UNIVERSITY OF CALIFORNIA
RIVERSIDE

Innovative Approaches to Hardware Acceleration Through Performance Analysis and Program Design

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Abenezer Yitagesu Wudenhe

September 2024

Dissertation Committee:

Dr. Hung-Wei Tseng, Chairperson
Dr. Nael Abu-Ghazaleh
Dr. Elaheh Sadredini
Dr. Yihan Sun

The Dissertation of Abenezer Yitagesu Wudenhe is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to acknowledge Professor Hung-Wei Tseng for his support as the chairman of my committee.

I would like to thank Professors Nael Abu-Ghazaleh, Elaheh Sadredini, and Yihan Sun for for their participation and constructive feedback as my committee members.

"I'm smart enough to know that I'm dumb."

— Richard Feynman

To my mother.

To my advisor, whom I owe my success, an immeasurable debt I cannot express.

To my parents and family for their belief when I had none.

To my lab-mates for their assistance.

To my friends for their fellowship.

ABSTRACT OF THE DISSERTATION

Innovative Approaches to Hardware Acceleration Through Performance Analysis and Program Design

by

Abenezer Yitagesu Wudenhe

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2024
Dr. Hung-Wei Tseng, Chairperson

The proliferation of new Artificial Intelligence (AI) and Machine Learning (ML) accelerators has enhanced the performance of domain-specific applications with tightly integrated software stacks. However, this focus often overlooks other critical applications that could benefit from these unique architectures. This dissertation examines whether AI/ML applications fully utilize these architectures, proposes an alternative to tightly integrated software stacks, and presents a novel approach to evaluating accelerators for both domain-specific and broader applications through three bodies of work. These three works collectively aim to expand the application domains of accelerators, benefiting a wide range of critical applications.

The first work presents TPUPoint, a profiling and optimization tool that assesses Google's Tensor Processing Units (TPUs). It addresses the issue of underutilized accelerators by classifying repetitive patterns into phases and identifying timing-critical operations within each phase. TPUPoint demonstrates that despite being designed for AI/ML, these accelerators may not be used to their full potential. This leads to a deeper investigation

into how TPUs can be further optimized for AI/ML workloads. Moreover, it highlights the importance of developing more sophisticated profiling tools to better understand the performance bottlenecks in TPUs. Prompting the question of whether other applications outside AI/ML might better utilize these devices.

The second work, T2SP, seeks to overcome the limitation of accelerators restricted to specific software stacks. It focuses on achieving platform-agnostic tensor computations by combining Data Parallel C++ (DPC++) and T2X, a framework that separates functional specifications from spatial mappings for architectures like FPGAs and CGRAs. This approach ensures portability, efficient hardware utilization, and ease of development by allowing users to create implementations that are not confined to specific architectures.

The final work, Accel-Bench, is a benchmark suite designed to quantify the performance gains from using hardware-accelerated functions across various application domains, both within and outside AI/ML. Accel-Bench includes ten applications that utilize hardware-accelerated functions such as GEMM, CONV, and FFT. The suite shows that applications can achieve comparable or superior performance with hardware accelerators, even with increased computational complexity.

Together, these projects provide comprehensive solutions for evaluating performance, enabling portability, and diversifying applications across domains, advancing the field of hardware-accelerated computing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Through the past decade, we have seen extensive growth within the field of Machine Learning and Artificial Intelligence. Due to greater processing power through parallelism and data management in new architectures, we have also seen growth in both variety and complexity of applications within their domain. This, in turn, has demanded even greater processing power, and continues in this cycle.

In industry, we have seen several unique architectures. This includes familiar SIMD, Vector, or Matrix based architectures such as Intel's Gaudi processor [94] and Nvidia's A100 GPUs [37]. Others pose more unique approaches such as Google's TPU's with systolic array [97], Graphcore's Intelligence Processing Unit or IPUs [104], or Cerebras [110] which has a silicon wafer containing 2.6 trillion transistors compared to the largest gpu with 54.2 billion transistors. However, many does not always mean good. their narrow focus does not lend itself to easy utilization of many accelerable domains outside of AI/ML.

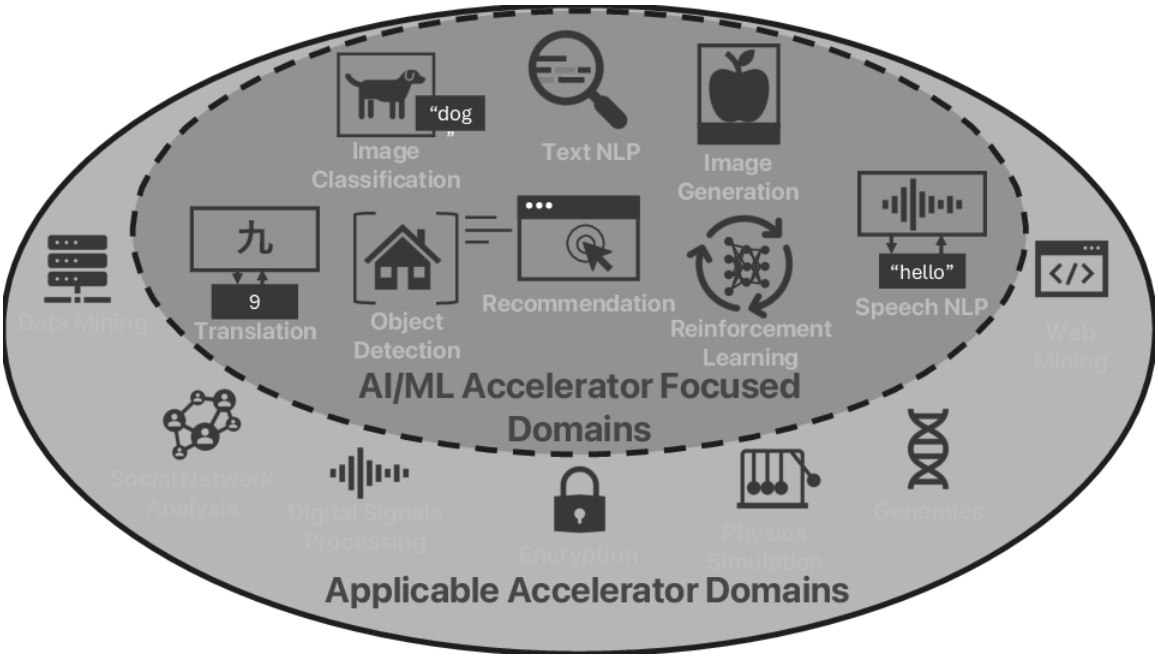

Figure 1.1: Current State of Accelerable Application Domains

Many of these accelerators bind developers to a software stack including; libraries, compilers, tools, and runtimes. Primary examples include TPUs with TensorFlow and it's XLA compiler [2], Nvidia's CUDA Compiler and associated drivers [44], and Cerebras compiler [114]. As more and more domains begin to turn to accelerators, such as genomics [183] and encryption [201], we need to assess and accommodate for these applications.

1.2 Solutions

Dr. Michael O'Boyle discusses in his article about the coming difficulties as new heterogeneous hardware innovations are held back by the construction of a new DSL and the inevitable refactorization of code [146]. Figure 1.1 demonstrates the space of growing applicable accelerator domains. This is not an all inclusive figure, but it conceptualized

that many accelerators disregard domains outside their primary focus, only considering their applicability adhoc (as displayed by the dashed border of the inner oval). Many works exist of lifting code and automating the refactorization process through methods such as static analysis, natural language processing, and machine learning. However, these are mitigating the symptoms rather than the root cause. The real challenge lies in designing a unified framework that can seamlessly integrate diverse hardware architectures. Without such a framework, the industry will continue to face fragmentation and inefficiencies.

Many of these issues can be addressed through the following:

- Software frameworks that are agnostic to the hardware architecture, but still display non-trivial utilization.
- Software frameworks enabling the expansion of new algorithms implementations for different hardware.
- Application implementation unshackled from redesigning and revisiting implementations as new algorithms develop.
- Application evaluation and insight methods that offer developers in-depth and high level insights.

1.3 Contributions

Throughout this thesis, I explore the current state of one of the most primarily used accelerators publicly available, expand a software framework to become hardware agnostic in its implementation, design and construct a new programing paradigm to accommodate

the growing fields of both AI/ML and non-AI/ML applications within three separate bodies of work. These three bodies of work are written in conference format, and can be viewed as separate or as the accumulation of multiple approaches to these issues.

This thesis makes the following contributions:

- It examines the conventional notions that applications enable perfect utilization of their domain accelerators.
- Demonstrates profiling and optimization techniques to demystify the performance of the TPU accelerator.
- Opens argument to investigate applications outside AI/ML domain for potential full utilization
- Demonstrates software stacks that can be made agnostic to accelerator hardware, rather then the conventional tightly intertwined hardware software stack.
- Provides evidence that an unbound accelerator software stack can achieve non-trivial performance to its counterparts when decoupled.
- Presents the first benchmark suite that considers a set of algorithms that allow applications beyond hardware accelerators' target domains to take advantage of the innovations of hardware accelerators.
- Presents insights into potential programming paradigms for composing performance code.

- This thesis presents a benchmark suite that can guide the development of democratized hardware accelerators.

1.4 Outline of the dissertation

The rest of this dissertation is organized as follows:

- Chapter 2 presents TPUPoint: Automatic Characterization of Hardware-Accelerated Machine-Learning Behavior for Cloud Computing. TPUPoint establishes that even when targeting AI/ML applications, some accelerators are still underutilized.

- Chapter 3 presents T2SP: Embedding a DSL in SYCL for Productive and Performant Computing on Heterogeneous Devices. Addresses the issue of ridged software stacks for accelerators through a demonstration of a hardware agnostic software stack for a domain specific language.

- Chapter 4 presents Accel-Bench: Exploring the Potential of Programming using Hardware-Accelerated Functions. A unique benchmark suite with the principle of application inclusivity through a novel API based framework and integrated simulator.

Chapter 2

TPUPoint: Auto-Characterization of an Accelerator

With the share of machine learning (ML) workloads in data centers rapidly increasing, cloud providers are beginning to incorporate accelerators such as tensor processing units (TPUs) to improve the energy-efficiency of applications. However, without optimizing application parameters, users may under-utilize accelerators and end up wasting energy and money.

This section presents TPUPoint to facilitate the development of efficient applications on TPU-based cloud platforms. TPUPoint automatically classifies repetitive patterns into phases and identifies the most timing-critical operations in each phase. Further, TPUPoint can associate phases with checkpoints to allow fast-forwarding in applications, thereby significantly reducing the time and money spent optimizing applications. Enabling more precise and effective optimization strategies.

By running TPUPoint on a wide array of representative ML workloads, we found that computation is no longer the most time-consuming operation; instead, the `infeed` and `reshape` operations, which exchange and realign data, become most significant. TPUPoint's advantages significantly increase the potential for discovering optimal parameters to quickly balance the complex workload pipeline of feeding data into a system, reformatting the data, and computing results.

2.1 Introduction

The rise of machine learning (ML) has created a strong demand for efficient ML systems designed for modern cloud-infrastructure applications [51, 52, 75, 103, 119, 118, 153, 164, 177, 136]. Because conventional, general-purpose processors and graphical processing units (GPUs) are optimized for scalar or vector operations, the modern computer architectures that rely on them waste energy when performing ML tasks. More efficient ML accelerators that rely on matrix-based neural networks (NNs) are thus gaining ground in data centers. Google's Tensor Processing Unit (TPU), which offers 70× better performance per watt than conventional GPUs, is by far the most representative case [97].

This section presents TPUPoint, an open-source toolchain[1] to characterize the behavior and optimize the performance of applications on Google Cloud TPUs. TPUPoint's profiler automatically classifies the recurrent patterns of TPU applications into phases and identifies the most timing-critical operations in each phase to inform optimization. TPUPoint can also associate each phase with checkpoints to restart an application right before

[1]You may find TPUPoint at https://github.com/escalab/TPUPoint

a target phase, and TPUPoint gives the user access to automated tools like the TPUPoint-Optimizer to examine performance changes with different configurations.

In this section, we show how TPUPoint may be used to characterize a set of popular ML workloads. We demonstrate that the iterative nature of NN models means that all ML workloads exhibit repetitive behavior that can easily be characterized via very few important phases. TPUPoint identifies time-consuming operators, such as `infeed`, `outfeed`, and `reshape`, that are commonly used among almost all NN models and are not directly related to computation; such indirect operators block the progress of computation if they cannot prepare datasets or swap out datasets fast enough.

As performance characteristics differ among heterogeneous architectural components and platforms, creating uniformly optimized ML programs is unrealistic. A more tenable approach is to automate the optimization process itself. The TPUPoint framework does this through the TPUPoint-Optimizer; the TPUPoint-Optimizer automatically and dynamically rewrites code on Cloud TPU platforms to reduce programmer effort. Our results show that optimal parameters dynamically determined using TPUPoint-Optimizer allow a reasonably written TensorFlow program to achieve at least the same level of performance as that achieved through exhaustive programmer optimizations.

By introducing TPUPoint, this section makes four key contributions:

1. It presents TPUPoint to accelerate the development and optimization of ML applications for emerging ML accelerator-based cloud architectures.

2. It validates TPUPoint functionality with a wide range of ML applications.

3. It identifies the common bottlenecks of ML applications.

4. It details a systematic approach for discovering optimal parameters for ML applications.

The rest of this section is organized as follows: Section 2.2 describes the architecture of TPUs and TPU-based cloud servers. Section 2.3 introduces TPUPoint's design. Sections 2.4 describes TPUPoint-Analyzer's implementation. Section 2.5 describes our experimental platform. Section 2.6 reviews insights gained from TPUPoint-Analyzer. Section 2.7 presents TPUPoint-Optimizer's results. Section 2.8 provides a summary of related work for context, and Section 2.9 offers concluding comments.

2.2 TPUs

Google has widely deployed TPUs in its data centers and made TPUs accessible for user applications through Google Cloud Services. This section briefly describes the capabilities and interfaces of Cloud TPUs.

2.2.1 Cloud TPUs

Google offers three different Cloud TPUs. Google uses the first-generation TPU internally for search and inference but makes the second and third-generation TPUs (TPUv2 and TPUv3, respectively) available via the Google Cloud Platform and TensorFlow Research Cloud (TFRC) program [68]. The TPUv2 chip contains two Matrix Units (MXUs), where each MXU is associated with 8 GiB of High Bandwidth Memory (HBM) to deliver a combined theoretical 45 TFLOPS of computation throughput for 200–250 W TDP. Google typically combines four TPUv2s on a single board [71].

Google does not disclose many details about the TPUv3 architecture. Nonetheless, the performance-number specifications, which include a capacity of 90 TFLOPS and 32 GB HBM for each chip, suggest that TPUv3 simply leverages more advanced process technologies to place four MXUs within the same chip while maintaining the same level of power consumption as TPUv2.

2.2.2 The Cloud TPU Hardware/Software Interface

A Google Cloud TPU is only accessible through a compute instance (Compute Engine) associated with a TPU instance. Along with the Compute Engine VM, a Google Cloud TPU requires cloud storage (Storage Buckets) for training data and model information during execution; the Compute Engine acts as a host, the TPU acts as a coprocessor, and the Storage Buckets act as persistent memory. These components comprise the Cloud TPU architecture.

TensorFlow [2] is another important part of the Cloud TPU equation. Google developed the TensorFlow framework to model and execute ML algorithms on single machines and heterogeneous/distributed systems. Google Cloud TPUs are readily integrated with TensorFlow. TensorFlow makes heavy use of Google's Protocol Buffers (Protobuf) and Google's Remote Procedural Call (gRPC). Both Protobuf and gRPC are crucial to the TensorFlow framework to allow communication to occur across TensorFlow. TensorFlow makes heavy use of Google's Protocol Buffers (Protobuf) and Google's Remote Procedural Call (gRPC). Protobuf allows for convenient data abstraction across multiple programming languages, and gRPC allows TensorFlow to share data between multiple servers and clients to facilitate execution across multiple devices. The gRPC server implements a method and

waits for client requests. A gRPC client uses an object referred to as a stub to provide a channel between the client and server. The stub handles gRPC client requests (with Protobuf) and server responses and uses efficient formats such as RDMA for communication between processes during execution. Both Protobuf and gRPC are crucial to the TensorFlow framework.

TensorFlow execution involves a client (the user), a master, and one or more worker processes. The client interacts with the master, and the master coordinates the workers. The master is responsible for handling device placement of graph nodes and partitions the graph into subgraphs to be executed by the workers. In addition to managing the entire computational graph, the master applies optimizations such as constant folding. The workers handle requests from the master, execute kernel operations, and manage communication between kernels.

Even though the Cloud TPU's implementation is not fully available to the public, registered API calls and serviceable requests are still available; a command-line tool called CLOUD-TPU-PROFILER may be used to generate a client-to-master gRPC call that requests a Cloud TPU profile for a small iteration. CLOUD-TPU-PROFILER is limited in its usefulness, however, because it cannot be integrated into training code, only permits insights to be gained post-execution, and only runs within a limited time range (and so cannot profile program execution in its entirety).

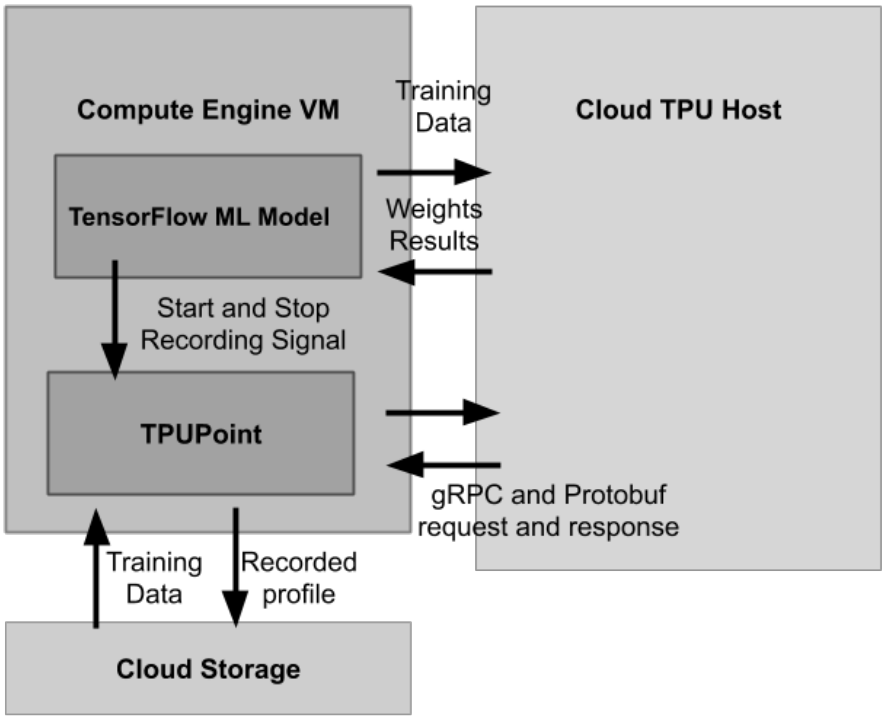

Figure 2.1: The TPUPoint system architecture

2.3 TPUPoint-Profiler: The Core of TPUPoint

TPUPoint offers a set of tools via the TPUPoint-Profiler module. TPUPoint-Profiler measures Cloud TPU performance and enables the other two elements of the TPUPoint toolchain: (1) TPUPoint-Analyzer (Section 2.4), a post-execution, offline analysis tool that identifies the most important application phase and the cause of under-utilized system components and (2) TPUPoint-Optimizer (Section 2.7), the online, automatic workload-optimization tool that dynamically adjusts and rewrites code running on Cloud TPU platforms. This section introduces the TPUPoint design and programming interface.

2.3.1 TPUPoint-Profiler Design

The complete TPUPoint toolchain consists of a set of extensions to the TensorFlow framework (the only programming interface for Cloud TPUs at this point). Figure 1 shows the interactions of the core TPUPoint-Profiler that drives TPUPoint-Analyzer and TPUPoint-Optimizer to work with a TensorFlow application.

TPUPoint creates a separate profiling thread upon initialization of the TPUPoint-Profiler. Once created, the TPUPoint-Profiler thread periodically sends profile requests to associated Cloud TPUs independently of the main TensorFlow thread, allowing TPU training to continue uninterrupted while profiling takes place. When a Cloud TPU sends a response back to the profiling thread, TPUPoint-Profiler generates a profile record containing operations along with meta-data of TPU idle time and MXU utilization provided with each response.

