_Security/11473377
- [86] T. Zhang, Y. Zhang, and R. B. Lee, “Analyzing cache side channels using deep neural networks,” in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018. [Online]. Available: <https://doi.org/10.1145/3274694.3274715> pp. 174–186.
- [87] S. Picek, A. Heuser, A. Jovic, S. A. Ludwig, S. Guilley, D. Jakobovic, and N. Mentens, “Side-channel analysis and machine learning: A practical perspective,” in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 4095–4102.
- [88] T. Kubota, K. Yoshida, M. Shiozaki, and T. Fujino, “Deep learning side-channel attack against hardware implementations of AES,” *Microprocessors and Microsystems*, p. 103383, 2020.
- [89] K. Ramezanzpour, P. Ampadu, and W. Diehl, “SCAUL: Power side-channel analysis with unsupervised learning,” *IEEE Transactions on Computers*, vol. 69, no. 11, pp. 1626–1638, 2020.
- [90] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, “Study of deep learning techniques for side-channel analysis and introduction to ASCAD database,” *ANSSI, France & CEA, LETI, MINATEC Campus, France*, vol. 22, p. 2018, 2018.
- [91] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples,” *arXiv preprint arXiv:1412.6572*, 2014.

- [92] N. Papernot, P. McDaniel, and I. Goodfellow, “Transferability in machine learning: From phenomena to black-box attacks using adversarial samples,” *arXiv preprint arXiv:1605.07277*, 2016.
- [93] M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein generative adversarial networks,” in *International conference on machine learning*. PMLR, 2017, pp. 214–223.
- [94] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.
- [95] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [96] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. 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)*, 2014, pp. 1724–1734.
- [97] G. Cherubin, R. Jansen, and C. Troncoso, “Online website fingerprinting: Evaluating website fingerprinting attacks on Tor in the real world,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 753–770.
- [98] M. Tan, J. Wan, Z. Zhou, and Z. Li, “Invisible Probe: Timing attacks with PCIe congestion side-channel,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 322–338.
- [99] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, “Wavenet: A generative model for raw audio,” *arXiv preprint arXiv:1609.03499*, 2016.
- [100] L. Wang, *Support vector machines: theory and applications*. Springer Science & Business Media, 2005, vol. 177.
- [101] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.
- [102] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, “High-speed high-security signatures,” *Journal of cryptographic engineering*, vol. 2, no. 2, pp. 77–89, 2012.
- [103] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, “DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.
- [104] S. Kullback and R. A. Leibler, “On information and sufficiency,” *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

- [105] C. Gentile and M. K. Warmuth, “Linear hinge loss and average margin,” *Advances in neural information processing systems*, vol. 11, 1998.
- [106] J. Johnson, “Rethinking floating point for deep learning,” *NIPS Systems for ML Workshop*, 2018.
- [107] A. Stillmaker and B. Baas, “Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm,” *Integration*, vol. 58, pp. 74–81, 2017.
- [108] TechPowerUp, “AMD Ryzen 7 5800X,” 2020. [Online]. Available: <https://www.techpowerup.com/cpu-specs/ryzen-7-5800x.c2362>
- [109] W. Koch and M. Schulte, “The libgcrypt reference manual,” *Free Software Foundation Inc*, pp. 1–47, 2005.
- [110] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., “PyTorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.
- [111] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj et al., “The gem5 simulator: Version 20.0+,” *arXiv preprint arXiv:2007.03152*, 2020.
- [112] J. Bucek, K.-D. Lange, and J. v. Kistowski, “SPEC CPU2017: Next-generation compute benchmark,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.
- [113] R. P. Pothukuchi, “Github repository for Maya: Obfuscating Power Side Channels with Formal Control,” 2021. [Online]. Available: <https://github.com/mayadefense/maya>
- [114] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg et al., “Scikit-learn: Machine learning in Python,” *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.
- [115] C. E. Shannon, “A mathematical theory of communication,” *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.
- [116] P. W. Deutsch, “Github repository for DAGguise,” 2022. [Online]. Available: <https://github.com/CSAIL-Arch-Sec/DAGguise>
- [117] R. Gu, P. Wang, M. Zheng, H. Hu, and N. Yu, “Adversarial attack based countermeasures against deep learning side-channel attacks,” *arXiv preprint arXiv:2009.10568*, 2020.
- [118] J. Su, D. V. Vargas, and K. Sakurai, “One pixel attack for fooling deep neural networks,” *IEEE Transactions on Evolutionary Computation*, 2019.

- [119] J. Rijdsdijk, L. Wu, and G. Perin, “Reinforcement learning-based design of side-channel countermeasures,” in *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 2021, pp. 168–187.
- [120] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, “Outrageously large neural networks: The sparsely-gated mixture-of-experts layer,” *arXiv preprint arXiv:1701.06538*, 2017.
- [121] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cuDNN: Efficient primitives for deep learning,” *arXiv preprint arXiv:1410.0759*, 2014.
- [122] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv:1609.02907*, 2016.
- [123] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” in *International Conference on Learning Representations*, 2018. [Online]. Available: <https://openreview.net/forum?id=rJXMpikCZ>
- [124] V. Bharadwaj, A. Buluç, and J. Demmel, “Distributed-memory sparse kernels for machine learning,” in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 47–58.
- [125] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean, “Efficiently scaling transformer inference,” *arXiv preprint arXiv:2211.05102*, 2022.
- [126] G. Li, S. Ye, C. Chen, Y. Wang, F. Yang, T. Cao, C. Liu, M. M. S. Aly, and M. Yang, “LUT-DLA: Lookup Table as Efficient Extreme Low-Bit Deep Learning Accelerator,” in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 671–684.
- [127] G. Park, H. Kwon, J. Kim, J. Bae, B. Park, D. Lee, and Y. Lee, “FIGLUT: An Energy-Efficient Accelerator Design for FP-INT GEMM Using Look-Up Tables,” in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 1098–1111.