UC Riverside
UC Riverside Electronic Theses and Dissertations

Title

General-Purpose Computing on Tensor Processors

Permalink

https://escholarship.org/uc/item/6rf257nn

Author

Hsu, Kuan-Chieh

Publication Date

2024

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

General-Purpose Computing on Tensor Processors

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

Doctor of Philosophy

in

Computer Science

by

Kuan-Chieh Hsu

September 2024

Dissertation Committee:

Dr. Hung-Wei Tseng, Chairperson
Dr. Zizhong Chen
Dr. Daniel Wong
Dr. Zhijia Zhao

Copyright by
Kuan-Chieh Hsu
2024

The Dissertation of Kuan-Chieh Hsu is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

This endeavor would not have been possible without the support of my advisor, Professor Hung-Wei Tseng. He is the committee chairman and also gave me advice and countless support along with my PhD study.

I would like to extend my sincere thanks to Professor Zizhong Chen, Professor Daniel Wong, and Professor Zhijia Zhao for their valuable comments and feedback as committee members.

Many thanks to my labmates and collaborators for supporting the journey this far.

I want to thank all my family members for supporting me in their own ways.

To my parents for all the support.

ABSTRACT OF THE DISSERTATION

General-Purpose Computing on Tensor Processors

by

Kuan-Chieh Hsu

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

Modern computer systems have become heterogeneous and consist of many emerging kinds of hardware accelerators as Dennard Scaling discontinues. Also, such domain-specific hardware accelerators fulfill the rapidly growing computing demands for applications including artificial intelligence (AI) and machine learning (ML). Beyond conventional computer components such as central processing units (CPUs) and memory, modern computers typically contain accelerators such as graphic processing units (GPUs), tensor processing units (TPUs), and neural processing units (NPUs). Although accelerators have various programming interfaces and execution models, a group of accelerators are tensor processors that improve system performance for any problem that uses matrix or tensors as input and/or outputs. Despite the differences among the microarchitectural designs of each, tensor processors essentially are hardware accelerators that focus on providing efficient matrix-based computation solutions.

In this dissertation, we envision a new programming paradigm that leverages tensor processors for general-purpose computing beyond the original application domains for

AI and ML. The framework should contain the following characteristics. First, the programming interface design for a heterogeneous system with tensor processors must be simple, easy to use, and can maintain great compatibility and portability across various systems. Second, the execution model of the framework should intelligently explore and exploit opportunities by using the tensor processors that deliver better performance and extend the spectrum of application domains. Finally, the framework solution must be cost-effective and energy-efficient and be able to accommodate algorithm redesign and transformation that support broader usages.

I proposed three research works in response to the envision. First, I proposed GPTPU, an open-source, open-architecture framework that allows users to explore the usage opportunity of tensor processors for general applications. Second, I proposed SHMT, a new programming and execution model that enables simultaneous parallel processing using heterogeneous processing units for the same function. Lastly, I proposed GSLD, a matrix computing library accommodating either dense or sparse matrix inputs that more intelligently uses dense matrix processors and scalar cores.

Contents

List of Figures **xi**

List of Tables **xii**

1 Introduction **1**

2 GPTPU: Accelerating Applications using Edge Tensor Processing Units **9**
2.1 Introduction . . . 10
2.2 Background . . . 14
2.2.1 TPU architecture . . . 14
2.2.2 Edge TPU . . . 15
2.3 Characterizing Edge TPUs . . . 16
2.3.1 The prototype Edge TPU accelerated machine . . . 17
2.3.2 Characterizing Edge TPU instructions . . . 19
2.3.3 Characterizing Edge TPU data and model formats . . . 21
2.4 Overview of the GPTPU System . . . 23
2.5 OpenCtpu—The GPTPU programming interface . . . 26
2.6 The GPTPU library and runtime system . . . 29
2.6.1 Task scheduling . . . 29
2.6.2 Tensorizer . . . 31
2.7 Optimizing applications for GPTPU . . . 35
2.7.1 General Matrix Multiply (GEMM) . . . 35
2.7.2 Other applications . . . 39
2.8 Experimental methodology . . . 41
2.8.1 The system platform . . . 41
2.8.2 The baseline application implementations . . . 42
2.9 Results . . . 43
2.9.1 Single core performance: GPTPU vs. CPU . . . 43
2.9.2 GPTPU-GEMM vs. 8-bit CPU GEMM . . . 47
2.9.3 Parallel processing with multiple Edge TPUs . . . 48
2.9.4 Comparison with GPUs . . . 50
2.10 Related work . . . 52