If the programmer intends to use TPUPoint-Analyzer, the TPUPoint-Profiler thread will create an additional recording thread to store the collected statistical information in Cloud Storage (otherwise, TPUPoint-Profiler simply buffers the profile in the host main memory). While the recording thread is storing data, TPUPoint-Profiler's profiling thread continues to request the next profile from the Cloud TPU. Reliably recording all events during a profile period can produce numerous records, as each profile can potentially include a maximum of 1,000,000 events lasting for a maximum duration of 60,000 ms in total elapsed time. By storing only statistical information in a profile, TPUPoint-Profiler reduces memory consumption and accelerates the post-processing in TPUPoint-Analyzer and TPUPoint-Optimizer. Once the TensorFlow application has completed or reached a

```
1  import tensorflow as tf
2  from tensorflow.contrib.tpu import TPUPoint as TP
3  #...
4  def main(argv):
5      #...
6      estimator = tf.contrib.tpu.TPUEstimator(...)
7      tpprofiler = TP(...)
8      #...
9      tpprofiler.Start(analyzer = true)
10     estimator.train(...)
11     tpprofiler.Stop()
12
13     if __name__ == "__main__":
14         tf.app.run()
```

Figure 2.2: Example TensorFlow code that initiates TPUPoint's profiling feature

user-specified breakpoint, TPUPoint-Profiler's profiling thread will send out the last request. All TPUPoint-Profiler threads terminate after TPUPoint-Profiler has received and appropriately saved the last profile record response to the Cloud TPUs. The number of profile records generated depend on the duration of the TensorFlow application.

2.3.2 The TPUPoint Programming Interface

The current version of TPUPoint presents a Python/TensorFlow-based front end to the programmer with backend features implemented in C++. Figure 2.2 shows example code that enables TPUPoint-Profiler in a TensorFlow application. The programmer needs to initiate TPUPoint usage by creating a TPUPoint-Profiler object (`tpprofiler` in line 7

of the example) with appropriate options. TPU training is executed though TensorFlow's high level `TPUEstimator` API (lines 6 and 10). If a programmer wishes to use TPUPoint-Analyzer to perform post-analysis, the `analyzer` flag must be set to `true` in the `Start()` function call (line 9) ; when the `analyzer` flag is set to `false`, TPUPoint-Profiler only enables TPUPoint-Optimizer. Once training is complete (i.e., `TPUEstimator.train()` has finished), TPUPoint-Profiler is halted via `Stop()` function (line 11). When post-execution analysis has been specified, as in the code example, `Stop()` will also instantiate the TPUPoint-Analyzer process for visualizing the profiling results. This implementation allows TPUPoint to profile the entire duration of an application, a feature unavailable in the CLOUD-TPU-PROFILER command line tool.

2.4 TPUPoint-Analyzer: Post-Execution Analysis

To address the challenge of deriving meaningful results from extensive profiling statistics, TPUPoint-Analyzer walks through and summarizes profiles into program *phases*. To address these challenges, TPUPoint implements TPUPoint-Analyzer's post-execution analysis. Each program phase from TPUPoint-Analyzer's post-execution processing identifies similar, repetitive program behaviors. Summarizing program behaviors into phases to facilitate analysis, visualization, and checkpointing/restarting for performance optimizations.

2.4.1 Profiling Algorithms

To reduce the TPUPoint-Analyzer search space for calculating program-behavior similarities, TPUPoint-Analyzer first leverages the step numbers that Google makes available for Cloud TPUs—step numbers that indicate coarse-grained, repetitive application behaviors. TPUPoint-Analyzer then uses these steps as the basic unit for similarity comparisons and creates visual summaries for the steps. TPUPoint-Analyzer offers three summarization methods: the conventional k-means algorithm [121, 127], Density Based Spatial Clustering of Applications with Noise (DBSCAN) [59, 171], and a lower-overhead online linear-scan (OLS) algorithm. k-means and DBSCAN run after all profiling records have been recorded, while OLS is executed during recording (hence the term "online" in it's name).

k-means: We evaluate TPUPoint-Analyzer using the k-means algorithm implementation by using three stages [174]:

1. Extract the records from all statistical profiles and aggregate records together using the TPU step numbers. For each step, we define dimensions in terms of TensorFlow operations, the accumulated number of invocations, and total durations. Using principal component analysis (PCA) for dimensional reduction [207], we have at most 100 distinct operations for frequency vector representation.

2. Try the k-means clustering algorithm on aggregated steps for values of k ranging from 1 to 15. Each run of k-means produces a clustering that partitions the steps into k different clusters.

3. For each cluster ($k = 1, \ldots, 15$), calculate the sum of squared distances of samples to cluster centers (centroids) for each value of k. Attempt to minimize the sum of squared distances while maximizing the number of clusters (k) using the elbow method.

TPUPoint-Analyzer implements k-means like SimPoint does [174, 77, 152]. SimPoint uses the Bayesian information criterion (BIC) [151] to measure the probability of clustering for a given simulation. Using instructions per cycle (IPC) as the metric, SimPoint compares using clusters rather than full simulations for analysis. TPUPoint aims to simulate complete program execution without architectural metrics such as IPC, instead employing the elbow method [188] as a heuristic to cut clustering off when improvement stops increasing significantly (i.e., when the sum of squared distances for a cluster stops improving significantly).

DBSCAN: DBSCAN [59, 171] follows the same general approach as k-means but relies on core samples of high-density clusters. DBSCAN provides an alternative method for comparison with k-means. DBSCAN also has three stages:

1. Extract the records from all statistical profiles and produce a frequency vector representation as done in k-means.

2. Apply DBSCAN on aggregated steps of 25, requiring a minimum number of samples from 5 to 200. As the minimum increases, the number of produced clusters decreases.

3. For each clustering minimum sample size, measure the ratio of the noise by counting the number of unlabeled points to the total number of points. Attempt to minimize noise while maximizing the number of required samples to form a cluster using the elbow method.

OLS: Both k-means and DBSCAN post-process all records after program execution, which requires the system to store large numbers of records and incur high computational overhead due to the dimensional complexity of each record. To address these issues, TPUPoint-Analyzer offers OLS, which identifies similar, consecutive program behaviors that approximate clustering with significantly lower overhead and reduced data-storage needs. With OLS, TPUPoint-Analyzer simply relies on records from the current step, from the previous step, and from two steps ago. OLS has four stages:

1. Extract the records from the incoming statistical profiles and group the records together using their step numbers. For each step, use all TensorFlow operations in the program as well as the accumulated number of invocations and the total duration of each operation.

2. When the program advances to another step, compare the previous step within a profile to its successor step and calculate their similarity using Equation 2.1. Equation 2.1 computes the similarity of two steps as the ratio of the intersection of the set of events from step $i-1$ and the set of events from step $i-2$ to the minimum size of the two sets, where step $i-1$ is the successor of step $i-2$. (A set of events for a step is defined as all the unique events that occur during that step.)

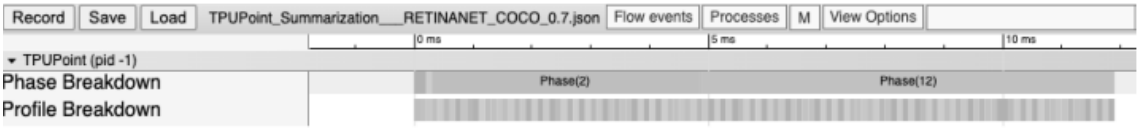

Figure 2.3: Visualization of TPUPoint profiling output

3. If the successor step is similar according to either a user-specified threshold or the default threshold (70% similarity), group the two steps together into a single phase. Otherwise, associate the later step with a new program phase.

4. Repeat the above stages and gradually aggregate consecutive steps until all steps from the stored profiles have been parsed.

$$StepSimilarity\left(Step_{i-1}, Step_{i-2}\right) = \frac{|Step_{i-1}| \cap |Step_{i-2}|}{min\left(|Step_{i-1}|, |Step_{i-2}|\right)} \tag{2.1}$$

2.4.2 Visualization

TPUPoint-Analyzer produces a JSON file to store the summarized view of application behavior. This file, along with a corresponding CSV file, contains (1) a formatted description of each phase and (2) the TPU and Host CPU operations executed during training steps. The JSON file is compatible with Google Chrome's event-profiling tool, `chrome://tracing`. Figure 2.3 shows a visualization of TPUPoint-Analyzer output for phases during TPU training from one such file. Each profile recorded is displayed as a small subsection of the overall execution time on the horizontal `Profile Breakdown` axis. Each phase identified is displayed as a larger subsection of the overall execution time on the horizontal `Phase Breakdown` axis. Figure 2.3 displays how each phase can expand over

multiple profile records, effectively summarizing the information from each profile. The time markers displayed in Figure 2.3 are not to scale, as TPUPoint-Analyzer's visualization of the profiles and phases are only a representation, meant to reduce the information a user must consume. Using Chrome's controls, a user can zoom in/out of each program phase to see more/less detail from the TPUPoint-Analyzer output.

2.4.3 Checkpointing and Restarting

Along with phases, TPUPoint records the closest checkpoint to each phase stored by the TensorFlow model. To identify checkpoints, TensorFlow compares the steps within a phase and finds the checkpoint with the smallest distance from those steps. This approach allows applications to be modified based on a targeted phase and executed without starting from step zero.

2.5 Experimental Methodology

To verify the TPUPoint-Profiler and TPUPoint-Analyzer designs and obtain initial insights to assist code optimizations, we ran a set of experiments on the Google Cloud Platform. Each instance consisted of a single host with a 16-core, 2-way SMT Intel Skylake CPU, 104 GB of main memory, and 250 GB of persistent disk [70]. To maintain implementation consistency, all instances used Docker version 19.03.1 and TensorFlow version 1.15 with TPUPoint installed. As mentioned in Section 2.2, each instance could access both TPUv2 and TPUv3—model implementations running on a single TPU instance such as TPUv2 could run on a single TPUv3 instance without code modifications. However, scal-

ing for multiple TPU implementations "requires significant tuning and optimization" [71]; to avoid any inefficient model execution, experiments were conducted only on single-TPU instances.

Table 2.1 describes the workloads we used to test and verify our designs and hypotheses. We chose publicly available workloads from the TensorFlow 1.14 TPU model library [186]: natural language processing (NLP) (BERT [191]), image generation (DCGAN [159]), question answering (Q/A) NLP (QANet[215]), object detection (RetinaNet [115]), and image classification(ResNet-50 [81]).

2.6 Observations and Insights Learned from TPUPoint-Analyzer

2.6.1 Representativeness of Phases

TPUPoint-Analyzer identifies similar, repetitive behaviors in applications and categorizes those behaviors into phases to facilitate analysis and optimization. This section discusses and compares the phases identified from the k-means, DBSCAN, and OLS clustering algorithms.

Figure 2.4 shows the clustering results for k-means with k between 1 and 15, inclusive. Each cluster represents a phase of more extensive program execution. In this case, TPUPoint-Analyzer determines that the sum of squared distances stops improving by a significant margin when k is between 4 and 6; that is, 4 to 6 clusters are sufficient to cover most program behaviors.

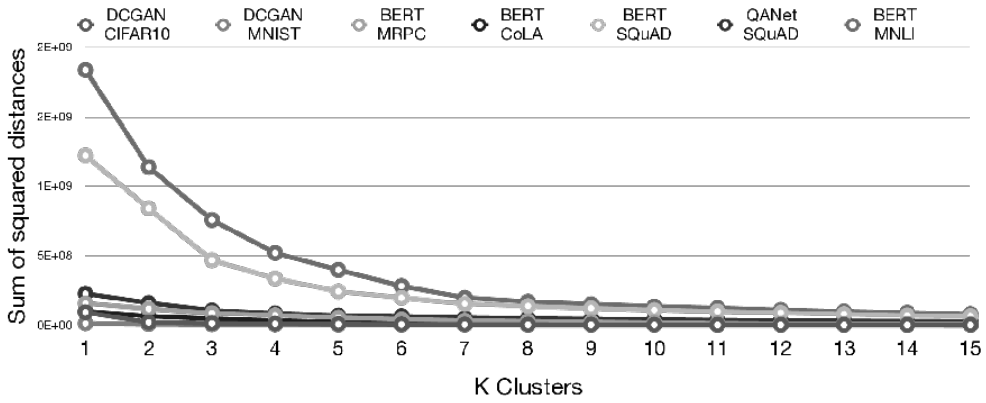

Figure 2.4: Clustering results for TPUPoint-Analyzer with scanning based on k-means with different workloads; the plot shows the sum of squared distances of samples to centroids for k clusters ($k = 1, \ldots, 15$)

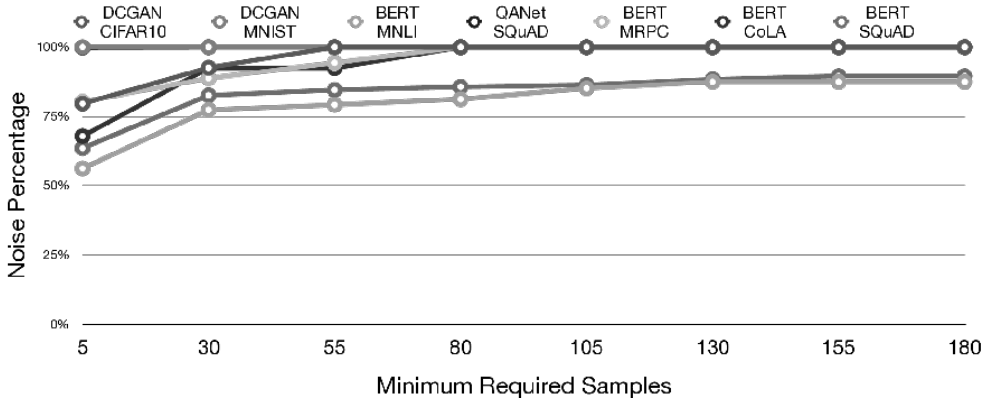

Figure 2.5: Clustering results for TPUPoint-Analyzer using DBSCAN with different workloads; the plot shows the ratios of noisy samples to total samples for 5 to 180 minimum required samples to form clusters in steps of 25

Although DBSCAN and k-means both use the elbow method, DBSCAN does not use centroids, so distance cannot be used as a clustering metric. Instead, DBSCAN varies the number of minimum required samples to form a cluster—designating a sample as either a cluster or a noisy sample. Figure 2.5 shows the ratio of noisy samples to all samples for the minimum number of required samples ranging from 5 to 180 in aggregated steps of 25. The elbow method was applied in attempt to reduce the noise percentage while maximizing the minimum number of samples required to form a cluster. Using DBSCAN,

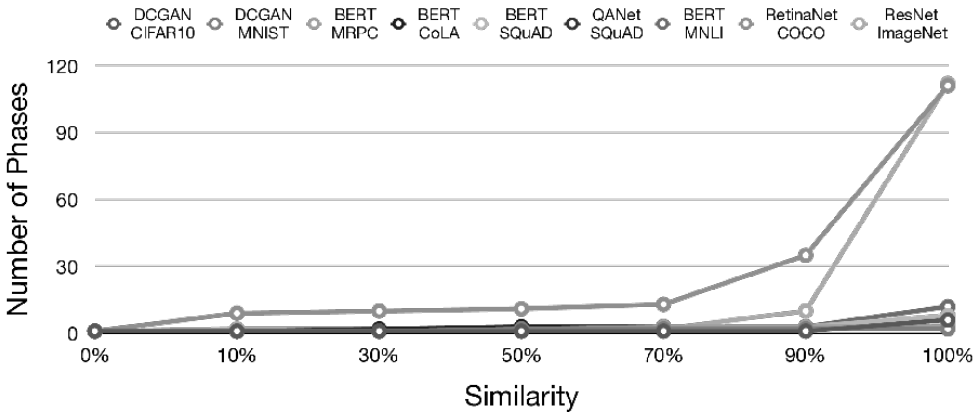

Figure 2.6: TPUPoint-Analyzer using OLS with different workloads; the plot shows the number of phases identified with similarity thresholds from 0% to 100%

TPUPoint-Analyzer found that a minimum of 30 to 80 samples was optimal to reduce noise and produced between 3 to 13 clusters. Again, each cluster represents a phase of more extensive program execution.

Figure 2.6 shows the number of phases that OLS identifies for varying similarities using Equation 2.1. With a similarity threshold of 70%, we found that most workloads are condensed into just 3 phases. For a similarity threshold above 70%, the number of phases identified grows significantly for the majority of the workloads. For these workloads, we further examined the operators within neighboring phases that cannot combine together, and we found the differences between neighboring phases are essentially ignorable, as they often represent a small amount of the application's execution time, turning even single operations into a phase—this creates a low similarity between phases and so creates an excessive number phases.

As OLS tends to break up steps with small differences into different phases, a high similarity threshold leads to a significant increase in the number of identified phases. That

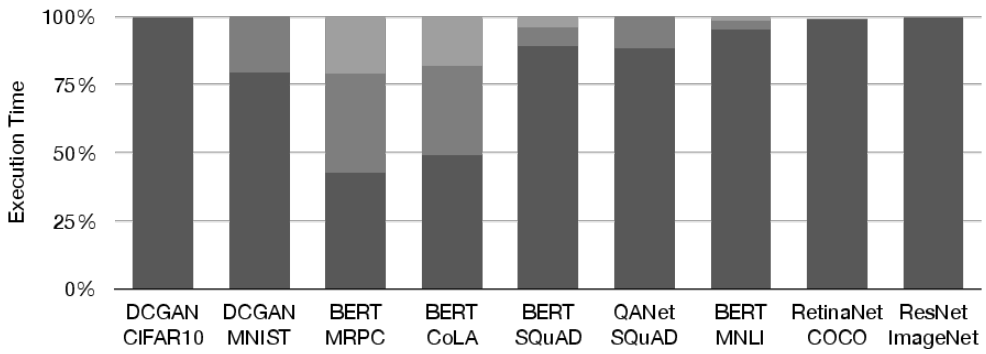

Figure 2.7: Coverage of total execution time by the top three phases from TPUPoint-Analyzer using OLS at the 70% similarity threshold with different workloads, where each color represents one of the three identified phases

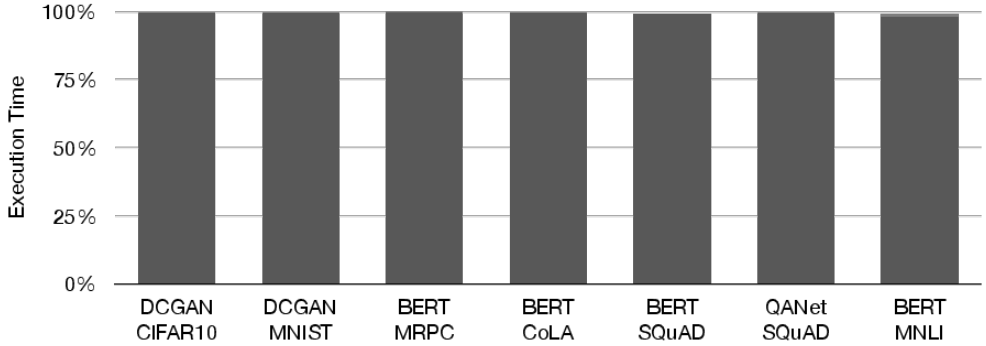

Figure 2.8: Coverage of total execution time by the top three phases from TPUPoint-Analyzer using DBSCAN with minimum samples of 30 to form clusters for different workloads, where each color represents one of the three identified phases

being said, k-means, DBSCAN, and OLS all aggregate the same set of phases into a single phase. Even when TPUPoint-Analyzer uses the extreme 100% *StepSimilarity* threshold (meaning that TPUPoint-Analyzer requires all steps in a phase to share exactly the same breakdown of operators), TPUPoint-Analyzer still breaks up most workloads into fewer than 15 phases, except for the RetinaNet-COCO and ResNet-ImageNet workloads. The above results give us the first observation for this section:

Observation 1: most TPU workloads can be summarized into a limited number of phases.

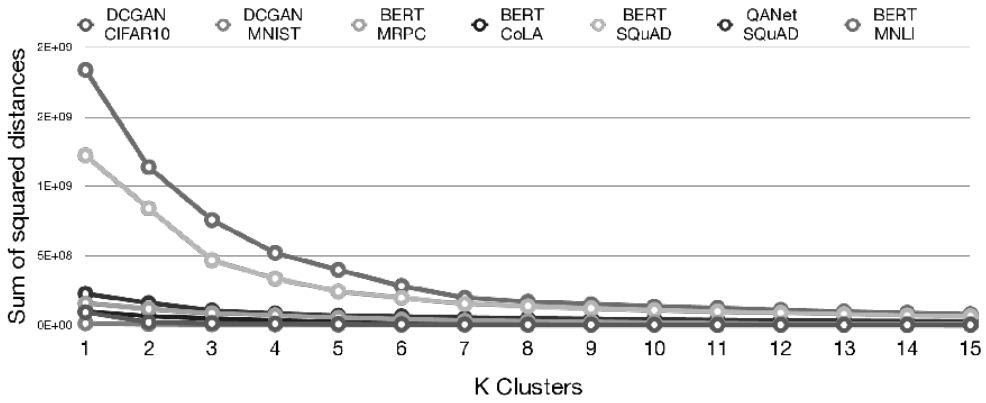

Figure 2.9: Coverage of total execution time by the top three phases from TPUPoint-Analyzer using k-means with $k = 5$ for different workloads, where each color represents one of the three identified phases

Another metric to judge phase selection is the coverage of execution time. Based on observation 1, we accumulated the total execution time of the 3 longest phases for different threshold values. Figure 2.7 shows that these top 3 phases encompass at least 95% of the entire execution of each workload at the 70% similarity threshold when using OLS. For the 70% threshold, TPUPoint-Analyzer can cover almost 100% of execution time for all workloads. The results are similar for k-means ($k = 5$) and DBSCAN (minimum samples = 30), as shown in Figure 2.9 and Figure 2.8, respectively. Because of the high number of noisy sample DBSCAN is unable to cluster, we consider these unlabeled samples to be a cluster as well. We find that these represent a majority of most workload's execution time shown in Figure 2.8. Figure 2.9 demonstrates that even With k-means set to larger than 3 clusters, will still be dominated by the top 3.

Observation 2: the 3 longest phases cover most of the execution time for TPU workloads.

2.6.2 Operators in Phases

Cloud TPUs are simply hardware accelerators in computer systems, so TPU-accelerated workloads still rely on a host program for workload distribution. We now describe the most time-consuming operations on both the host CPU programs and the TPU.

Table 2.2 shows the top 5 most time-consuming operations from the top 3 longest phases on both the CPU/host program and the TPU program using TPUv2. For k-means and DBSCAN, the identified phases are mostly identical with nearly the same set of top operators. For OLS, which tends to divide similar phases into multiple phases, the top 5 operators are slightly different from the top 5 k-means and DBSCAN operators.

Differences notwithstanding, all three algorithms identify a common set of the most time-consuming operators on TPUv2 across workloads, with the `fusion` operator being the most time-consuming overall. The identified `fusion` operator combines compute-intensive operations from the XLA compiler and is intended to help reduce memory operations [187]. The `reshape` operator is also one of the most time-consuming operators. Unlike `fusion`, `reshape` is not algorithm-related, but rather serves only to prepare input data for subsequent TPU computations.

The most critical operators on the host side are `TransferBufferToInfeedLocked` and `OutfeedDequeueTuple`. Both operators exchange data with TPUs. Figure 2.10 shows the percentage of idle time on TPUs for each workload; the Cloud TPUs are, on average, idle for 38.90% of the time for TPUv2 and 43.53% of the time for TPUv3. Figure 2.11

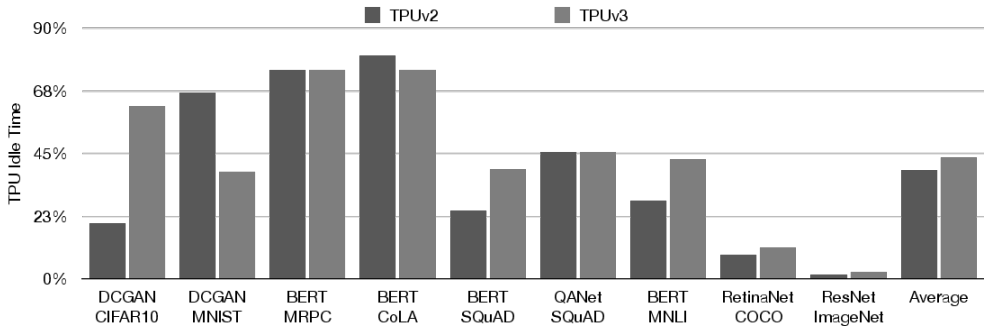

Figure 2.10: Idle time for TPUv2 and TPUv3 across workloads

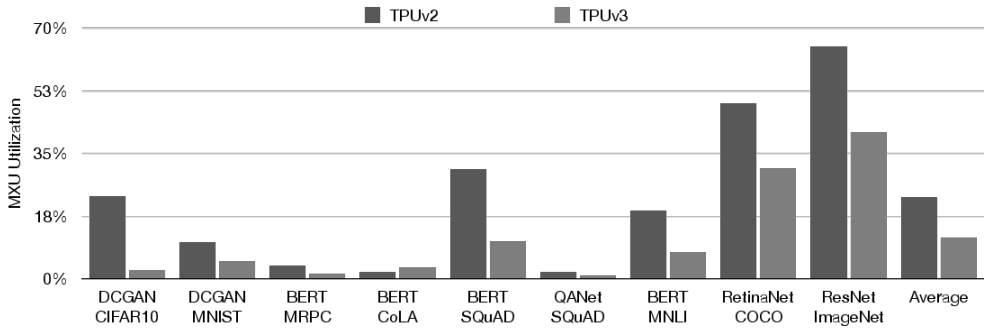

Figure 2.11: MXU utilization for TPUv2 and TPUv3 across workloads

explores the underutilization of the MXUs—on average, from 22.72% for TPUv2 to 11.34% for TPUv3. During idle time, the host is busy preparing and sending data with the top operators listed in Table 2.2. We now have two additional observations:

Observation 3: current TPU workloads incur a significant amount of overhead from data preparation and data exchange.

Observation 4: improving TPU data-preparation and TPU data-exchange efficiency on the host computer is key to improving TPU utilization and TPU workload performance.

To identify the differences between Cloud TPUs, we repeated our analysis with the same workloads, datasets, and parameters with TPUv3. Using OLS, k-Means, and

DBSCAN, we identified the top five operators for the longest identified phase and corresponding cluster. Table 2.2 also shows that the top five operators generally remain consistent for TPUv2 and TPUv3 (as well as the host). Notably, k-Means and DBSCAN reach memory limitations for larger workloads such as RetinaNet and ResNet, which affirms that the TPUPoint-Analyzer/OLS combination can compete with clustering methods implemented in SimPoint [77, 152].

For TPUv3, the most time-consuming operators are the same as those for TPUv2 across workloads, but the total utilization of TPU resources changes. The QANet and RetinaNet workloads reduce flop utilization from about 16% on TPUv2 to 13% on TPUv3 for QANet and from about 46% on TPUv2 to 32% on TPUv3 for RetinaNet. The increased percentage of time required for `infeed` operations indicates the parameters related to memory operators such as `outfeed` need to change to fully utilize TPUv3. However, the observed differences are mainly due to the improved computational capabilities of TPUv2 over TPUv3. The increased percentage of time observed for `infeed` implies that the non-computational overhead in the later-generation TPUs may be more significant.

Observation 5: the significance of non-computational overhead increases as computational throughput improves.

2.6.3 Datasets

For the BERT and DCGAN workloads, we used different datasets to help understand the impact of inputs on the associated models. For BERT workloads with 4 different input datasets, the top 5 operators in Table 2.2, the TPU idle time in Figure 2.10, and the

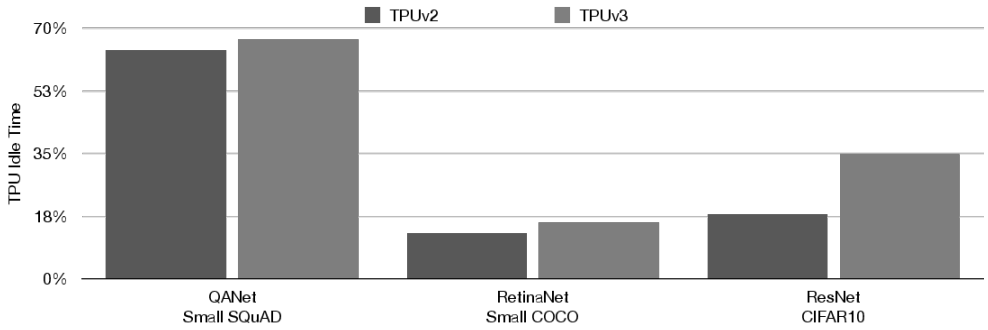

Figure 2.12: Idle time for TPUv2 and TPUv3 across QANet, RetinaNet, and ResNet using smaller datasets

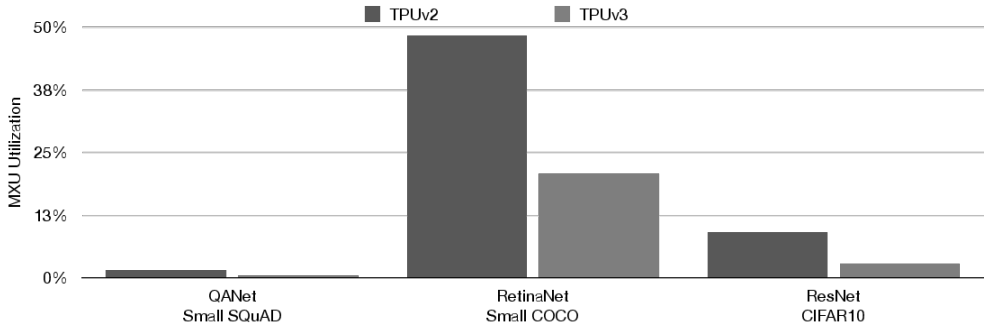

Figure 2.13: MXU utilization for TPUv2 and TPUv3 across QANet, RetinaNet, and ResNet using smaller datasets

MXU utilization in Figure 2.11 are different, just as they are different for the two workloads that use the DCGAN model.

To further observe model behavior across datasets sizes, QANet, RetinaNet, and ResNet were ran with reduced datasets. QANet and RetinaNet were ran by reducing their original SQuAD and COCO datasets in half. ResNet was ran using the CIFAR10 dataset. Figure 2.12 and Figure 2.13 display the idle TPU time and matrix utilization percentage respectively. All models experience a reduction in MXU utilization, and an increase in idle time percentage overall. ResNet in particular experiences the greatest change from it's original ImageNet dataset observations in Figure 2.10 and Figure 2.11 even though using the same methodology to feed in the CIFAR10 dataset. These observations

provide another insight into performance tuning for ML applications: *Observation 6: the performance bottleneck can change as the input dataset changes, even with the same model.* Observation 6 implies that if a programmer optimizes a program with a specific model using a certain dataset, that optimization may not carry over to different datasets. Observation 6 thus points to the need for dynamic runtime optimization to achieve the best performance for ML workloads.

2.7 TPUPoint-Optimizer

Based on the observations from TPUPoint-Profiler, we designed TPUPoint-Optimizer, an automatic tool that helps to fine-tune the performance of an identified phase in a workload. TPUPoint-Optimizer works without programmer input and *ensures that tuning does not affect program-execution output.* TPUPoint-Optimizer does the following to help optimize a workload: (1) It analyzes code and automatically instruments code to assist optimization. (2) It allows for online tuning without the need for complete program execution. (3) It controls the output quality. This section describes the design of TPUPoint-Optimizer.

2.7.1 Program Analysis

If the user enables TPUPoint-Optimizer, TPUPoint-Optimizer will analyze a TensorFlow program between the calls to start and stop TPUPoint-Profiler. During the program-analysis phase, TPUPoint-Optimizer first identifies *adjustable parameters* originally defined by the user. These adjustable parameters include buffer size, the number of threads dedicated to an operation, and the order of operations that can be rearranged while

maintaining correctness. If any of these adjustable parameters cause errors when altered, TPUPoint-Optimizer will not treat them as adjustable. Using the list of input/output variables and adjustable parameters, TPUPoint-Optimizer instruments code to produce checkpoints before each function call within the profiled program.

2.7.2 Online Tuning

TPUPoint-Optimizer provides an online performance-tuning feature that adjusts the performance of TPU workloads without requiring the program to finish a complete execution cycle. The design of TPUPoint-Optimizer's online tuning algorithm comes primarily from two observations described in the previous section: Observation 1—most TPU workloads can be summarized into a limited number of phases. Observation 2—the 3 longest phases cover most of the execution time for TPU workloads. Taken together, these observations suggest that optimization of a small portion of program execution can have a significant impact on program execution as a whole.

After TPUPoint-Optimizer analyzes input/output variables and instruments code for checkpointing, it will start running the workload using the normal inputs and default parameters. At the same time, TPUPoint-Optimizer tracks the accumulated execution time in different code segments using the statistical model that we developed for TPUPoint-Profiler. If TPUPoint-Profiler observes the most common pattern of operators described in Section 2.6 (e.g., `reshape`, `infeed`, `fusion`, `outfeed`) within the most time-consuming phases, or the current phase accounts for more than half of the aggregated execution time, TPUPoint-Optimizer will designate the current code segment as having already entered the performance-critical phase and will optimize accordingly, maintaining correctness.

If performance improves and output does not change, TPUPoint-Optimizer continues adjusting parameter values in the same direction until an optimal value for that specific parameter is found. If no other neighboring values are better than the default value, TPUPoint-Optimizer will keep the default value. Finally, TPUPoint-Optimizer uses the improved adjusted parameters to complete rest of the program's execution.

2.7.3 Performance of TPUPoint-Optimizer

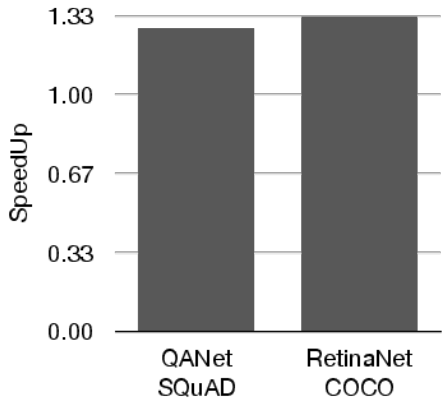

Figure 2.14: TPUPoint-Optimizer speedups for TPUv2

Figure 2.14 shows optimized program performance after using TPUPoint-Optimizer to adjust the default parameters and the execution times on TPUv2 (for naive implementations). Figure 2.14 only shows the workloads that originally took twenty minutes or more to complete—other workloads with much shorter execution times (e.g., DCGAN and BERT) show minimal performance gains from TPUPoint-Optimizer and can actually take a performance hit by waiting for TPUPoint-Optimizer to complete any post processing tasks. Using the default parameters from TPUv2, the workloads with long execution times

achieve a speedup of about 1.12× on average. This indicates that with longer workloads, there is a stronger ability for TPUPoint-Optimizer to isolate parameters that would be beneficial to the execution and speedup of said workloads.

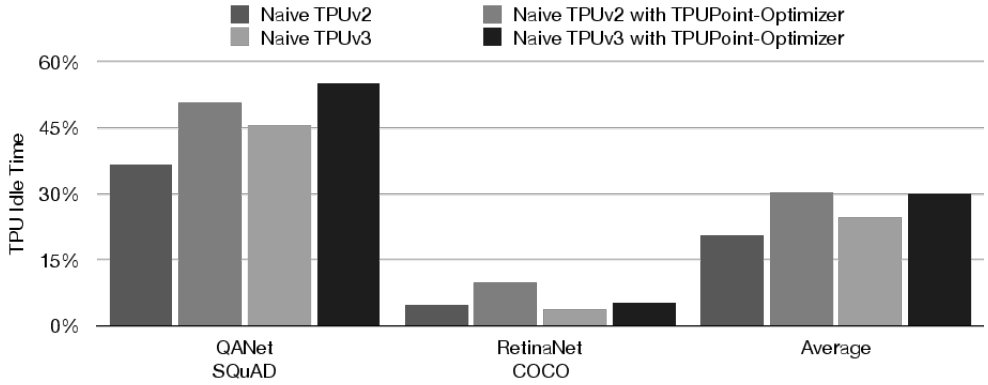

Figure 2.15: Idle time for TPUv2 and TPUv3 across workloads optimized with TPUPoint

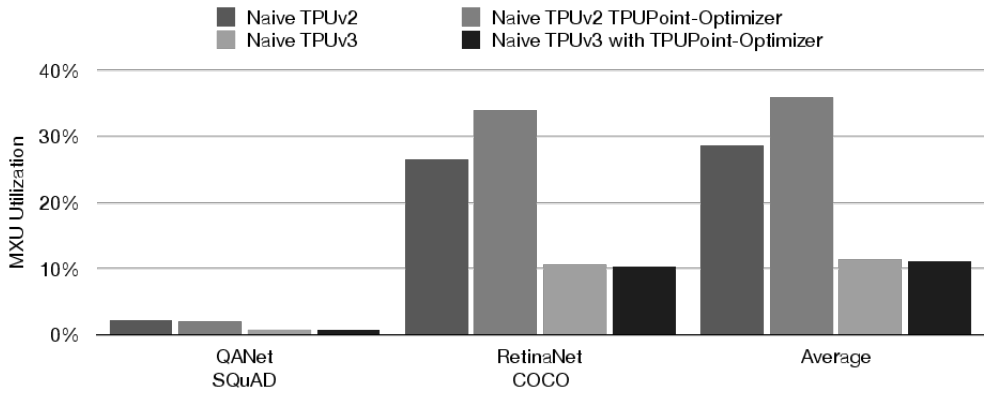

Figure 2.16: MXU utilization for TPUv2 and TPUv3 across workloads optimized with TPUPoint

It's important to note that most of the publicly available ML workloads used in this study were manually optimized by Google engineers. So to test TPUPoint-Optimizer, we developed an original naive implementation to see if TPUPoint-Optimizer could improve poor performance. Figure 2.15 displays TPU idle time of the naive implementation with

and without TPUPoint-Optimizer for both TPUv2 and TPUv3. Figure 2.16 displays the MXU utilization of the naive implementation with and without TPUPoint-Optimizer for both TPUv2 and TPUv3. TPUPoint-Optimizer increased the TPU idle time of the naive implementation for both TPUv2 and TPUv3 (Figure 2.15) and increased MXU utilization for TPUv2 (Figure 2.16). Thus, TPUPoint-Optimizer is able to yield performance gains from more efficient use of Cloud TPUs with TPUv2 exhibiting a pronounced change matrix-operation reliance.

When we applied TPUPoint-Optimizer to our naive workloads that originally had execution times of less than twenty minutes (BERT and DCGAN), the workloads showed no notable change in speed compared to their original performance. In contrast, when we applied TPUPoint-Optimizer to our naive workloads that originally took *more* than twenty minutes (QANet and RetinaNet), we did see improvements in speed—not surprising given that the workloads with longer execution times involve larger and more complex datasets and deeper implementations relative to the workloads with shorter execution times. Because TPUv3 simply contains twice as many MXUs and HBM as TPUv2, we did not observe performance gains from TPUPoint-Optimizer for TPUv3. In fact, we observed an average performance loss under 10% due to the overhead of our profiling/optimization tools. Nonetheless, these results indicate that the overhead associated with TPUPoint-Optimizer is relatively insignificant compared with the overhead associated with complete program execution.

2.8 Related Work

Targeting architectural simulation instead of full-system profiling (the key concept of SimPoint [174, 77, 152]) and clustering similar program behaviors into program phases (as with HyGCN [212]) inspired the development of TPUPoint. TPUPoint also incorporates the checkpointing and restarting features of TurboSMARTS [204] to save time when undertaking architectural simulation and to reduce the cost of cloud computing.

Both TPUPoint and ParaDnn [202] offer tools and systematic methodologies to analyze Cloud TPU performance. TPUPoint provides direct feedback to programmers while automatically and implicitly rewriting under-performing code. In contrast, ParaDnn focuses on systematic testing and optimization insight on architectural perspectives and is therefore complementary to TPUPoint.

In addition to using Cloud TPUs, data centers have often relied on heterogeneous hardware components to accelerate ML workloads [62, 40, 56, 80, 209, 129, 54, 218, 34, 194, 173, 216, 180, 25, 137, 162, 32, 33]. However, hardware solutions are generally not distribution friendly. As TPUPoint works at the programming-language/application level to observe and optimize performance, TPUPoint is portable; simply changing the low-level library function calls that TPUPoint uses to retrieve statistics makes TPUPoint's profiling and optimization available on a wide variety of platforms.

Some benchmark suites also attempt to standardize ML workload management: μ Suite [179], BigDataBench [199], AI Benchmark [92], EEMBC MLMark Benchmark [190, 189], Fathom [3], AI Matrix [6], DeepBench [18], DAWNBench [42], and MLPerf [133], and mixed-precision benchmarks as well [124, 213]. When benchmarking Cloud TPUs, we

can only test a subset of each benchmark suite due to the limited front-end programming-language support for the Cloud TPU platform. That being said, many benchmarks rely on the same models and datasets, varying only frameworks and implementations. We have tried our best to cover the spectrum of ML workloads. There have been several prior works on summarizing ML such as EcoRNN [221], SeqPoints [149], and TBD [222]. These works do not attempt to profile/optimize the same range of benchmarks as TPUPoint does, where computation could be input independent or heterogeneous across iterations. EcoRNN and TBD take a sampling and iteration-based approach to LSTM RNN and DNN respectively, while SeqPoint considers how input variation effects sequence-based neural networks (SQNNs). To provide insight to such a wide range of ML workloads, TPUPoint aims for high coverage but low overhead regardless of the ML workload.

As ML workloads predominate in cloud services, methods for optimizing resource utilization have received significant attention. Some methods use performance estimation algorithms [219, 78, 101, 36, 178] or training models [198, 22, 50] to select optimal parameters. Such methods tend to have stagnant selectors, and while they offer lower overhead, they are limited in their ability to adapt to new workloads. Instead of focusing only on a specific workload, TPUPoint provides a more generic framework applicable to a much broader range of ML tasks.

2.9 Conclusion

This section presents TPUPoint, a toolchain that collects, analyzes, and optimizes the performance of TPU-accelerated ML workloads. Using the post-analysis tool,

TPUPoint-Analyzer, we determined that most TPU-accelerated ML workloads are under-utilizing precious TPU resources. Moreover, because workload behavior varies by model and dataset, manually optimizing workloads is not feasible. Fortunately, the behavior within a workload is often repetitive, opening the door for dynamic optimizations. Using the observations learned from TPUPoint-Analyzer, we designed TPUPoint-Optimizer to detect the main application phase and dynamically adjust parameters in running code. Our results show a 1.12× speedup over default parameters without programmer intervention.