2.11 Conclusion 54

3 SHMT: Simultaneous and Heterogenous Multithreading **55**
3.1 Introduction 56
3.2 Background and motivation 60
3.2.1 Modern heterogeneous components 60
3.2.2 Generalization of domain-specific accelerators 62
3.2.3 Potential and challenges of SHMT 63
3.3 SHMT 66
3.3.1 Overview 66
3.3.2 Virtual Operations (VOPs) and High-Level Operations (HLOPs) . . 69
3.3.3 SHMT's runtime system 72
3.3.4 The basic work-stealing scheduler 74
3.3.5 Quality-Aware Work-Stealing (QAWS) policy 75
3.4 The SHMT system prototype 80
3.4.1 The system assembly 80
3.4.2 NPU implementations 82
3.5 Results 83
3.5.1 Benchmark applications 84
3.5.2 Speedup of end-to-end latency 85
3.5.3 Quality of QAWS policies 86
3.5.4 QAWS sampling rate 90
3.5.5 Energy consumption 90
3.5.6 Memory and communication overhead 94
3.5.7 Discussion on SHMT's limitation 95
3.6 Related work 96
3.7 Conclusion 100

4 GSLD: a Globally Sparse Locally Dense Matrix Computing Library **101**
4.1 Introduction 102
4.2 Background and motivation 105
4.2.1 Observation on real-world sparse data 105
4.2.2 The GSLD property 107
4.3 Overview of GSLD 112
4.4 GSLD implementation 114
4.4.1 The core design algorithm 114
4.4.2 The proposed diagonal-dominant CSR sparse format 115
4.4.3 The sparse computation of applications 118
4.5 Experimental methodology 118
4.6 Results 119
4.6.1 Speedup of end-to-end latency 120
4.6.2 Full-scale testing on real-world samples 121
4.6.3 Diagonal area density 121
4.6.4 Memory space saving of sparse formats 122
4.7 Related work 123

4.8 Conclusion 124

5 Conclusions **125**

Bibliography **127**

List of Figures

1.1 Architecture design of tensor processors (Google's TPU as an example). . . 2
1.2 FMAs in NVIDIA Tensor Cores [107] 5
1.3 A sub-optimal performance case 6

2.1 The custom-built quad-EdgeTPU PCIe card 16
2.2 The GPTPU system overview 23
2.3 An OpenCtpu code sample 27
2.6 Speedup of GEMM GPTPU implementations using `FullyConnected` and `conv2D`, relative to the baseline CPU OpenBLAS implementations. 39

3.1 The execution model of (a) conventional heterogeneous computers (b) conventional heterogeneous computers with software pipelining, and (c) SHMT. 56
3.3 SHMT overview 67
3.4 The SHMT's programming model 70
3.12 Speedup v.s. problem sizes 95

4.1 32 representative real-world sparse matrices. 106
4.2 The Globally Sparse, Locally Dense (GSLD) property (number of samples) 108
4.3 Globally Sparse 109
4.4 Locally Dense 110
4.5 Performance of sparse matrix multiplication 113
4.6 The proposed diagonal-dominant + CSR sparse format (diablock+CSR) . . 116
4.7 GSLD - spMV end-to-end Speedup 119
4.8 Full-scale testing 120
4.9 Diagonal area density 122
4.10 Memory space saving of sparse formats 123

List of Tables

1.1 Commercialized AI/ML accelerators 4

2.1 The maximum OPS and RPS for each Edge TPU operator/instruction . . . 18
2.2 Sample functions from the OpenCtpu API 25
2.3 The input dataset sizes for the GPTPU benchmark applications 42
2.4 The (a) MAPEs and (b) RMSEs for GPTPU applications 46
2.5 The speedup and RMSE for GPTPU's GEMM library function relative to FBGEMM 47
2.6 The cost and power consumption of hardware accelerators that we compared in this work 50

3.1 The VOPs list in either vector or matrix tiling processing model types. . . . 71
3.2 Table of benchmarks 83
3.3 Communication Overhead 94

4.1 Sparse matrix charecterization 111
4.2 Sparse matrix computations 117

Chapter 1

Introduction

The evolutionary cycle of computing in history consistently comes from special-purpose accelerators to general-purpose processors. Central Processing Units (CPUs) were originally specialized machines with limited functionality for computing a few equations. In the '80s, the Intel 8087 floating-point coprocessors (FPU) [128] was one of the famous examples that was a separated floating point processing unit; Nowadays, floating pointing computations are already part of the functionality of the Arithmetic Logic Units (ALUs) in a general-purpose CPU. Similarly, Graphic Processing Units (GPUs) were originally accelerators for rendering 3-D graphics specifically. Today, GPUs are often referred to as general-purpose GPUs (GPGPUs) for most scenarios, including the famous usage of training or inferencing artificial intelligence (AI) and machine learning (ML) models.

Due to the rapid growth of computing demands of AI and ML workloads, emerging AI/ML accelerators focus on accelerating critical computational demands or computing kernels to facilitate the development of the AI revolution. Although these accelerators may