Workload Name	Workload Type	Model	Dataset	Dataset Size	Default Training Parameters
BERT	Natural Language	BERT	Stanford Question Answering Dataset (SQuAD) Microsoft Research Paraphrase Corpus (MRPC) Multi-Genre Natural Language Interface (MNLI) Corpus of Linguistic Acceptability (CoLA)	422.27 MiB 2.85 MiB 430.61 MiB 1.44 MiB	max seq length: 128 train batch size: 32 learning rate: 2e-5 num train epochs: 3
DCGAN	Image Generation	DCGAN	CIFAR10 MNIST	178.87 MiB 56.21 MiB	batch size: 1024 num shards: 8 train steps: 10000 train steps per eval: 1000 iterations per loop: 100 learning rate: 0.0002
QANet	Q/A Natural Language	QANet	Stanford Question Answering Dataset (SQuAD)	422.27 MiB	train batch size: 32 steps per epoch: 20000 num epochs: 5
RetinaNet	Object Detection	RetinaNet	Common Objects in Context (COCO)	48.49 GiB	train batch size: 64 image size: 640 num epochs: 15 num examples per epoch: 120k
ResNet	Image Classification	ResNet-50	ImageNet	143.38 GiB	Default Network Depth: 50 Train Steps: 112590 Default Batch Size: 1024

Table 2.1: Workload breakdown and specifications

		BERT MRPC			BERT SQuAD			BERT CoLA			BERT MNLI			DCGAN CI-FAR10			DCGAN MNIST			QANet SQuAD			RetinaNet COCO			ResNet Ima-geNet			Total TPUv2	Total TPUv3
		OLS	*k*-means	DBSCAN	OLS	*k*-means	DBSCAN	OLS	*k*-means	DBSCAN	OLS	*k*-means	DBSCAN	OLS	*k*-means	DBSCAN	OLS	*k*-means	DBSCAN	OLS	*k*-means	DBSCAN	OLS	*k*-means	DBSCAN	OLS	*k*-means	DBSCAN		
Host Operations	OutfeedDequeueTuple	○			◇	◇	◇	○	○	○	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇			◇			21	17
	TransferBufferToInfeedLocked	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○			○			◇	◇	◇	◇			◇			19	17
	RunGraph	○	○	○				◇	◇	◇	○			◇	◇	◇	□	◇	◇	○	○	○							15	9
	Send	◇	◇	◇	○			□	□	□	◇	◇	◇				○	○	○										10	9
	Linearizex32	□	◇	◇	◇	◇	◇	□	□	□	◇	◇	◇				◇	□	□										9	15
	LSRAv2	○						○						◇	◇	◇	□	□	□	◇	◇	◇							8	9
	InfeedEnqueueTuple				□	◇	◇	○				◇	◇							◇	◇	◇							8	8
	InitializeHostForDistributedTpu								○	○				◇	◇	◇		○	○										7	3
	Restorev2	□	◇	◇				□	◇	◇																			4	6
	DisconnectHostFromDistributedTPUSystem														○	○		○	○										4	0
	ReadHbm				○	○	○																□						3	1
	Recv										□						○												1	1
	Maximum																						◇						1	1
	Minimum																						○						1	0
	Sub																						◇						1	1
	Cast																									◇			1	1
	DecodeAndCropJpeg																									◇			1	1
	ResizeBicubic																									◇			1	1
	StartProgram				□	□	□							□	□	□	□	□	□	□	□	□							0	12
	BuildPaddedOutput	□	□	□																									0	3
TPU Operations	fusion	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇			◇			23	23
	MatMul	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇				◇	◇	◇										15	15
	Reshape	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇				□	□	□	◇	◇	◇							15	18
	L2Loss	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇																12	12
	Conv2DBackpropFilter													◇	◇	◇	○	○	○				○			◇			8	4
	Mul	◇	◇					◇	◇		◇	◇																	6	6
	Transpose			◇	◇	◇	◇			◇			◇																6	6
	BiasAddGrad													◇	◇	◇	◇	◇	◇										6	6
	Conv2DBackpropInput													◇	◇	◇	○	○	○										6	3
	FusedBatchNormV3													◇	◇	◇							◇			◇			5	5
	Infeed																□	□	□	◇	◇	◇							3	6
	all-reduce																			◇	◇	◇							3	3
	Sum																			◇	◇	◇							3	3
	Copy																						◇						1	1
	InfeedDequeueTuple																						◇						1	1
	FusedBatchNormGradV3																						□			◇			1	2
	Relu																									◇			1	1

Table 2.2: The top 5 most time-consuming operators in the most time-consuming phase using different phase-detection algorithms with TPUv2, shown with ○, TPUv3, shown with □, and ◇ for both.

Chapter 3

T2SP: Embedding a DSL in SYCL

The wide spread of tensor computations throughout may domain specific areas has led to a plethora of novel algorithms/workloads. On account of these unique workloads, many unique architectures have been developed to optimize, and exploit parallelism found within tensor computations, utilizing FPGAs, GPUs, TPUs, ASICs, and so on. However, many of these optimizations are restrictive to their unique platforms. Often creating frameworks and libraries to support these optimized implementations, creating platform dependent code. Attempts to run on novel platforms requires new implementations to showcase the platform's individual characteristics. Preventing portability and increasing the cost of development.

In order to gain speedup from tensor computation's parallelism, many of these platforms focus on optimizing latency, throughput, or a combination of the two. This project focuses on combining data parallelism provided though Data Parallel C++ (DPC++), industry-driven standard that adds data parallelism to C++ for heterogeneous systems,

and T2X. T2X is a programming model which realizes parallelism of tensor computations though the efficient use of tensor primitives and systolic arrays. Thought the employment of both these tools, users are able to create a platform agnostic implementation of novel algorithms, without the restriction of architecture dependent frameworks/libraries. Providing portability, efficient utilization of hardware resources, and ease of development into more domain specific areas in computing.

3.1 Introduction

Within the last few deceases, tensor computations have grown to affect many domains including scientific computation, engineering, machine learning, and many other sub-domains [9, 17]. Although all these domains can utilize tensor computation, each individual application can vary in their overall implementation. This creates a diverse and unique range of application, which may or may not preform reasonably on conventional hardware.

In response to the demand in tensor computations, comes the proliferation of new hardware platforms to perform more efficient execution. From TPUs [98], GPUs [130], FPGAs [181], and NPUs [63, 41]. Many of these architectures utilize matrix-vector units, systolic arrays [108], or some novel implementation to exploit the parallelism found within tensor computations.

To support many of these new and developing hardware architectures, industry and academia alike has produces libraries, compilers, and frameworks to exploit hardware resources [1, 76, 181, 155, 95, 31]. This involves very close understanding of individual

hardware characteristics, their downfalls, and well-engineered software stack to navigate them effectively. As such, trade-offs between hardware/software implementations must be made by researchers and developers alike [105, 158, 112, 48].

Creating an inflexible ecosystem and becomes a hurdle for the development of new algorithms that utilize tensor computations [168, 48, 20, 217].

Creating an inflexible software ecosystem becomes a hurdle for the development of new algorithms that utilize tensor computation. This project focuses on combining data parallelism provided though the combination of Data Parallel C++ (DPC++), and Temporal To Spatial Programming (T2SP) to provide a hardware agnostic programming model to construct new tensor computations. DPC++ is an open source compiler project that is based on SYCL, an industry-driven Khronos standard adding data parallelism to C++ for heterogeneous systems. T2SP is both a novel programming framework and compiler which helps enables tensor computations.

DPC++ is an open source compiler project that is based on SYCL, an industry-driven Khronos standard adding data parallelism to C++ for heterogeneous systems. T2SP is both a novel programming framework and compiler which helps enables tensor computation for both spatial and vector architectures such as CPUs/GPUs and FPGAs respectively. T2SP does this by dissolving the marriage between functional specification from spatial mapping. T2SP is based on several observations, notably that spatial architectures favor optimized dataflow and partitioning the computation into many sub-computations distributed over spatial architecture. T2SP allows programmers to describe the computation separately from spatial mapping, partitioning, and dataflow of a spatial architecture.

Allowing programmers to quickly develop various spatial optimizations without having to reconstruct an applications core functional implementation between architectures such as CPUs, GPUs, and FPGAs.

Through the employment of both DPC++ and T2SP, users are able to create a platform agnostic implementation of novel algorithms, without the restriction of architecture dependent software. Providing portability, efficient utilization of hardware resources, and ease of development for tensor applications. Initial evaluations were preformed using General Matrix Multiply (GEMM), 2 Dimensional Convolution (CONV), and Capsule Convolution (CAPSULE), for an Arria-10 FPGA on Intel's FPGA DevCloud Platform. Results show that this project has been able to achieve an average of over 302.402, 285.301, and 231.877 GFLOPs for GEMM, CONV, and CAPSULE respectively. Achieve an average of over 60% of the original T2SP's performance. With minor adjustments, it can be said with confidence that this combination of DPC++ and T2SP can provide competitive performance of tensor applications between special and tensor architectures without extra effort on the end user.

3.2 Overview of T2SP Framework

This section discusses the T2SP programming and compiling overview. Section 3.2.1 delivers a brief overview of the building framework, Temporal to Spatial (T2S), that T2SP is built upon. Section 3.2.2 Discusses the design behind the T2SP and it's compilation onto FPGA. Section 3.3 then goes onto the programming of T2SP specifications and FPGA architecture.

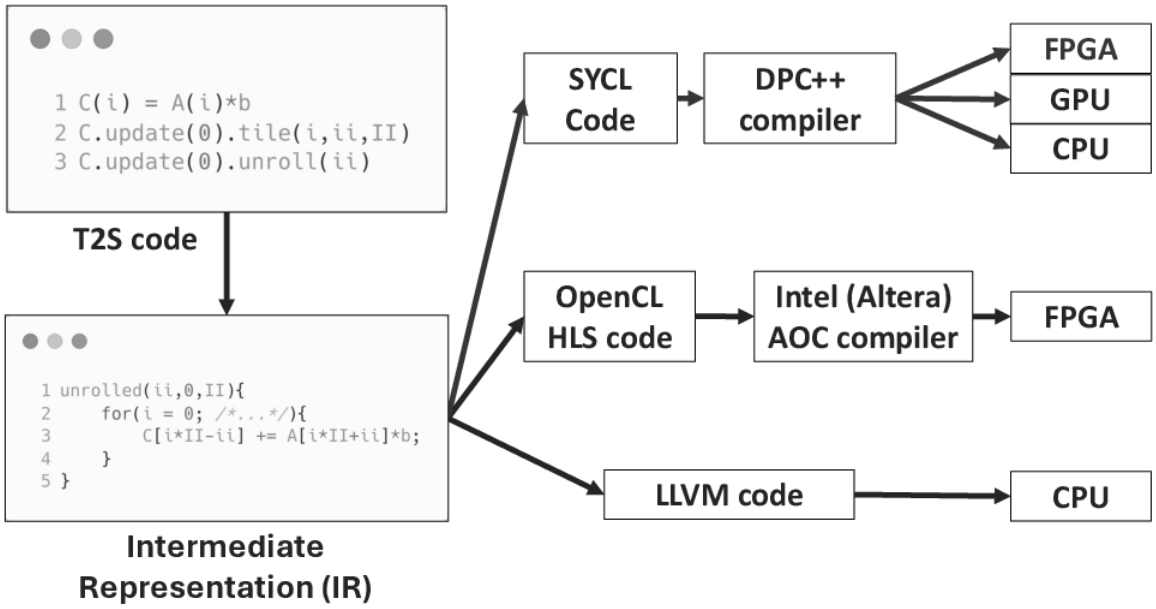

Figure 3.1: T2SP and framework on top of T2S

3.2.1 T2S Framework

The T2S framework [181] provides a language and compiler for efficiently generating high-performance systolic arrays for dense tensor kernels on spatial architectures, such as FPGAs and CGRAs. This was done by decoupling functional specifications from spatial mappings. T2S allows programmers to explore various spatial optimizations without manually implementing them, significantly enhancing productivity and performance. T2S was able to implement several important dense tensor kernels, including GEMM, MTTKRP, TTM, and TTMc, achieving up to 92% of the performance of manually optimized implementations. The framework's efficacy was demonstrated on an Arria-10 FPGA and a research CGRA. Figure 3.1 displays the construction of T2SP on top of the T2S framework. Originally, T2S utilized OpenCL and LLVM code to compile to FPGA and CPU

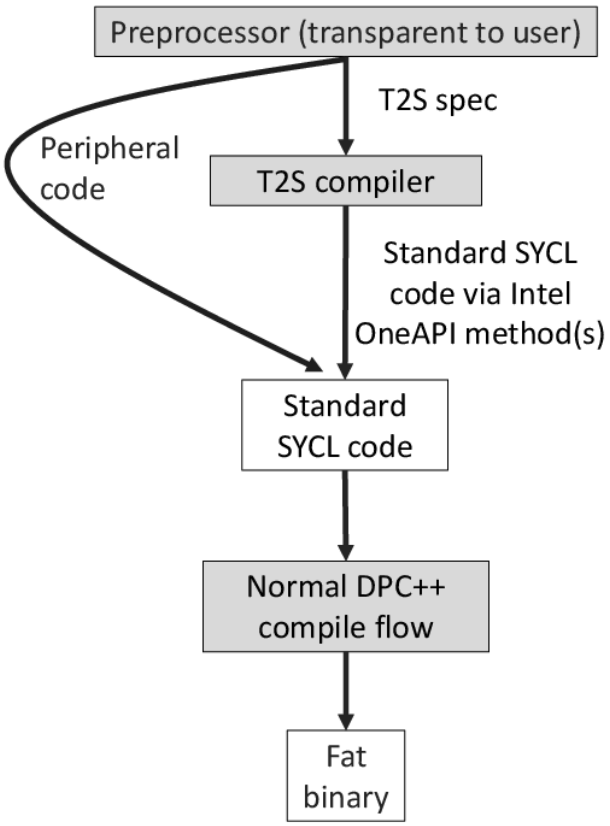

Figure 3.2: T2SP Compiling and Code Translation onto FPGA

respectively. However, through T2SP and it's extension into SYCL and the DPC++ compiler, the T2S framework now has the support needed to compile to devices such as FPGA, GPU, and CPUs without having to choose between compiling frameworks. This work only extended onto FPGA devices as a proof of concept that the union into SYCL was possible and enables performance retention.

3.2.2 T2SP Design

Figure 3.2 Demonstrates the flow a programer uses to create a FAT binary to compile onto FPGA. To begin, a developer will create T2S specifications for the target

hardware, such as defining a systolic array to execute GEMM on an FPGA. This is done using T2SP pragma specification directives to outline the desired hardware-specific operations. This allows the preprocessor to identify the T2S specs and pass them along to the T2S compiler.

Once the specifications are set, developers need to allocate and define argument parameters on the host side to ensure proper data handling and communication. This is seen as peripheral code and is left untouched. Once T2S specifications are passed along through the T2S compiler, both host and desired hardware-specific operations are in standard SYCL code. The developer can now proceed with standard DPC++ OneAPI compiling to produce an executable FAT binary and carried out onto the FPGA.

3.3 T2SP Programming

This section provides an overview of the T2SP Programming flow.

3.3.1 T2S Original Code

Figure 3.3 displays original T2S code. Here, we have an example of a simplified GEMM implementation that a developer would create for the FPGA. The developer will create T2S specification, such as for a systolic array to execute GEMM on FPGA, outlined via the `t2s_spec_start` and `t2s_spec_end` pragma directives. A key part of the specifications is the`compile_to_oneapi()` method within the T2S language. The developers will also allocate and define argument parameters on the host side. Lastly, the developer will invoke the matrix multiple via the `t2s_submit` pragma directive.

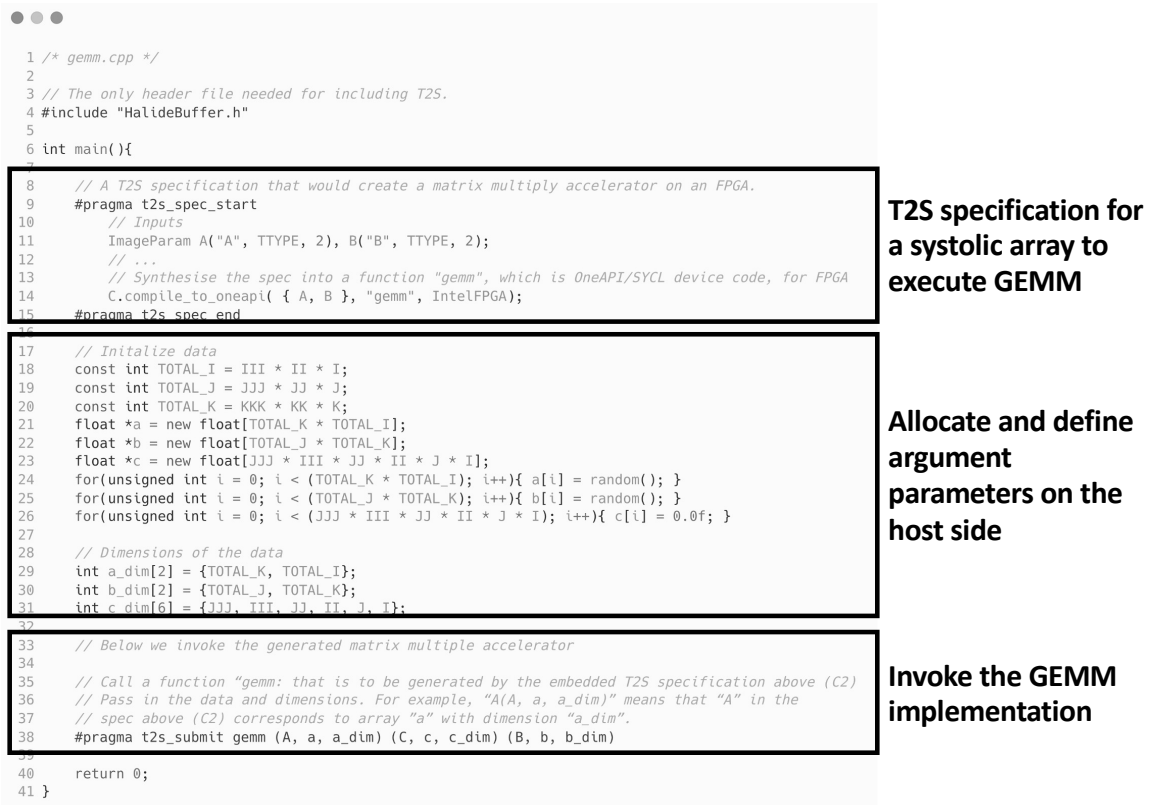

Figure 3.3: Inital T2S Code.

3.3.2 T2SP Generated Code

Figure 3.4 displays the generated T2SP code. T2SP is a combination of the OneAPI/SYCL Code Generator within T2S, exposed by the `compile_to_oneapi` method. The function takes several arguments including: a set of input arguments (`const std vector<Argument> &`, a string that will be the name of device function as well as the generated file name (`const std::string & fn_name`), and lastly a set of T2S targets (i.e. `Target::IntelFPGA` or `Target::IntelGPU`). This is passed into the preprocessor implemented with CLang and a new file is created with the `run.cpp` suffix. The generated code is lengthy, but includes:


```
1  /* gemm.run.cpp */
2  #include "HalideBuffer.h"
3  #include "gemm.sycl.h"
4
5  // Loop bounds
6  #define KKK         16
7  #define JJJ         8
8  #define III         10
9  #define JJ          32
10 #define II          32
11 #define KK          32
12 #define K           32
13 #define J           32
14 #define I           32
15
16 int main(){
17
18     // Initalize data
19     const int TOTAL_I = III * II * I;
20     const int TOTAL_J = JJJ * JJ * J;
21     const int TOTAL_K = KKK * KK * K;
22     float *a = new float[TOTAL_K * TOTAL_I];
23     float *b = new float[TOTAL_J * TOTAL_K];
24     float *c = new float[JJJ * III * JJ * II * J * I];
25     for(unsigned int i = 0; i < (TOTAL_K * TOTAL_I); i++){ a[i] = random(); }
26     for(unsigned int i = 0; i < (TOTAL_J * TOTAL_K); i++){ b[i] = random(); }
27     for(unsigned int i = 0; i < (JJJ * III * JJ * II * J * I); i++){ c[i] = 0.0f; }
28
29     // Dimensions of the data
30     int a_dim[2] = {TOTAL_K, TOTAL_I};
31     int b_dim[2] = {TOTAL_J, TOTAL_K};
32     int c_dim[6] = {JJJ, III, JJ, II, J, I};
33
34     // Call a function "gemm: that is to be generated by the embedded T2S specification above (C2)
35     // Pass in the data and dimensions. For example, "A(A, a, a_dim)" means that "A" in the
36     // spec above (C2) corresponds to array "a" with dimension "a_dim".
37
38     // Preprocessor has wrapped input/output memory in DSL usable Runtime Buffer
39     Halide::Runtime::Buffer<float, (sizeof(a_dim)/sizeof(int))> A_h(
40         a, std::vector<int>(std::begin(a_dim), std::end(a_dim)));
41     Halide::Runtime::Buffer<float, (sizeof(c_dim)/sizeof(int))> C_h(
42         c, std::vector<int>(std::begin(c_dim), std::end(c_dim)));
43     Halide::Runtime::Buffer<float, (sizeof(b_dim)/sizeof(int))> B_h(
44         b, std::vector<int>(std::begin(b_dim), std::end(b_dim)));
45
46     // Preprocessor has created OneAPI/SYCL device selector for FPGA
47 #if defined(FPGA_EMULATOR)
48     std::cout << "USING FPGA EMULATOR\n";
49     sycl::ext::intel::fpga_emulator_selector device_selector; // (NOTE) for emulation
50 #else
51     std::cout << "USING REAL FPGA\n";
52     sycl::ext::intel::fpga_selector device_selector; // (NOTE) for full compile and hardware profiling
53 #endif
54
55     // Below we invoke the generated matrix multiple accelerator
56
57     // Preprocessor has invoded the "gemm" function defined by the developers T2S spec
58     // the generated "gemm.sycl.h" file
59     std::cout << "Start Run\n";
60     double exec_time = 0;
61     exec_time = gemm(device_selector, A_h.raw_buffer(), B_h.raw_buffer(), C_h.raw_buffer());
62     std::cout << "Run completed!\n";
63     std::cout << "kernel exec time: " << exec_time << "\n";
64
65     return 0;
66 }
```

Figure 3.4: Generated T2SP Code.

- SYCL generated header file which defines the FPGA device kernels.
- SYCL memory management with domain specific class structures, necessary to interact with the generated device kernels. Examples include:
  - `sycl::malloc_device()`
  - `std::malloc()`

- SYCL device selector, used by SYCL queues within the device kernels as well.
- Generated implemented the execution of the generated GEMM Function along with printing out device kernel execution time for the developer.

3.4 Experimental Methodology

T2SP was carried out on the Intel DevCloud using the Arria A10 FPGA. Three benchmark matrix kernels were tested; GEMM, CONV, CAPSULE. Experiments were compared to it's predecessor T2S.

At this time, we only have GEMM for T2S, We evaluate the utilization of the Arria A10 hardware for all three kernels. And use GEMM as a comparison for Throughput and clock frequency.

3.5 Results

In this section, we describe our findings of the T2SP. We find that T2SP is able to deliver non-trivial (above 50% performance) in throughput for GEMM compared to it's predecessor, T2S.

3.5.1 Hardware Utilization

Table 3.1 displays the synthesised hardware utilization of GEMM, CONV, and CAPSULE on the Aria A10 FPGA. Our belief that this initial utilization of the hardware has been altered from what is expected to be produced from the original T2S. Through

Benchmark	# ALUT	# Registers	# DSP Blocks	# RAM Blocks	Clock Freq. (MHz)
GEMM	105,151	180,874	89 / 1,518 (6%)	462 / 2,713 (17%)	264
CONV	98,648	168,673	64 / 1,518 (4%)	499 / 2,713 (18%)	276
CAPSULE	119,038	198,909	112 / 1,518 (7%)	607 / 2,713 (22%)	275

Table 3.1: T2SP FPGA Synthesis Results

SYCL, higher clock frequency is achieved, 271.66 MHz on average. This could be due to several sources including alternative memory management from OpenCL which is found within T2S. More investigation is required at this time to establish the root cause.

3.5.2 Performance

Implementation	Throughput (GFLOPs)	Clock Freq. (MHz)
T2S	549	215
T2SP	302	264

Table 3.2: GEMM FPGA Synthesis Results for T2S and T2SP

Table 3.2 displays the compared throughput and clock frequency achieved by T2S and T2SP. We find that T2SP is able to achieve about 55% of the original GFLOPs performed by T2S. Only the GEMM kernel has been evaluated at this time. One reason, we suspect, that we do not achieve higher throughput is due to Halide. A particular reason T2S is able to achieve the throughput it does is due to its reliance on the Halide memory optimization and management system. SYCL may obfuscate or neutralize some of these optimizations, encumbering T2SP from higher performance throughput. However, achieving such roughly 1.22× clock frequency compared to T2S.

3.6 Related Work

Embedding DSLs in various methods has been advanced through innovative tools and techniques. Yogo [157], a semantic code search tool, leverages dataflow graphs and rewrite rules to recognize operations, accommodating variations in APIs, temporary variables, and interleaved code structures. On another front, HAZE [43] introduces a gray-box program synthesis tool focused on dynamic observations such as execution time and memory access patterns to guide program synthesis and lifting. Lastly, GraphCode2Vec [125] employs a Graph Neural Networks approach to generate task-agnostic code embeddings. All of which have the same core focus of maintaining or improving code performance through DSLs on accelerators.

T2SP represents a significant advancement by embedding DSLs within SYCL for productive and performant computing on heterogeneous devices, particularly for its enablement of FPGAs which other works do not. Other works, such as Polly [74], ATC [131], C2TACO [100], KernelFaRer [49], use drastically different methods; LLVM, program synthesis , IO examples and source code analysis, etc.; however none go beyond targeting CPUs or GPUs. This is unique to T2SP as integration with SYCL constructs the groundwork to efficient utilization of GPU resources, but building upon its founding work of T2S [168] which utilizes Halide, T2SP leverages foundational groundwork that includes OpenMP support, enhancing its versatility across different architectures inclusive of CPUs and GPUs.

T2SP represents a novel approach in the realm of heterogeneous computing by embedding a DSLs within SYCL, thereby offering unprecedented flexibility across a wide spectrum of architectures. This capability is particularly groundbreaking as it enables de-

velopers to write performance-critical code and deploy it efficiently across diverse platforms without extensive rewriting.

3.7 Conclusion

T2SP is both a novel programming framework and compiler which helps enables tensor computation for both spatial and vector architectures such as CPUs/GPUs and FPGAs respectively. This work constructs the foundation for the successfully expands Intel's T2SP by integrating SYCL to create a uniform compiling flow for hardware-agnostic acceleration. The development of a code generator for SYCL Device and Host Code, alongside the implementation of a Clang source-to-source preprocessor, has displayed non-trivial performance retention using FPGAs. T2SP has been able to achieve an average of over 302.402, 285.301, and 231.877 GFLOPs for GEMM, CONV, and CAPSULE respectively, achieve an average of over 60% of the original T2SP's performance.

This research not only highlights the potential of SYCL in domain-specific languages but also paves the way for future investigations into the hurdles in the extension of this approach to GPUs, CPUs, and other tensor computing accelerators.

Chapter 4

Accel-Bench: Programming Using Accelerated Functions

Integrating hardware accelerators in modern computers has brought on a new age of programming and application design. Initially, hardware accelerators target a specified domain to reduce computation time and increase energy efficiency or some other metric ideal for a set of applications. However, recent research has shown that hardware accelerators designed for one application domain have the potential to increase performance well outside that domain. Programming using hardware-accelerated functions needs more analysis to quantify this potential across accelerators and applications.

This section presents Accel-Bench, a benchmark suite that aims to capture the performance of accelerator-intensive programming. To the best of our knowledge, Accel-Bench is the first benchmark suite that utilizes applications that can invoke different domain kernels in their algorithm and quantifies the potential performance gain of using hardware-

accelerated functions to compose programs agnostic to their domain. Accel-Bench contains a total of 10 applications currently. In addition to conventional CPU and GPU implementation, applications in Accel-Bench also have another version that invokes popular hardware-accelerated functions, including General Matrix Multiplication (GEMM), Convolution (CONV), or Fast Fourier Transform (FFT). Accel-Bench finds that applications can counter-intuitively obtain similar or even better performance despite increased computational complexity when utilizing hardware accelerators. These applications can also scale up well with the advancement of hardware accelerators. Along with Accel-Bench, this section presents insight into future applications and hardware development in this new age of programming on hardware accelerators.

4.1 Introduction

The introduction of hardware accelerators has brought exotic flavors of computing models into computer systems. Instead of implementing a rich set of fine-grained mathematical or logical operations, each operation in a hardware accelerator can implement a complete compute kernel in the accelerator's target application domain. As each operation covers a coarse-grained computation and each hardware accelerator has a limited target application domain, the design of hardware accelerators can use transistors more efficiently and deliver better performance or energy consumption than general-purpose processors when accomplishing the same task.

From the software design perspective, integrating hardware accelerators and the domain-specific interface these accelerators expose to the rest of the world dramatically

shifted the programming paradigm. Programmers no longer describe the low-level detail of the algorithm using the authoring programming language but only need to invoke a function/method that maps to an algorithm in an application domain that an underlying hardware accelerator implements.

In addition to addressing the demands in accelerators' original target domains, recent research projects have successfully demonstrated the potential of accelerating a broader spectrum of problems using these domain-specific functions. Examples include using AI/ML accelerators for database queries [86, 45, 90], fast Fourier transforms [111], or scientific applications [123, 122, 88, 55, 57], and using ray tracing accelerators for data analytics [223]. As computer systems continue to seek performance gain and energy efficiency through heterogeneous computing and hardware accelerators, computer programming will continue moving toward an accelerator-intensive model. Coped with emerging research outcomes in democratizing hardware accelerators, the future driving force of performance will scale with the relatively faster-growing performance gain on functions that hardware accelerators implement.

However, none of the existing benchmark suites can evaluate the performance growth of leveraging domain-specific accelerators due to the following challenges:

- Conventional applications implement their versions of kernel algorithms. The implementation can only automatically leverage hardware accelerators' functions by significantly revisiting the code.

- As many hardware-accelerated functions are traditionally slower or higher-complexity algorithms (e.g., matrix multiplications, fast Fourier transforms), existing applica-

tions tend to avoid using these functions, eliminating the opportunities for hardware acceleration.

- As the hardware/software interfaces of hardware accelerators are more domain-specific, we need to perform non-trivial code transformations to map an application outside an accelerator's target domain into a problem inside the domain.

This section presents Accel-Bench, a benchmark suite targeting the future world of accelerator-intensive programming. Accel-Bench contains applications from various domains that can leverage the most promising hardware accelerators, including tensor processors, digital signal processors, and ray tracing accelerators. In contrast to conventional programs, Accel-Bench provides alternative implementations that invoke hardware-accelerated functions whenever possible. To increase the opportunities for using hardware accelerators, we carefully re-engineered the algorithms or used different approaches in several application kernels to map their core computation into accelerated, domain-specific functions. In addition to running applications on real hardware, the resulting applications can use simulators like Accel-Sim as long as the framework provides the required hardware-accelerated functions. This section derives the following insights by evaluating Accel-Bench on various platforms. First, the performance of Accel-Bench's implementations scales better than existing state-of-the-art GPU implementations generation-by-generation. Second, though Accel-Bench's implementation sometimes adopts algorithms with higher algorithmic complexities, the actual performance is more competitive due to hardware acceleration. Finally, hardware accelerators offer alternative parallelism that conventional programming models cannot provide to improve performance further.

In summary, this section makes the following contributions.

1. Presents the first benchmark suite that considers the use of hardware accelerators in heterogeneous computers.

2. Presents a set of algorithms that allow applications beyond hardware accelerators' target domains to take advantage of the innovations of hardware accelerators.

3. Presents insights into potential programming paradigms for composing performance code.

4. Presents a benchmark suite that can guide the development of democratized hardware accelerators.

4.2 Motivation

With performance improvements relying more on hardware accelerators, programmers must depend more on functions with hardware implementations to take advantage of accelerators' relatively faster performance scaling when composing applications. As a result, the future high-performance programming paradigm should intensively invoke hardware-accelerated functions instead of encouraging programmers' customized implementation of algorithms.

Accel-Bench fills in the gap of evaluating the benefit of the emerging, hardware-accelerated programming paradigm and adopting new hardware technologies in a broader spectrum of applications. This section will motivate this work and highlight popular, commercialized hardware accelerators.

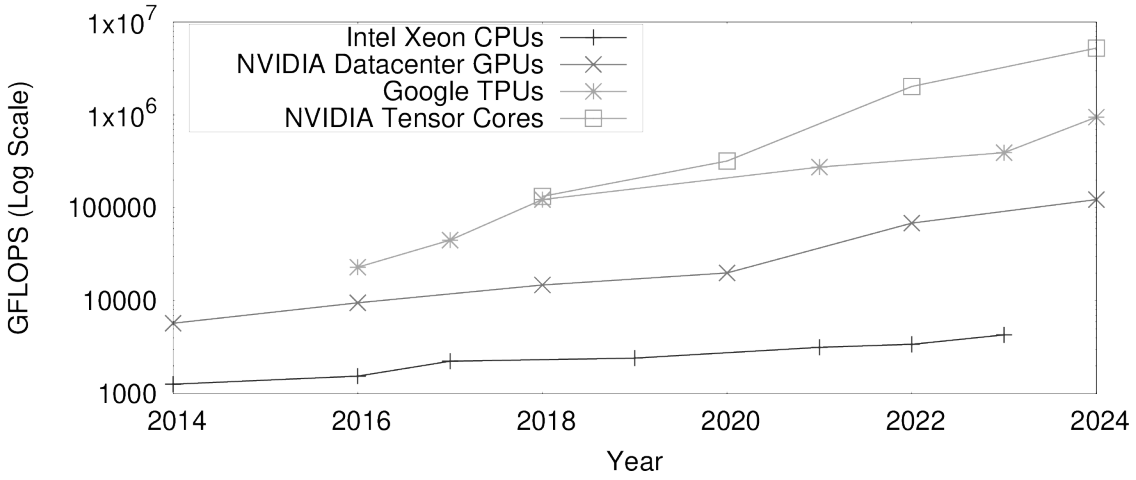

Figure 4.1: The floating point operations per second (FLOPS) of the top-performing CPUs, GPUs, and TPUs between 2017 and 2023

4.2.1 The case of Accel-Bench

Several key technology trends push the development of Accel-Bench. These trends include the relatively faster performance scaling and the popularity of hardware accelerators, the encouraging research outcomes of democratizing hardware accelerators, the shift to domain-specific style programming models, and, most importantly, the absence of a benchmark suite reflecting the paradigm shifts.

The discontinuation of Dennard scaling and the adoption of hardware accelerators

The discontinuation of Dennard scaling increases the power density as the process technology shrinks. The increased power density limits the performance gain of general-purpose processors. Figure 4.1 shows the giga-floating point operations per second (GFLOPS) that best-performing Intel Xeon processors, NVIDIA data center GPUs (conventional vector/CUDA Cores Only), and two representatives of AI/ML accelerators,

Google's data center TPUs and NVIDIA's Tensor Cores, can deliver at the year each generation of the commercialized product rolled out. Despite the lithography improvement from 14 nm to 7nm for Intel CPUs, the FLOPS per core only improves by 3.4×. On the other hand, NVIDIA GPUs' CUDA core performance also receives 21× improvement in the same period as the increasing number of CUDA cores per chip. In the meantime, the FLOPS of the most representative AI/ML accelerator, TPUs reveals 41× improvement. NVIDIA's Tensor Cores also see a very close 39× improvement as TPUs. Using the log-scale representation in Figure 4.1, this figure shows that AL/ML accelerators consistently deliver performance at a higher order of magnitude than GPUs. Based on the trend of faster performance scaling and the current performance of hardware accelerators, we can project the gap between conventional general-purpose scalar/vector cores and AI/ML accelerators to widen.

Broader spectrum of applications

Similar to the technology trend as GPGPUs, we have seen the emerging research outcomes of applying AI/ML accelerators on other application domains. For example, as most accelerators can perform general matrix multiply efficiently, existing projects have demonstrated the use of AI/ML accelerators in other matrix-based applications, including image processing and database queries [87, 45, 90, 123, 122, 88, 111, 55, 57].

```
__global__ void matrixMultiplication(float *A, float *B, float *C,
                                     int numARows, int numAColumns,
                                     int numBRows, int numBColumns,
                                     int numCRows, int numCColumns)
{
    __shared__ float sA[TILE_WIDTH][TILE_WIDTH];   // Tile size of
32x32
    __shared__ float sB[TILE_WIDTH][TILE_WIDTH];

    int Row = blockDim.y * blockIdx.y + threadIdx.y;
    int Col = blockDim.x * blockIdx.x + threadIdx.x;
    float Cvalue = 0.0;
    sA[threadIdx.y][threadIdx.x] = 0.0;
    sB[threadIdx.y][threadIdx.x] = 0.0;

    for (int ph = 0; ph < (((numAColumns - 1) / TILE_WIDTH) + 1);
ph++) {
        if ((Row < numARows) && (threadIdx.x + (ph * TILE_WIDTH)) <
numAColumns) {
            sA[threadIdx.y][threadIdx.x] = A[(Row * numAColumns) +
threadIdx.x + (ph * TILE_WIDTH)];
        } else {
            sA[threadIdx.y][threadIdx.x] = 0.0;
        }
        if (Col < numBColumns && (threadIdx.y + ph * TILE_WIDTH) <
numBRows) {
            sB[threadIdx.y][threadIdx.x] = B[(threadIdx.y + ph *
TILE_WIDTH) * numBColumns + Col];
        } else {
            sB[threadIdx.y][threadIdx.x] = 0.0;
        }
        __syncthreads();

        for (int j = 0; j < TILE_WIDTH; ++j) {
            Cvalue += sA[threadIdx.y][j] * sB[j][threadIdx.x];
        }
    }
    if (Row < numCRows && Col < numCColumns) {
        C[Row * numCColumns + Col] = Cvalue;
    }
}
```

(a)

```
cublasErrCheck(cublasCreate(&cublasHandle));

cublasErrCheck(cublasSetMathMode(cublasHandle,
CUBLAS_TENSOR_OP_MATH));
cublasErrCheck(cublasGemmEx(cublasHandle, CUBLAS_OP_N, CUBLAS_OP_N,
               MATRIX_M, MATRIX_N,
               MATRIX_K, &alpha, a_fp16, CUDA_R_16F, MATRIX_M,
               b_fp16, CUDA_R_16F, MATRIX_K,
               &beta, c_cublas, CUDA_R_32F, MATRIX_M, CUDA_R_32F,
               CUBLAS_GEMM_DEFAULT_TENSOR_OP));
```

(b)

```
x_sqrt = sqrt(x_dim);
y_sqrt = sqrt(y_dim);
for(int i = 0; i < x_dim; i++) {
    for( int j = 0; j < y_dim; y++) {
        A_conv[(i/x_sqrt-1)*x_sqrt+i%x_sqrt][(j/
y_sqrt-1)*y_sqrt+j%y_sqrt] = A[i][j];
}
for(int i = 0; i < x_dim; i++) {
    for( int j = 0; j < y_dim; y++) {
        B_conv[(i/x_sqrt-1)*x_sqrt+i%x_sqrt][(j/
y_sqrt-1)*y_sqrt+j%y_sqrt] = B[i][j];
}

conv2D(C_conv, A_conv, B_conv, x_dim, y_dim, x_sqrt, y_sqrt);

for(int i = 0; i < x_dim; i++) {
    for( int j = 0; j < y_dim; y++) {
        C[i][j] = C_conv[(i/x_sqrt-1)*x_sqrt+i%x_sqrt][(j/
y_sqrt-1)*y_sqrt+j%y_sqrt];
}
```

(c)

Figure 4.2: Programming models using (a) GPGPU (b) Hardware-accelerated functions and (c) Hardware accelerators without natively support of functions

Hardware-accelerated-function-based Programming model

Unlike general-purpose processors, the programming models on hardware accelerators are more domain-specific and API-based. Figure 4.2 illustrates the difference between the emerging and conventional models. In conventional programming models (i.e., Figure 4.2(a)), the programmer typically describes the algorithm and resource allocation at a very fine-grained level. In contrast, the hardware-accelerated function programming model (i.e., Figure 4.2(b)) only allows the programmer to call the API function through parameters without modifying the detail inside the `cublasGemmEx()` function. As a result, when the underlying accelerator lacks support for the desired functionality, the programmer must manually perform some data transformation before calling the hardware-accelerated function to perform the computation. In this work, we name this style of programming paradigm as hardware-accelerated-function-based programming model. Figure 4.2(c) illustrates such an example from a prior work that implements GEMM using the convolution 2D function on Edge TPU [88]. As Edge TPUs focus on inference, Edge TPU only implements matrix-vector and convolution 2D functions. Therefore, the programmer must explicitly compose two additional for-loops before calling the hardware-accelerated `conv2D` to re-layout input data and one for-loop after the function call to present the result as the application desires. However, prior work still shows that a hardware-accelerated-function-based programming model outperforms its counterparts even with such overhead [88].

Type of Accelerators	Commericalized Examples	Accelerated Functions
AI/ML Accelerators	[97] [10] [38, 144] [73, 72] [7]	GEMM/CONV
DSP	[8, 150, 141]	FFT
Ray Tracing Accelerators	[38, 144]	BVH Tree

Table 4.1: Popular accelerators and their accelerated functions

The absence of benchmark suite

To our knowledge, Accel-Bench is the first benchmark suite targeting the hardware-accelerated-function-based programming model. Existing heterogeneous benchmark suites only focus on supporting CPU and GPU implementations but do not provide versions that can leverage hardware accelerators [89, 27, 67]. Benchmark suites like MLPerf [163] can leverage AI/ML accelerators but only offer insights to applications within the AI/ML domain, not others.

4.2.2 Commercialized Hardware Accelerators

Due to the demands of real applications, hardware accelerators have been rapidly adopted and commercialized in modern architectures. The most promising categories of hardware accelerators include AI/ML accelerators, digital signal processors, and ray tracing accelerators. Table 4.1 summarizes these accelerators and their commonly supported functions.

AI/ML accelerators

AI/ML accelerators are ubiquitous in all forms of modern computer systems. On high-performance computers, AI/ML accelerators can exist as standalone co-processors as Google's Tensor Processing Units (TPUs) [97] or part of the GPU cores as NVIDIA's Tensor Cores [38, 144] or AMD's Matrix Cores [7]. AI/ML accelerators can also be part of system-on-chip (SoC) solutions in personal computers or mobile/embedded platforms. Famous examples include Intel's GNA accelerators, Apple's Neural Engines [10], and Google's Edge TPUs [73, 72].

For accelerators targeting the forward pass or inference in AI/ML applications, these accelerators provide functions corresponding to 1D or 2D convolutions or fully connected operations (i.e., matrix-vector multiplications). Accelerators targeting the training process, especially the backward propagation phase, must implement domain-specific functions with matrix multiplications at their cores.

Due to the importance of AI/ML applications and mismatching processor-application demand, this work envisions AI/ML accelerators will last in the foreseeable future and related mathematical functions will continue to be the target of performance improvement. Accel-Bench therefore focuses on using mathematical functions in these accelerators.

Digital Signal Processors and Hardware Codecs

Microprocessors and software implementations once replaced DSPs and hardware codecs. However, we have seen the renaissance of DSPs and hardware codecs in modern

architectures for several reasons. First, as the human-computer interface changes from text-based commands to voice control and image/video sensor inputs, together with the demand for immersive experiences and high-resolution video, the relatively slow improvement of microprocessor performance cannot catch up with the application demand. Second, as popular AI/ML applications rely on image, audio, and video as inputs, data center servers must extract helpful information and features from these inputs at a speed that matches the performance of AI/ML accelerators. Third, as wireless communication becomes the primary data exchange method and the high bandwidth demand, computer systems also need an accelerator to parse signals from the antenna. Finally, even though microprocessors can perform these tasks, they cannot offer the same energy/power efficiency as DSPs and codecs.

Beyond traditional standalone DSPs and codecs [8, 150, 141], they have become a standard integration in modern GPU architectures [38, 144] and SoC solutions. These DSPs and codecs typically offer transformations that convert signals from one space into a numerical system that software can digest, clean up the noises in the signals, or decrypt/encrypt/compress/decompress data.

We found fast Fourier transforms (FFTs) relatively common among supported transformation methods in these accelerators. Therefore, HDBench focuses on using FFTs in our applications.

Ray Tracing Accelerators

As the demand for immersive user experience grows, ray tracing hardware gains ground in modern GPU architectures. For example, NVIDIA has integrated RT Cores since

Volta architecture. Ray tracing algorithm is irregular and more complex to compute than conventional rendering methods that are embarrassingly parallel and highly regular. Current ray tracing hardware accelerates the bounding volume hierarchy (BVH) tree traversal process that tests if a ray intersects with a bounding volume/object. Research projects have demonstrated the use of RT Cores in accelerating nearest neighbor search problems [223] and Monte Carlo simulations [169].

Accel-Bench includes RTNN to compare alongside other implementations of nearest neighbor search. However, as ray tracing hardware is still evolving, we aim to add more applications using ray tracing hardware in the future.

4.3 The Accel-Bench Benchmark Suite

Accel-Bench aims to allow the community to access the potential of programming using hardware-accelerated functions, evaluate performance scaling with emerging hardware accelerators, and assist the architecture design of more general hardware accelerators. Accel-Bench identifies a broad spectrum of applications that cover several vital dwarfs while their algorithmic problems can map to hardware-accelerated functions. Accel-Bench revisited these applications and revised their state-of-the-art implementations to leverage existing/potential hardware-accelerated functions. This section will describe the goals and the detailed implementations of Accel-Bench.

Benchmark	Dwarf	Application Domain	Hardware accelerated function(s)	Baseline Implementation
Canny Edge Detection (CED)	Dense Linear Algebra	Image Processing	CONV	RosettaCode [65], Chai [67]
Fully Homomorphic Encryption (FHEW)	Structured Grid	Security/Encryption	FFT	FHEW [35]
Genomic Relationship Matrix (GRM)	Dense Linear Algebra	Bioinformatics / Genomics	GEMM	GenomicsBench [183]
Heat (Heat2D/Head3D)	Structured Grid	Physics Simulation	CONV/FFT	TEAlab [185], FDTD3D [140]
KMeans (KM)	Dense Linear Algebra	Data Mining	GEMM	Rodinia [27]
K Nearest Neighbor (KNN)	Dense Linear Algebra	Data Mining	GEMM	KNNCUDA [196], RTNN [223]
PageRank (PR)	Graph Traversal	Web Minning	GEMV	GAP [21]
Short-Time Fourier Transform (STFT)	Spectral Methods	Digital Signals Processing	FFT	RosettaCode [60]
Triangle Counting (TC)	Graph Traversal	Social Network Analysis	GEMM	GAP [21]

Table 4.2: Table of benchmarks

4.3.1 Overview

To achieve the goals of designing and evaluating future accelerator-rich computer architectures, Accel-Bench has the following features.

Efficient implementations on various programming models Each application in Accel-Bench offers efficient implementations of various programming paradigms on different computation models to enable the access of the programming paradigms relying on hardware-accelerated functions. Each application contains at least three versions of implementations with (1) a pure CPU-based implementation, (2) a GPU-accelerated implementation, and (3) an implementation using the hardware-accelerated-function-based programming model.

Hardware-accelerable function interface the application code in Accel-Bench calls high-level hardware-accelerable function interfaces to facilitate the evaluation of different types of hardware accelerators. These functions map to popular hardware-accelerated functions (e.g., GEMM, FFT, etc.) that a future general-purpose programming paradigm can potentially implement. In addition to the default implementation of functions we provided, the user can easily customize the back-end implementations without significantly rewriting the code.

Algorithms that use hardware accelerated functions As hardware accelerators implement a coarse-grained algorithm in their hardware design, the hardware-accelerated version of each application in Accel-Bench will contain non-trivial transformations that allow the application to leverage hardware-accelerated functions. These transformations typically include a change in data dimensionalities and types. Some transformations may even use a different algorithm to tackle the same problem.

Supporting architectural simulators Another target of Accel-Bench is encouraging the development of more generic hardware accelerators. Therefore, applications in Accel-Bench support architectural simulators like Accel-Sim [102]. The aforementioned virtual function feature also helps support simulators as the user can call functions that the underlying architectural simulators support as the back-end.

Publicly available source code and datasets Finally, we will open-source Accel-Bench applications [1]. All applications will contain pointers to real datasets or provide generators.

4.3.2 Accel-Bench's Accelerated Library and Application

As Accel-Bench promotes and evaluates the emerging hardware-accelerated-function-based programming model, each application will contain a version besides the state-of-the-art CPU/GPU implementations. Therefore, Accel-Bench provides an acceler-

[1]The current version is anonymized in https://anonymous.4open.science/r/hdbench-B2C9 before this work is published.

```
int accel_gemm(int data_type, void *input_a, void *input_b,
void_*output_c, size_t M, size_t N, size_t K);
```

(a)

```
#ifdef CUDA
// use GPU/CUDA to run GEMM
int accel_gemm(int data_type, void *input_a, void *input_b,
void_*output_c, size_t M, size_t N, size_t K) {
    …… // skipped data preparation code
    ret = cublasHgemm(handle, transa, transb, M, N, K, alpha, A,
lda, B, ldb, beta, C, ldc);
    …… // skipped wrap-up code
    return ret;
}
#elif ETPU
// use ETPU to run GEMM
int accel_gemm(int data_type, void *input_a, void *input_b,
void_*output_c, size_t M, size_t N, size_t K) {
    …… // skipped data preparation code
    float* input = interpreter->typed_input_tensor<float>(0);
    interpreter->Invoke();
    output_c = interpreter->typed_output_tensor<float>(0);
    …… // skipped wrap-up code
    return ret;
}
#else
// plain, not-accelerated GEMM
int accel_gemm(int data_type, void *input_a, void *input_b,
void_*output_c, size_t M, size_t N, size_t K) {
    …… // skipped data preparation code
    …… // skipped wrap-up code
    return ret;
}
#endif
```

(b)

Figure 4.3: The Accel-Bench library's (a) function definition in the header file and (b) hardware-accelerable functions

ated function library that provisions function with the potential mapping to existing/future hardware accelerators. Accel-Bench's unique version of the benchmark implementation leverages these functions as much as possible.

Each Accel-Bench's hardware accelerated version of the benchmark will include the header file of HDBench's library. The current implementation of the library consists of frequently used functions, including convolution (Conv2D/Conv3D), general matrix multiply (GEMM), discrete Fourier transforms (FFT), and matrix-vector/fully connected (GEMV).

Figure 4.3(a) shows an exemplary function declaration of a hardware-accelerable function in the library header of Accel-Bench. Following the interface defined by the function declaration in the header file, the function implementation in Figure 4.3(b) can provide

multiple versions of code, each potentially mapping to a different accelerator. In the example of Figure 4.3(b), the `accel_hgemm()` will use the implementation of Figure 4.2(b) if the user sets the TensorCore flag when making the library or Figure 4.2(c) if the user sets the ETPU flag when making the library. When running an application on real machines, the user can choose the version of the accelerated library to link dynamically.

Figure 4.3(c) shows a hardware-accelerated coding example from genetic relationship matrix (GRM) in Accel-Bench. The core algorithm only invokes `accelbench_gemm()`, the implementation handles the data transformation and internally calls the cuBLAS library that Tensor Cores can accelerate or any other implementation if the user sets the corresponding compilation flag.

4.3.3 Workloads

Table 4.2 lists the applications that Accel-Bench includes. These applications cover dwarfs, including dense linear algebra, Structured Grid, Spectral Methods, and Graph Traversal. These applications fall into image processing, data mining, physics simulations, genomics, signal processing, web mining, and social network analysis beyond the domains of the hardware accelerators that modern computers provide.

However, as hardware accelerators implement complete kernel functions, we must revise these applications' algorithms and perform non-trivial code rewriting to leverage hardware-accelerated kernel functions. The following paragraphs describe our efforts to allow these applications to use hardware accelerators.

Canny Edge Detection (CED)

CED contains a series of processes to detect the edges of images. CED includes Gaussian Filtering, Non-Maximum Suppression, and edge detection with Hysteresis. The Accel-Bench's accelerated version revisited the Convolution for Gaussian Filtering and data-level parallelism with hardware-accelerated convolution functions.

For state-of-the-art GPU version, Accel-Bench includes pure CUDA implementation from Chai benchmark suite [67]. We use RosettaCode [65] as baseline CPU implementation.

Fully Homomorphic Encryption (FHEW)

Fully homomorphic encryption (FHE) is an emerging approach that enables confidential computing using encrypted data on an untrusted third party without decryption. However, as the encrypted noise aggregates after each operation, FHE must perform additional computations to reduce the noise level. The common practice is to apply Gentry's Bootstrapping [64, 53] mechanism. The modern implementation adopts the idea from Chillotti [35] by reducing the complexity of solving the polynomials using Fast Fourier Transform (FFT) or numerical theoretic transform (NTT). Accel-Bench's accelerated version replaces the FFT/NTT code in our baseline with hardware-accelerated FFT. The current version calls functions from the hardware-accelerated cuFFT library. The CPU baseline comes from the GAP Benchmark suite [21] as a baseline. For the GPU, we utilized cuBLAS [142] with and without Tensor Cores to accelerate matrix multiplication, along with CUDA kernels, to calculate the Hamming product and summarize the final count.

Genomic Relationship Matrix (GRM)

GRM is an important application in the genetic analysis of human traits. Equation 4.1 computes the average genetic similarity between individuals for each element of the GRM.

$$GRM_{ij} = \frac{1}{S} * \sum_{s=1}^{S} \frac{(x_{is} - 2p_s)(x_{js} - 2p_s)}{2p_s(1 - p_s)} \tag{4.1}$$

In Equation 4.1, x_{is} and x_{js} indicate the number of copies of the non-reference base at location s for individuals i and j, and p_s is the expected frequency of the non-reference base at location s in the population. The genome contains a total of S SNV (Single Nucleotide Variation) location markers.

The GRM workload starts with initializing a standardized genome matrix W as shown in Equation 4.2.

$$W_{ij} = \frac{(x_{ij} - 2p_j)}{\sqrt{2p_j(1 - p_j)}} \tag{4.2}$$

Then, GRM averages the multiplication of W with its transposed W^T, as in Equation 4.3.

$$GRM = \frac{(W * W^T)}{S} \tag{4.3}$$

The continuous iterations of Equation 4.1 to Equation 4.3 make matrix multiplication accelerators well-suited option for GRM. However, Accel-Bench has to pre-process to calculate the expected frequencies p_s so enable matrix multiplication in Equation 4.3.

Accel-Bench's CPU baseline comes from Plink2 [26] in Genomics Benchmark suite [183]. There is no state-of-the-art CUDA/GPU implementation of GRM, so we degrade Accel-Bench's implementation to use non-tensor-core accelerated cuBLAS function (but still enjoys highly optimized CUDA core acceleration) as the pure GPU baseline.

Heat (Heat2D/Heat3D)

Accel-Bench includes two benchmark applications in simulating heat dissipation in 2D and 3D structured chips/materials. The benchmark has wide applications in many engineering fields, such as fluid dynamics [23, 61, 84], electromagnetism [12, 107, 192], mechanical engineering [161, 14, 184], meteorology [15, 99, 165, 166], cellular automata [134, 139, 175, 176], and image processing [154, 195, 203], use stencil computations.

The simulation performs stencil operations on 2D or 3D neighbors, depending on the dimensionality of the design. Each stencil operation changes an element's temperature by multiplying a set of coefficients on its neighbors and itself. One of Accel-Bench's acceleration is mapping the stencil operation to convolutions. As modern AI/ML accelerators offer hardware-accelerated convolutions for convolutional neural networks and image processing, mapping stencils to convolutions will enable the use of these functions (e.g., tensor core accelerated convolution in cuDNN [145]). However, the trade-off is the waste of multiplications on zeros for coefficients mapping to non-immediate neighbors in the convolution kernel. Accel-Bench also offers another accelerated implementation using Fast Fourier Transform (FFT) based on the recent research projects [4]. The FFT version of H2D/H3D uses cuFFT [143].

KMeans

KMeans is a well-known and popular clustering algorithm in data-mining with high dimensionality. The input dataset of KMeans contains n records standing for n input data points. Each record contains d different attributes that represent d dimensions in a

high-dimensional space. The application receives k initial query cluster centers, and the goal is to assigns each point to its closest cluster and then updates each of the k cluster centers with the mean of that cluster group. This process repeats for a defined number of iterations iteration and until the result converges.

The main computation in KMeans is to compute the Euclidean distances between a data point and all cluster centers as Equation 4.4.

$$d(u, v) = \sqrt{\sum_{i=0}^{d} (u_i - v_i) z^2} \tag{4.4}$$

The cost is especially high as the data dimensionality goes large. Fortunately, Equation 4.4 can map to matrix multiplications, and since the application is typically error-tolerant. Equation 4.4 becomes a great candidate using matrix multiplication accelerators. AccelBench's implementation recalculating and applying weights to the nearest k to enable the application of GEMM in calculating Equation 4.4.

KMeans is a very popular application in other benchmark suites [27, 89]. Our baseline implementation chooses [27] as the same benchmark suite includes a GPU implementation, and there is no significant performance difference from the other.

K Nearest Neighbor (KNN)

KNN is a common algorithm where KNN returns the k closest neighbor and the k associated Euclidean distances. Researchers and practitioners apply KNN in domains such as 3-dimensional object rendering, content-based image retrieval, statistics, and biology (gene classification). Computer scientists have intensively investigated solutions to accelerate KNN on platforms including CPU [93, 46, 138], GPU [160, 85, 120], and hardware

accelerators [156, 211]. As the most time-consuming computation in KNN is performing the Euclidean distances, Accel-Bench's accelerated version also focuses on Euclidean distance between data points and randomized cluster centers using accelerated matrix multiplications.

We leverage the implementations from KNN-CUDA [196] repo for state-of-the-art CPU and GPU baselines. Accel-Bench's accelerated version modified the Euclidean distances implementation as KMeans to apply cuBLAS and invoke Tensor Cores for further acceleration. In addition, Accel-Bench also includes RTNN [223] implementation as another accelerated implementation using Ray Tracing cores.

PageRank (PR)

PR [148] is a representative graph algorithm that is especially useful in Search engines [167], social networks [83, 220], and data mining [205]. The input of PR is a vector that contains the score of all vertices (i.e., web pages or users) in a graph and an adjacency matrix that represents the connectivity of vertices in the page. The main algorithm of PR computes Equation 4.5 iteratively.

$$PR(v) = \frac{1-d}{|V|} + d \sum_{u \in N^-(v)} \frac{PR(u)}{|N^+(u)|} \tag{4.5}$$

In Equation 4.5, V is a set of vertices v, d is a set damping factor, $|V|$ is the number of vertices, $N^-(v)$ is the set of vertices that are incoming neighbors of v, $|N^+(u)|$ is the number of vertices that are outgoing neighbors of u, and $PR(u)$ is the score of vertex u. Since Equation refeq:PR is essentially a matrix-vector multiplication. Therefore, PR is an ideal candidate for using the acceleration (GEMV) in AI/ML accelerators. For the

baseline CPU implementation, Accel-Bench uses the GAP Benchmark suite [21] with a sparse vector matrix implementation (SpMV). We also use cuBLAS [142] without Tensor Cores to accelerate the vector-matrix multiplication on the GPU.

Short-Time Fourier Transform (STFT)

STFT is a well-known Fourier transform used for spectral analysis in applications such as digital signal processing [24], seismic analysis [116, 117], music and audio analysis [126, 132], Electroencephalography (EEG) [109, 5]. STFT breaks a longer time signal into shorter segments of equal length, and then computing the forward Fourier transform on each segment. STFT plots these segments to produce a spectrogram. As each segment is performing an iteration of FFT, Accel-Bench's implementation of STFT leverages hardware-accelerated FFT functions from tcFFT [111].

Accel-Bench's CPU baseline uses the implementation from Rosetta Code [60]. The GPU baseline comes from cuFFT [143] without Tensor Core supports.

Triangle Counting (TC)

TC plays a crucial role in characterizing graphs [39]. TC is also popular in computing important statistics such as clustering coefficients. Several ways to implement the TC algorithm include those centered on Parallelism [16, 208, 200], MapReduce [106], and wedge (a path of length two) sampling [172].

Accel-Bench's accelerated version computes TC using matrix multiplication by taking the lower and upper halves matrices, L and U, of adjacency matrix, A, so that $A = L + U$. Then, Accel-Bench's accelerated version calculates $B = LU$, followed by

$C = A \circ B$ where $\circ$ is the Hamming product. Finally, compute the total sum of triangles as $n = \frac{1}{2} \sum_i \sum jC_{ij}$. The baseline CPU implementation also comes from the GAP [21].

4.4 Experimental Methodology

Accel-Bench can run on both real machines and architectural simulators. This section provides the details of our evaluation platforms.

4.4.1 Evaluation Platform

Name	Component	Notes
CPU	Core i5 12600K	3.7 GHz
Main memory	64 GB	DDR4-3200
GPU (default)	RTX 2080	2944 CUDA Cores 368 Tensor Cores 8GB Device Memory
GPU (3090)	RTX 3090	10496 CUDA Cores 328 Tensor Cores 24GB Device Memory
GPU (4090)	RTX 4090	16384 CUDA Cores 512 Tensor Cores 24GB Device Memory

Table 4.3: Machine configurations

To evaluate the effect of programming in hardware accelerated API and compare the performance, Accel-Bench evaluates the benchmark applications using machines with various generations of NVIDIA GPUs. We selected NVIDIA GPUs for the following reasons.

Benchmark	Small	Medium	Large
CED	6 MB[197]	25 MB[197]	100 MB[197]
FHEW	20.64 MB [35]	516 MB [35]	1032 MB [35]
GRM	246 MB [183]	984 MB [183]	1.4 GB [183]
Heat2D/Head3D	7.5GB [27]	25.6 GB [27]	59 GB [27]
KM	67 MB [27]	134 MB [27]	268 MB [27]
KNN	8 GB	32 GB	128 GB
PR	37.9 MB [135]	201 MB [135]	461 MB [135]
STFT	1 MB [206]	2 MB [206]	4 MB [206]
TC	37.9 MB [135]	201 MB [135]	461 MB [135]

Table 4.4: Volume and sources of the datasets used

First, recent NVIDIA GPUs offer various hardware accelerators, including Tensor Cores for AI/ML workloads, RT cores for ray tracing algorithms, and NVENC as hardware video codecs. Second, despite the absence of dedicated hardware accelerators, some CUDA library functions (e.g., cuFFT for FFT, convolution in cuDNN) can map to existing hardware accelerators, and the function call can be easily swapped to the corresponding hardware accelerators, if necessary. Some CUDA library functions can internally leverage Tensor Cores to achieve better performance than the programmer's implementation. Third, due to CUDA's popularity, simulators support CUDA functions that reduce the complexity of porting. Finally, using the same library implementations will allow this work to compare the performance among different generations quickly without revisiting the code.

Table 4.3 shows the machine configuration and 3 GPUs used in this work. The default server in this work uses an Alder Lake processor with base frequency running at 3.7 GHz. We turned off the efficient cores on this processor to ensure the best single-thread performance. The server contains 64 GB of main memory and runs Ubuntu 20.04 with

kernel version 5.15.76. We used three different GPUs at comparable market segments in Turing (RTX 2080), Ampere (RTX 3090), and Ada Lovelace (RTX 4090) architectures. The default configuration uses the RTX 2080 server.

To evaluate Accel-Bench on architectural simulators, we used the default server to run Accel-Sim. We selected Accel-sim as this simulator supports the hardware accelerated APIs that we used to implement Accel-Bench.

4.4.2 Datasets

This work evaluates Accel-Bench with different dataset sizes. For each benchmark, we create datasets with small, medium, and large volumes to access the effect of Accel-Bench under different working set sizes. We used datasets from publicly available sources for CED, STFT, PR, and TC. For the rest, we use publicly recognized generators to synthesize the data Table 4.4 summarizes the sources and the size of our datasets.

4.5 Results

This section summarizes our evaluation of Accel-Bench. The result shows that the hardware-accelerated API-based paradigm that Accel-Bench promotes is on par with or outperforms the state-of-the-art implementation's performance.

4.5.1 Performance

On average, using hardware accelerated API as the programming paradigm can speed Accel-Bench applications by 1.14× to 1.77× compared to the state-of-the-art GPU

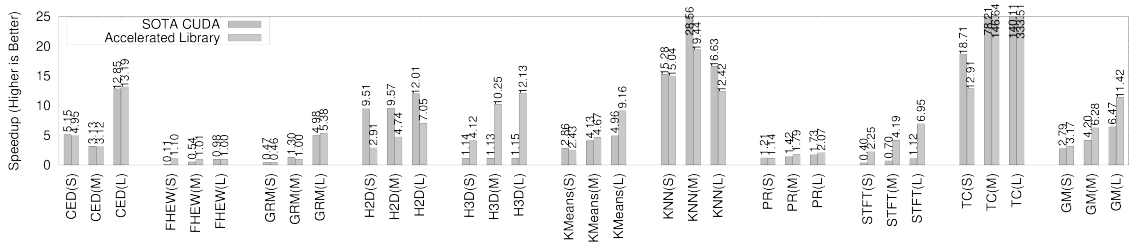

Figure 4.4: The performance result on the default machine

implementations, despite that GPU implementations are already 2.8× to 6.5× faster than CPU implementations. Figure 4.4 details the relative speedup of running Accel-Bench on the default server, compared to the CPU baseline.

The performance gain of using hardware-accelerated APIs is generally more significant when dataset sizes become more extensive. As the data structures that hardware-accelerated APIs accept do not always fit the original application's data structures, Accel-Bench must contain code to explicitly convert and prepare data structures for the inputs and outputs of hardware-accelerated APIs. When the dataset becomes larger, the increased complexity in the accelerated function's counterpart helps mitigate the overhead of such a process.

H2D and H3D provide another aspect that shows the strength of hardware-accelerated APIs. Despite simulating the thermal states using stencil operations, H3D additionally considers twice as many diagonals as H2D. Therefore, the computation of H3D becomes significantly higher than H2D. The current Accel-Bench implementation maps stencil operations into convolution and the rest in FFT. Using the convolution function for stencil will result in zero elements in computation and negate the effect of hardware acceleration to a certain degree. FFT requires high overhead in data conversion but is also

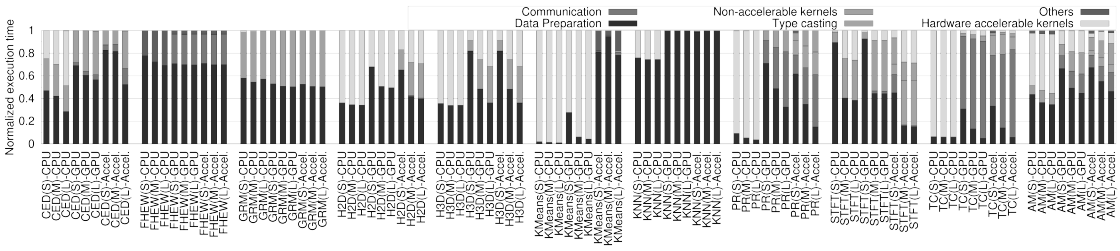

Figure 4.5: The latency breakdown in Accel-Bench applications

less optimized due to the absence of FFT accelerators on NVIDIA GPUs. Therefore, the amount of computation in functions that hardware-accelerated APIs can accelerate does not allow hardware-accelerated APIs to gain performance. In contrast, hardware-accelerated APIs become more effective as these functions take more computation in H3D.

Figure 4.5 provides the breakdown of latencies in various versions of Accel-Bench applications. Without hardware-accelerated functions, the CPU baseline averages 46%, 50%, and 52% for the small, medium, and large datasets in the hardware-accelerable code regions. This confirms that Accel-Bench selected the most critical code in each application. With customized GPU kernels, the hardware-accelerable code regions only account for these three datasets' 12%, 18%, and 19% of time. The hardware-accelerated version brings down the portion of hardware-accelerable code to just 3%, 8%, and 9%.

The file I/O and data deserialization code emerge as the most critical stages and consume 67%, 55%, and 46% of the time of the three datasets. The data movement overhead between the host main memory and GPU/accelerators also consumes a comparable amount of time as the hardware-accelerated code, with an average of 4%, 8%, and 7% in these three datasets. The result suggests the increasing importance of data movements in hardware-accelerator-based architectures. Even if the file I/O can be fully optimized with in-memory

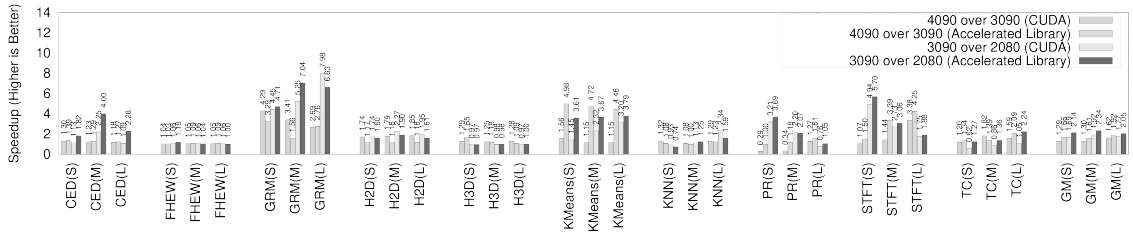

Figure 4.6: The performance comparison between two generations of NVIDIA GPUs and Tensor Cores

database/storage, the movement overhead between the host main memory and accelerators is still not negligible.

4.5.2 Performance comparison and projection for later generations

Another reason for advocating programming using hardware-accelerated API is the potential of performance scaling with the evolvement of hardware accelerators. We perform experiments that run unmodified Accel-Bench implementations using CUDA cores, and hardware accelerated library on RTX 2080, RTX 3090, and RTX 4090 to compare their relative performance on two generations of cards targeting similar market segments. Figure 4.6 shows our result. Despite RTX 3090 has 10496 CUDA cores, 3.57× more than that on RTX 2080 of the default server using RTX 2080, the geometric mean of speedup over the same application on RTX 2080 only ranges between 1.69× and 1.92×. However, using the hardware accelerated library, the same benchmark on RTX 3090 achieves an average speedup over RTX 2080 by 2.05× to 2.34×, even though the amount of tensor cores on RTX 3090 is slightly fewer than RTX 2080. The result reveals that the architectural innovation of hardware accelerators like Tensor Cores would power more performance gain than relying on increasing conventional GPU cores.

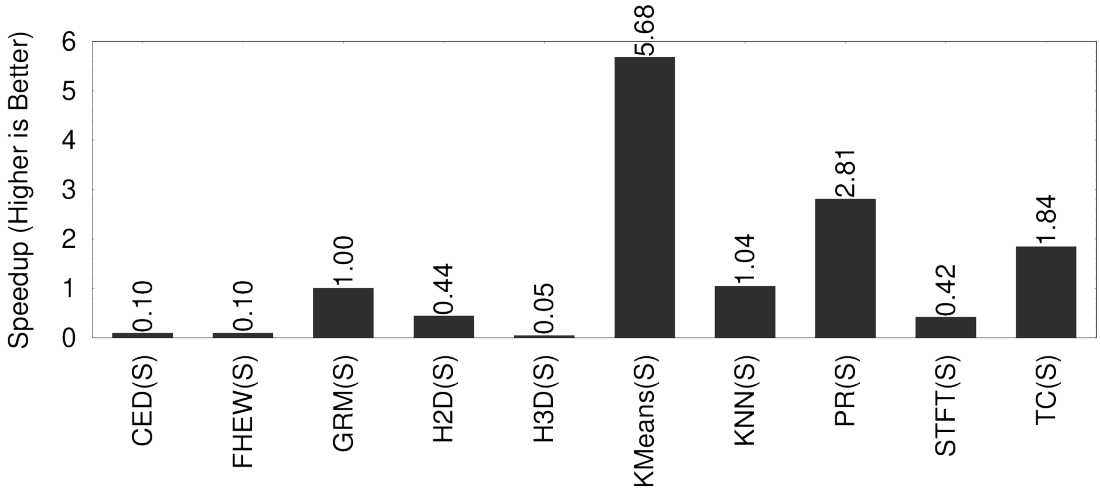

Figure 4.7: The speedup of hardware-accelerated function in Accel-sim over their GPU baseline implementations

Comparing RTX 3090 with RTX 4090, which has 16,384 CUDA cores, 1.56× more than that of RTX 3090, the performance gain has geometric means between 1.29× and 1.62×. RTX 4090 also has 1.56× more Tensor Cores than RTX 3090, but the performance gain has geometric means between 1.61× and 1.96×, revealing more significant performance scaling over hardware accelerated function potentially due to the new type of parallelism (e.g., matrix tiling parallelism) that hardware-accelerated functions can better exploit.

4.5.3 Support for Accel-Sim

Supporting architectural simulators allows Accel-Bench to assist the development of more generic hardware accelerators on selected functions. We examined the composed hardware-accelerated Accel-Bench applications and compared the performance of GPU implementations both on Accel-sim for this purpose.

Figure 4.7 shows the relative speedup of using hardware-accelerated functions compared to their GPU counterparts. Unlike running on real hardware, where hardware-

accelerated functions can help most applications to receive speedup, we only see KNN, H2D, KMeans, PR, and TC get a significant performance gain. These applications all use cuBLAS GEMM/GEMV, which can potentially use Tensor Cores acceleration in their implementations. It seems that the version of Accel-sim that we are working on has implemented the GEMM/GEMV function using hardware accelerators.

However, for other benchmark applications, we have seen a significant slowdown because the current implementations of CONV and FFT in Accel-sim do not leverage hardware accelerators but use pure CUDA software implementations. The higher complexity of implementing these functions hurts performance. On the other hand, this result also pointed out the need for hardware accelerators in the programming paradigm that we envisioned in this work.

4.6 Related Work

Here, in this section, we focus solely on prior work relevant to accelerator Accelerator centric programming. Compared to many previous benchmark suites that have attempted to capture aspects of accelerators, we present a very diverse collection of applications. Spanning 3 kernels over 7 domains, that can be categorized into 4 dwarfs.

Previous benchmarks often focus on the investigation of certain characteristics or methods of performance improvement. Parboil [182] contains a collection of benchmarks from throughput computing application from several domains. Benchmarks, including Chai [67] and GenomicsBench [183], have a some focus on analysis of task and data parallelism, although contain very somewhat target domains. Some benchmark suites all

together lean into concentrated domains. SpMV [224] provides a number of applications for Sparse matrix vector multiplication on FPGA. The GAP Benchmark suite [21] focuses on graph applications, characterization, and optimizations, providing multiple implementations. GenomicsBench [183] addresses the lack of benchmark suites for bioinformatics in computing research.

Altis [89], Rodinia [27], and Chai [67] each target heterogeneous computing similarly to Accel-Bench. Altis [89], much like Accel-Bench takes inspiration from other benchmark suites, in particular Rodinia [27] and SHOC [47], and modernize the applications. However, these benchmark suites do not support emerging hardware accelerators that Accel-Bench does, to help construct new architecture. In regards to their approach of applications, none have a concentrated effort on specific kernels, but do provide several alternative implementations of an application, such as Chai [67] and GAP [21]. There has been great interest in the direction of application designs, trying to identify the direction that the next generation of accelerators and how to best utilize them. Daniel suggests that as the computational complexity of programs, and the problem they were designed to solve grow, such as large language models contain parameters in the billions [30, 113, 13], the "cost of building and running large DL models has led some researchers to declare further improvements in DL are becoming unsustainable" [214]. O'Boyle has suggested creating another layer of abstraction between the programmer and hardware accelerators through the compiler to allow the advancement of hardware, without the tax of refactoring code and mapping to new APIs being placed on the programer [146]. Accelerators will continue to make improve their performance with each generation, such as TPUs [97, 96]. Furthermore, these accel-

erators will continue to be applied to applications outside of their domain [88, 87, 19, 11]. Accel-Bench is the first benchmark suite to quantify hardware accelerators performance outside of their domain, allow for the design and evaluation of new accelerators, and allow for the isolated improvements of a single kernel and its uses to a wide breath of applications, verifying improvements to previous and alternative implementations.

4.7 Conclusion

As computer architecture shifts the path of seeking performance for applications using hardware accelerators, we must let more applications capable of leveraging these architectural innovations. To achieve this goal, we need to research in two directions: a programming paradigm that helps exploit hardware accelerators and hardware accelerators with features supporting a broader spectrum of applications. Accel-Bench fills in the demand by revisiting the design of a set of applications and recomposing these applications using popular mathematical kernels of modern accelerators. Accel-Bench runs on both real hardware and architectural simulators. This work generates two critical insights. First, a hardware-accelerated-API-centric programming model is evenly or more competitive than conventional performance programming methods. Second, as architectural innovations focus more on hardware accelerators, we have seen more significant gains with upgraded hardware using a hardware-accelerated-API-centric programming model.

We envision Accel-Bench will encourage more exploration of hardware-accelerated-API-centric programming models. We also anticipate Accel-Bench can help identify and design architectures to optimize the potential performance issues in such models.

Chapter 5

Conclusion

Through these three works, TPUPoint, T2SP, and Accel-Bench, I have constructed a case to not simply improve upon what is known as these accelerators continue to be constructed in the foreseeable future, but considering the wider impact to many domains critical to fields outside of Ai/ML. Through the works displayed, the opportunity of underutilized accelerators, that their hardware and software can be untangled and less-restricting, and the foundation to expand to many hardware's and domains alike, are the groundwork for such improvements within the field of accelerated hardware.

TPUPoint examines the conventional notions that applications enable perfect utilization of their domain accelerators. Demonstrating profiling and optimization techniques to demystify the performance of the TPU accelerator. Opens argument to investigate applications outside AI/ML domain for potential full utilization. T2SP demonstrates software stacks that can be made agnostic to accelerator hardware, rather than the conventional tightly intertwined hardware software stack. Providing evidence that an unbinded accelera-

tor software stack can achieve non-trivial performance to its counterparts when decoupled. In culmination of the previous two works, Accel-Bench presents the first benchmark suite that considers a set of algorithms that allow applications beyond hardware accelerators' target domains to take advantage of the innovations of hardware accelerators.Presenting insights into potential programming paradigms for composing performance code. This work can guide the development of democratized hardware accelerators. Encapsulation the essence of the total of these works.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. *USENIX*, 12th(265-283):265–283, November 2016.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow, Large-scale machine learning on heterogeneous systems, November 2015.

[3] R. Adolf, S. Rama, B. Reagen, G. Wei, and D. Brooks. Fathom: reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[4] Zafar Ahmad, Rezaul Chowdhury, Rathish Das, Pramod Ganapathi, Aaron Gregory, and Yimin Zhu. Fast stencil computations using fast fourier transforms. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 8–21, 2021.

[5] Amirmasoud Ahmadi, Vahid Shalchyan, and Mohammad Reza Daliri. A new method for epileptic seizure classification in eeg using adapted wavelet packets. In *2017 Electric Electronics, Computer Science, Biomedical Engineerings' Meeting (EBBT)*, pages 1–4. IEEE, 2017.

[6] Alibaba. AI Matrix, 2018.

[7] AMD. AMD matrix cores. https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf, 2023.

[8] AMD. Versal ACAP DSP Engine Architecture Manual. https://docs.xilinx.com/r/en-US/am004-versal-dsp-engine/Advanced-Math-Applications, 2023.

[9] Anima Anandkumar, Rong Ge, Daniel Hsu, Sham M. Kakade, and Matus Telgarsky. Tensor Decompositions for Learning Latent Variable Models:, 2012.

[10] Apple Inc. Apple M1. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/, 11 2020.

[11] M. Arora, S. Nath, S. Mazumdar, S.B. Baden, and D.M. Tullsen. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *IEEE Micro*, 32(6):4–16, Nov 2012.

[12] Abdon Atangana and Juan Jose Nieto. Numerical solution for the model of rlc circuit via the fractional derivative without singular kernel. *Advances in Mechanical Engineering*, 7(10):1687814015613758, 2015.

[13] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[14] Roni Avissar and Roger A Pielke. A parameterization of heterogeneous land surfaces for atmospheric numerical models and its impact on regional meteorology. *Monthly Weather Review*, 117(10):2113–2136, 1989.

[15] Roni Avissar and Roger A Pielke. A parameterization of heterogeneous land surfaces for atmospheric numerical models and its impact on regional meteorology. *Monthly Weather Review*, 117(10):2113–2136, 1989.

[16] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811, 2015.

[17] Brett W. Bader, Michael W. Berry, and Murray Browne. *Discussion Tracking in Enron Email Using PARAFAC*, pages 147–163. Springer London, London, 2008.

[18] Baidu. DeepBench: Benchmarking deep learning operations on different hardware, 2017.

[19] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 94–103,

New York, NY, USA, 2010. ACM.

[20] Paul Barham and Michael Isard. Machine Learning Systems are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183, Bertinoro Italy, May 2019. ACM.

[21] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[22] Sanjukta Bhowmick, Victor Eijkhout, Yoav Freund, Erika Fuentes, and David Keyes. *Application of Alternating Decision Trees in Selecting Sparse Linear Solvers*, pages 153–173. Springer New York, New York, NY, 2010.

[23] Jiri Blazek. *Computational fluid dynamics: principles and applications*. Butterworth-Heinemann, 2015.

[24] Eduardo Cabal-Yepez, Armando G Garcia-Ramirez, Rene J Romero-Troncoso, Arturo Garcia-Perez, and Roque A Osornio-Rios. Reconfigurable monitoring system for time-frequency analysis on industrial equipment through stft and dwt. *IEEE transactions on industrial informatics*, 9(2):760–771, 2012.

[25] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[26] Christopher C Chang, Carson C Chow, Laurent CAM Tellier, Shashaank Vattikuti, Shaun M Purcell, and James J Lee. Second-generation plink: rising to the challenge of larger and richer datasets. *Gigascience*, 4(1):s13742–015, 2015.

[27] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)* [28], pages 44–54.

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[29] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11, 2010.

[30] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[31] Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. A versatile software systolic execution model for GPU memory-bound kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–81, Denver Colorado, November 2019. ACM.

[32] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 269–284, New York, NY, USA, 2014. Association for Computing Machinery.

[33] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. DaDianNao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, 2014.

[34] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 367–379. IEEE Press, 2016.

[35] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds. Cryptology ePrint Archive, Paper 2016/870, 2016. https://eprint.iacr.org/2016/870.

[36] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 115–126, New York, NY, USA, 2010. Association for Computing Machinery.

[37] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[38] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation, 2021.

[39] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680, 2011.

[40] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger. Serving DNNs in real time at datacenter scale with project Brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[41] Eric Chung, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Jeremy Fowers, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Kalin Ovtcharov, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Michael Papamichael, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, Doug Burger, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, and Shlomi Alkalay. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38(2):8–20, March 2018.

[42] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. DAWNBench: An end-to-end deep learning benchmark and competition. *NIPS ML Systems Workshop*, 2017.

[43] Bruce Collie and Michael F.P. O'Boyle. Program lifting using gray-box behavior. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 60–74, 2021.

[44] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[45] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 46–57, 2019.

[46] Dan Koschier. CompactNSearch. https://github.com/InteractiveComputerGraphics/CompactNSearch, 2022.

[47] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, pages 63–74, 2010.

[48] Luka Daoud, Dawid Zydek, and Henry Selvaraj. A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing. In *Advances in Systems Science: Proceedings of the International Conference on Systems Science 2013 (ICSS 2013)*, pages 483–492. Springer, 2014.

[49] João P. L. De Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. Kernelfarer: Replacing native-code idioms with high-performance library calls. *ACM Trans. Archit. Code Optim.*, 18(3), jun 2021.

[50] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, 2005.

[51] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai. Deep direct reinforcement learning for financial signal representation and trading. *IEEE Transactions on Neural Networks and Learning Systems*, 28(3):653–664, 2017.

[52] Xiao Ding, Yue Zhang, Ting Liu, and Junwen Duan. Deep learning for event-driven stock prediction. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 2327–2333. AAAI Press, 2015.

[53] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 617–640. Springer, 2015.

[54] Luke Durant, Olivier Giroux, Mark Harris, and Nick Stam. Inside Volta: The world's most advanced data center GPU, May 2017.

[55] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wen-mei Hwu. Accelerating fourier and number theoretic transforms using tensor cores and warp shuffles. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 345–355, 2021.

[56] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* [58], pages 449–460.

[57] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* [58], pages 449–460.

[58] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460, 2012.

[59] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[60] Alexandre Felipe. Fast Fourier transform. https://rosettacode.org/wiki/Fast_Fourier_transform, December 2023.

[61] Joel H Ferziger, Milovan Perić, and Robert L Street. *Computational methods for fluid dynamics*. springer, 2019.

[62] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[63] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, Los Angeles, CA, June 2018. IEEE.

[64] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[65] Tom Gibara. Canny edge detector. https://rosettacode.org/wiki/Canny_edge_detector, November 2015.

[66] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. Automatic matching of legacy code to heterogeneous apis: An idiomatic approach. *SIGPLAN Not.*, 53(2):139–153, mar 2018.

[67] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Victor García-Floreszx, Simon Garcia De Gonzalo, Thomas B Jablin, Antonio J Pena, and Wen-mei Hwu. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 43–54. IEEE, 2017.

[68] Google. Tensorflow research cloud, 2020.

[69] Google Cloud. CPU platforms compute engine documentation, 2020.

[70] Google Cloud. Machine Types Compute Engine Documentation, 2020.

[71] Google Cloud. System architecture cloud TPU, 2020.

[72] Google LLC. Coral M.2 Accelerator Datasheet. https://coral.withgoogle.com/static/files/Coral-M2-datasheet.pdf, 2019.

[73] Google LLC. Coral USB Accelerator Datasheet. https://coral.withgoogle.com/static/files/Coral-USB-Accelerator-datasheet.pdf, 2019.

[74] TOBIAS GROSSER, ARMIN GROESSLINGER, and CHRISTIAN LENGAUER. Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[75] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 855–864, New York, NY, USA, 2016. Association for Computing Machinery.

[76] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 71–82, Virtual Event GA USA, September 2020. ACM.

[77] G. Hamerly, E. Perelman, and B. Calder. Comparing multinomial and k-means clustering for SimPoint. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 131–142, 2006.

[78] N. Hasabnis. Auto-tuning TensorFlow threading model for CPU backend. In *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, pages 14–25, 2018.

[79] J. W. Haskins and K. Skadron. Minimal subset evaluation: rapid warm-up for simulated hardware state. In *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*, pages 32–39, 2001.

[80] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.

[81] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[82] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 173–182, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[83] Julia Heidemann, Mathias Klier, and Florian Probst. Identifying key users in online social networks: A pagerank based approach. *https://aisel.aisnet.org/icis2010_submissions/79/*, 2010.

[84] Charles Hirsch. *Numerical computation of internal and external flows: The fundamentals of computational fluid dynamics*. Elsevier, 2007.

[85] Rama C Hoetzlein. Fast fixed-radius nearest neighbors: Interactive million-particle fluids. 2014. *http://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf*, 2014.

[86] Pedro Holanda and Hannes Mühleisen. Relational queries with a tensor processing unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA, 2019. Association for Computing Machinery.

[87] Pedro Holanda and Hannes Mühleisen. Relational queries with a tensor processing unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* [86].

[88] Kuan-Chieh Hsu and Hung-Wei Tseng. Accelerating Applications using Edge Tensor Processing Units. In *SC: The International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC

2021, 2021.

[89] Bodun Hu and Christopher J Rossbach. Altis: Modernizing gpgpu benchmarks. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–11. IEEE, 2020.

[90] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. TCUDB: Accelerating Database with Tensor Processors. In *the 2022 ACM SIGMOD/PODS International Conference on Management of Data*, SIGMOD 2022, 2022.

[91] W. Hummer, V. Muthusamy, T. Rausch, P. Dube, K. El Maghraoui, A. Murthi, and P. Oum. ModelOps: Cloud-based lifecycle management for reliable and trusted AI. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 113–120, 2019.

[92] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. Van Gool. AI Benchmark: All about deep learning on smartphones in 2019. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3617–3635, 2019.

[93] Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. A parallel sph implementation on multi-core cpus. In *Computer Graphics Forum*, volume 30, pages 99–112. Wiley Online Library, 2011.

[94] Intel. Gaudi® Training Platform White Paper. https://www.intel.com/content/www/us/en/content-details/784830/gaudi-training-platform-white-paper.html.

[95] Jihyuck Jo, Suchang Kim, and In-Cheol Park. Energy-Efficient Convolution Architecture Based on Rescheduled Dataflow. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(12):4196–4207, December 2018.

[96] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A Domain-specific Supercomputer for Training Deep Neural Networks. *Communications of the ACM*, 63(7):67–78, 2020.

[97] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12. Association for Computing Machinery, New York, NY, USA, 2017.

[98] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a Tensor Processing Unit. [97], page 1–12.

[99] Eugenia Kalnay, Masao Kanamitsu, and Wayman E Baker. Global numerical weather prediction at the national meteorological center. *Bulletin of the American Meteorological Society*, 71(10):1410–1428, 1990.

[100] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 711–726, New York, NY, USA, 2016. Association for Computing Machinery.

[101] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, and Mike Burrows. A learned performance model for the Tensor Processing Unit, 2020.

[102] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.

[103] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. arXiv, 2017.

[104] Simon Knowles. Graphcore. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–25, 2021.

[105] Tamara G. Kolda and Brett W. Bader. Tensor Decompositions and Applications. *SIAM Review*, 51(3):455–500, 2009.

[106] Tamara G Kolda, Ali Pinar, Todd Plantenga, C Seshadhri, and Christine Task. Counting triangles in massive graphs with mapreduce. *SIAM Journal on Scientific Computing*, 36(5):S48–S77, 2014.

[107] Serguei S Komissarov. Time-dependent, force-free, degenerate electrodynamics. *Monthly Notices of the Royal*

Astronomical Society, 336(3):759–766, 2002.
[108] Kung. Why systolic architectures? *Computer*, 15(1):37–46, January 1982.
[109] M.Kemal Kıymık, İnan Güler, Alper Dizibüyük, and Mehmet Akın. Comparison of stft and wavelet transform methods in determining epileptic seizure activity in eeg signals for real-time application. *Computers in Biology and Medicine*, 35(7):603–616, 2005.
[110] Gary Lauterbach. The path to successful wafer-scale integration: The cerebras story. *IEEE Micro*, 41(6):52–57, 2021.
[111] Binrui Li, Shenggan Cheng, and James Lin. tcfft: A fast half-precision fft library for nvidia tensor cores. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, 2021.
[112] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The Deep Learning Compiler: A Comprehensive Survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, March 2021.
[113] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.
[114] Sean Lie. Cerebras architecture deep dive: First look inside the hardware/software co-design for deep learning. *IEEE Micro*, 43(3):18–30, 2023.
[115] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal loss for dense object detection. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.
[116] Naihao Liu, Jinghuai Gao, Xiudi Jiang, Zhuosheng Zhang, and Qian Wang. Seismic time–frequency analysis via stft-based concentration of frequency and time. *IEEE Geoscience and Remote Sensing Letters*, 14(1):127–131, 2016.
[117] Naihao Liu, Jinghuai Gao, Bo Zhang, Qian Wang, and Xiudi Jiang. Self-adaptive generalized s-transform and its application in seismic time–frequency analysis. *IEEE Transactions on Geoscience and Remote Sensing*, 57(10):7849–7859, 2019.
[118] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 21–37, Cham, 2016. Springer International Publishing.
[119] Xin Liu, Huanrui Yang, Ziwei Liu, Linghao Song, Hai Li, and Yiran Chen. DPatch: An adversarial patch attack on object detectors. arXiv, 2019.
[120] Lixin Xue. Fixed Radius NN Search. https://github.com/lxxue/FRNN.
[121] S. Lloyd. Pleast squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.
[122] Tianjian Lu, Thibault Marin, Yue Zhuo, Yi-Fan Chen, and Chao Ma. Accelerating mri reconstruction on tpus. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2020.
[123] Tianjian Lu, Thibault Marin, Yue Zhuo, Yi-Fan Chen, and Chao Ma. Nonuniform fast fourier transform on tpus. In *2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pages 783–787, 2021.
[124] P. Luszczek, J. Kurzak, I. Yamazaki, and J. Dongarra. Towards numerical benchmark for half-precision floating point arithmetic. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5, 2017.
[125] Wei Ma, Mengjie Zhao, Ezekiel Soremekun, Qiang Hu, Jie M. Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. Graphcode2vec: generic code embedding via lexical and program dependence analyses. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 524–536, New York, NY, USA, 2022. Association for Computing Machinery.
[126] Tobias Maas. Sound analysis using stft spectroscopy [d]. *Bachelor Thesis, University of Bremen*, pages 1–47, 2011.
[127] J Macqueen. Some methods for classification and analysis of multivariate observations. *Multivariate Observations*, page 17, 1967.
[128] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O'Boyle. C2taco: Lifting tensor code to taco. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2023, page 42–56, New York, NY, USA, 2023. Association for Computing Machinery.
[129] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter. NVIDIA Tensor Core programmability, performance precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, 2018.
[130] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. Nvidia tensor core programmability, performance amp; precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, Vancouver, BC, Canada, 2018. IEEE.
[131] Pablo Antonio Martínez, Jackson Woodruff, Jordi Armengol-Estapé, Gregorio Bernabé, José Manuel García, and Michael F. P. O'Boyle. Matching linear algebra and tensor code to specialized hardware accelerators. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, CC 2023, page 85–97, New York, NY, USA, 2023. Association for Computing Machinery.
[132] Paul Masri, Andrew Bateman, and Nishan Canagarajah. A review of time–frequency representations, with application to sound/music analysis–resynthesis. *Organised Sound*, 2(3):193–205, 1997.

[133] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. MLPerf training benchmark. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 336–349, 2020.

[134] Giuseppe Mendicino, Jessica Pedace, and Alfonso Senatore. Stability of an overland flow scheme in the framework of a fully coupled eco-hydrological model based on the macroscopic cellular automata approach. *Communications in Nonlinear Science and Numerical Simulation*, 21(1-3):128–146, 2015.

[135] Henning Meyerhenke. Kronecker generator graphs. https://sites.cc.gatech.edu/dimacs10/archive/kronecker.shtml.

[136] Riccardo Miotto, Fei Wang, Shuang Wang, Xiaoqian Jiang, and Joel T Dudley. Deep learning for healthcare: review, opportunities and challenges. *Briefings in Bioinformatics*, 19(6):1236–1246, 05 2017.

[137] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 603–614, 2015.

[138] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340):2, 2009.

[139] UC Nkwunonwo, Malcolm Whitworth, and Brian Baily. Urban flood modelling combining cellular automata framework with semi-implicit finite difference numerical formulation. *Journal of African Earth Sciences*, 150:272–281, 2019.

[140] Nvidia. Fdtd3d - cuda c 3d fdtd. https://github.com/olcf/cuda-training-series/blob/master/exercises/hw2/readme.md.

[141] NVIDIA. NVIDIA Video Codec SDK. https://developer.nvidia.com/video-codec-sdk.

[142] NVIDIA. cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html, 2019.

[143] NVIDIA. cuFFT. https://docs.nvidia.com/cuda/cufft/index.html, 2019.

[144] NVIDIA. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core, 2022.

[145] NVIDIA. cuDNN. https://docs.nvidia.com/deeplearning/cudnn/index.html, 2023.

[146] Michael O'Boyle. Rethinking the Role of the Compiler in a Heterogeneous World. https://web.archive.org/web/20230609152845/https://www.sigarch.org/rethinking-the-role-of-the-compiler-in-a-heterogeneous-world/, July 2020.

[147] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. PMLB: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(1):36, Dec 2017.

[148] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web., November 1999. Previous number = SIDL-WP-1999-0120.

[149] S. Pati, S. Aga, M. D. Sinclair, and N. Jayasena. SeqPoint: Identifying Representative Iterations of Sequence-Based Neural Networks. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 69–80, 2020.

[150] Paul Beckmann. Hardware Accelerators Boost the Performance of Next-Generation SHARC Processors. https://www.analog.com/media/en/technical-documentation/tech-articles/hardware-accelerators-sharc.pdf, 2008.

[151] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, page 727–734, 2000.

[152] E. Perelman, M. Polito, J. . Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, page pp. 10, 2006.

[153] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, page 701–710, New York, NY, USA, 2014. Association for Computing Machinery.

[154] Gabriel Peyré. The numerical tours of signal processing-advanced computational signal and image processing. *IEEE Computing in Science and Engineering*, 13(4):94–97, 2011.

[155] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 65–78, Providence RI USA, April 2019. ACM.

[156] Reid Pinkham, Shuqing Zeng, and Zhengya Zhang. Quicknn: Memory and performance optimization of kd tree based nearest neighbor search for 3d point clouds. In *2020 IEEE International symposium on high performance computer architecture (HPCA)*, pages 180–192. IEEE, 2020.

[157] Premtoon, Varot and Koppel, James and Solar-Lezama, Armando. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1066–1082, New York, NY, USA, 2020. Association for Computing Machinery.

[158] Christos Psarras, Lars Karlsson, Jiajia Li, and Paolo Bientinesi. The Landscape of Software for Tensor Computations. *arXiv:2103.13756 [cs]*, 1(1):1–16, May 2021.
[159] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv, 2016.
[160] Rama C. Hoetzlein. cuNSearch. https://github.com/InteractiveComputerGraphics/cuNSearch, 2022.
[161] Michel Rappaz, Michel Bellet, Michel O Deville, and Ray Snyder. *Numerical modeling in materials science and engineering*. Springer, 2003.
[162] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278, 2016.
[163] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark, 2019.
[164] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc., 2015.
[165] André Robert. A stable numerical integration scheme for the primitive meteorological equations. *Atmosphere-Ocean*, 19(1):35–46, 1981.
[166] Andre Robert. A semi-lagrangian and semi-implicit numerical integration scheme for the primitive meteorological equations. *Journal of the Meteorological Society of Japan. Ser. II*, 60(1):319–325, 1982.
[167] Ian Rogers. The google pagerank algorithm and how it works. *http://www.iprcom.com/papers/pagerank*, 2002.
[168] Hongbo Rong. Programmatic control of a compiler for generating high-performance spatial hardware, 2017.
[169] Justin Salmon and Simon McIntosh-Smith. Exploiting hardware-accelerated ray tracing for monte carlo particle transport with openmc. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 19–29, 2019.
[170] Malavika Samak, Deokhwan Kim, and Martin C. Rinard. Synthesizing replacement classes. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.
[171] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. *ACM Trans. Database Syst.*, 42(3), July 2017.
[172] C Seshadhri, Ali Pinar, and Tamara G Kolda. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 7(4):294–307, 2014.
[173] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery.
[174] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 45–57, New York, NY, USA, 2002. Association for Computing Machinery.
[175] Mateusz Sitko, Maciej Pietrzyk, and Lukasz Madej. Time and length scale issues in numerical modelling of dynamic recrystallization based on the multi space cellular automata method. *Journal of computational science*, 16:98–113, 2016.
[176] JA Somers. Direct simulation of fluid flow with cellular automata and the lattice-boltzmann equation. *Applied Scientific Research*, 51:127–133, 1993.
[177] L. Song, F. Chen, S. R. Young, C. D. Schuman, G. Perdue, and T. E. Potok. Deep learning for vertex reconstruction of neutrino-nucleus interaction events with combined energy and time data. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3882–3886, 2019.
[178] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen. AccPar: Tensor partitioning for heterogeneous deep learning accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 342–355, 2020.
[179] A. Sriraman and T. F. Wenisch. μ Suite: A Benchmark Suite for Microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2018.
[180] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang. Tensaurus: A versatile accelerator

for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 689–702, 2020.

[181] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, Geoff Lowney, Adam Herr, Christopher Hughes, Timothy Mattson, and Pradeep Dubey. T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 181–189, San Diego, CA, USA, 2019. IEEE.

[182] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127:27, 2012.

[183] Arun Subramaniyan, Yufeng Gu, Timothy Dunn, Somnath Paul, Md Vasimuddin, Sanchit Misra, David Blaauw, Satish Narayanasamy, and Reetuparna Das. Genomicsbench: A benchmark suite for genomics. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–12, 2021.

[184] Rudolph Szilard. Theories and applications of plate analysis: classical, numerical and engineering methods. *Appl. Mech. Rev.*, 57(6):B32–B33, 2004.

[185] TEAlab. TEAlab/FFTStencils. https://github.com/TEAlab/FFTStencils/tree/main.

[186] TensorFlow. TensorFlow TPU models, 2019.

[187] TensorFlow. XLA: Optimizing compiler for machine learning, 2020.

[188] Robert L. Thorndike. Who belongs in the family? *Psychometrika*, 18(4):267–276, Dec 1953.

[189] Peter Torelli and Mohit Bangale. Measuring inference performance of machine-learning frameworks on edge-class devices with the MLMark Benchmark. White Paper.

[190] Peter Torelli and Mohit Bangale. Introducing the EEMBC MLMark Benchmark, 2019.

[191] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models. arXiv, 2019.

[192] Ursula Van Rienen. *Numerical methods in computational electrodynamics: linear systems in practical applications*, volume 12. Springer Science & Business Media, 2001.

[193] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[194] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. ScaleDeep: A scalable compute architecture for learning and evaluating deep networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 13–26, New York, NY, USA, 2017. Association for Computing Machinery.

[195] Luminita A Vese and Stanley J Osher. Numerical methods for p-harmonic flows and applications to image processing. *SIAM Journal on Numerical Analysis*, 40(6):2085–2104, 2002.

[196] Michel Barlaud Vincent Garcia, Éric Debreuve. kNN-CUDA. https://github.com/vincentfpgarcia/kNN-CUDA, 2018.

[197] Bay Wallpapers. Bay wallpapers. https://wallpapercave.com/bay-wallpapers.

[198] G. Wang, J. Xu, and B. He. A novel method for tuning configuration parameters of spark based on machine learning. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 586–593, 2016.

[199] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. BigDataBench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499, 2014.

[200] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D Owens. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 1–8, 2016.

[201] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Accelerating fully homomorphic encryption using gpu. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5, 2012.

[202] Yu Wang, Gu-Yeon Wei, and David Brooks. A systematic methodology for analysis of deep learning hardware and software platforms. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 30–43, 2020.

[203] Joachim Weickert. Applications of nonlinear diffusion in image processing and computer vision. *None*, 2000.

[204] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, page 408–409, New York, NY, USA, 2005. Association for Computing Machinery.

[205] Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S Dhillon, and Keshav Pingali. Scalable data-driven pagerank: Algorithms, system issues, and lessons learned. In *Euro-Par 2015: Parallel Processing: 21st Interna-*

tional Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings 21, pages 438–450. Springer, 2015.

[206] wiki. Violin sound. https://upload.wikimedia.org/wikipedia/commons/d/d1/Violin_for_spectrogram.ogg.

[207] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, 2(1):37 – 52, 1987. Proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists.

[208] Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[209] C. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. Machine learning at Facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, 2019.

[210] Abenezer Wudenhe and Hung-Wei Tseng. TPUPoint Github.

[211] Tiancheng Xu, Boyuan Tian, and Yuhao Zhu. Tigris: Architecture and algorithms for 3d perception in point clouds. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 629–642, 2019.

[212] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. HyGCN: A GCN accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29, 2020.

[213] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Design Test*, 34(2):60–68, 2017.

[214] Daniel M Yellin. The premature obituary of programming. *Communications of the ACM*, 66(2):41–44, 2023.

[215] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V. Le. QANet: Combining local convolution with global self-attention for reading comprehension. arXiv, 2018.

[216] Q. Yu, C. Wang, X. Ma, X. Li, and X. Zhou. A deep learning prediction process accelerator based FPGA. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1159–1162, 2015.

[217] Tim Zerrell and Jeremy Bruestle. Stripe: Tensor Compilation via the Nested Polyhedral Model. *CoRR*, abs/1903.06498:1–13, 2019.

[218] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(11):2072–2085, 2019.

[219] J. Zhang, J. Sun, W. Zhou, and G. Sun. An active learning method for empirical modeling in performance tuning. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 244–253, 2020.

[220] Huan Zhao, Xiaogang Xu, Yangqiu Song, Dik Lun Lee, Zhao Chen, and Han Gao. Ranking users in social networks with motif-based pagerank. *IEEE Transactions on Knowledge and Data Engineering*, 33(5):2179–2192, 2019.

[221] Bojian Zheng, Abhishek Tiwari, Nandita Vijaykumar, and Gennady Pekhimenko. Echo: Compiler-based GPU Memory Footprint Reduction for LSTM RNN Training, 2019.

[222] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko. Benchmarking and Analyzing Deep Neural Network Training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 88–100, 2018.

[223] Yuhao Zhu. RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 76–89, 2022.

[224] Ling Zhuo and Viktor K. Prasanna. Sparse matrix-vector multiplication on fpgas. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, FPGA '05, page 63–74, New York, NY, USA, 2005. Association for Computing Machinery.