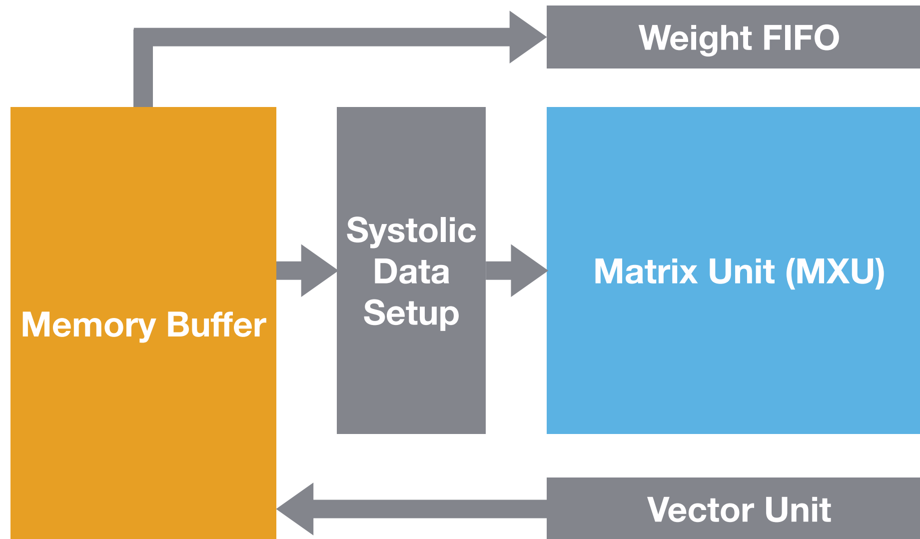


Figure 1.1: Architecture design of tensor processors (Google's TPU as an example).

have various microarchitectures or system component designs, they are tensor processors as they are essentially efficient matrix multiply processing units. Famous examples include Tensor Cores (TCs) [123, 122], Tensor Processing Units (TPUs) [81, 80, 78], and Neural Processing Units (NPUs) [11, 42, 184].

Characteristics of tensor processors

Figure 1.1 illustrates a high-level architecture overview of Google's TPU as a representative tensor processor. The matrix multiply units (MXU) is the core unit of TPU. MXU is a common design of tensor processors that can efficiently perform matrix multiply computation, serving the main purpose of accelerating AI/ML workloads with massive matrix-based computing kernels. The on-chip memory buffer can hold intermediate results of computation serving as inputs to the MXU. One critical characteristic of tensor processors is that the design purpose is to address any problem that uses tensors as inputs

and/or outputs. Beyond the CPU's capability of computing any pairs of numbers and GPTPU's capability of computing any pairs of vectors, we believe that tensor processors will eventually be capable of computing any pairs of tensors or matrices. Table 1.1 summarizes other characteristics of some commercial AI/ML accelerators.

However, the existing challenges limit the applicability of tensor processors toward general-purpose computing. We summarize these challenges including: (1) approximate result, (2) limited operator set, (3) lack of toolchain support, and (4) sub-optimal performance.

Name	Architecture	Usage	Cost	Standalone	Throughput	Power	Throughput/W
Apple M1 Neural Engine	NA	Inference	Mini: $669 Air: $999 Pro: $1,299	Integrated	11 TOPS	39 W	0.282 TOPS/W
Cloud TPUs	[81]	Inference:V1 Both:V2,V3	$2.40 / hour	Yes	V1: 23 TOPS	40 W	0.575 TOPS/W
					V2: 45 TOPS	200 W	0.225 TOPS/W
					V3: 90 TOPS	200 W	0.45 TOPS/W
Edge TPU	NA	Inference	$24.99	Yes	4 TOPS	2W	2 TOPS/W
A100 Tensor Core	[138, 107]	Both	$8,000	Integrated	624 TOPS	250 W	2.496 TOPS/W
Baselines							
RTX 3090	Ampere architecture	Both	$1,309	GPU	FP32: 36 TFLOPS	350 W	0.103 TFLOPS/W
VC707 FPGA	[116]	Both	$4,370 (Evaluation Kit)	Yes	1.877 TOPS	18.29 W	0.103 TOPS/W

Table 1.1: Commercialized AI/ML accelerators

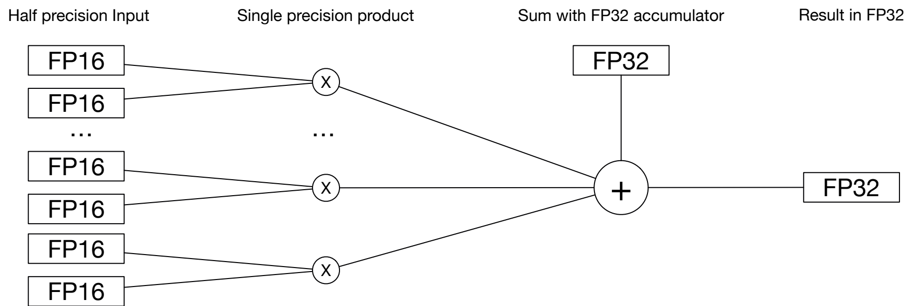


Figure 1.2: FMAs in NVIDIA Tensor Cores [107]

Approximate result

Many existing tensor processors can deliver high performance for AI/ML workloads in exchange for incorporating low-precision computing outputs. Figure 1.2 depicts the low-precision aspect of the Fused-Multiply-Add (FMA) operations design of NVIDIA's Tensor Cores [107]. The half-precision input design requires data type conversion of the intermediate result from a previous layer of operation and also contributes to the approximate results of the computing task. Although AI/ML tasks tend to be able to tolerate approximate results in many ways such as classification, this characteristic will become a challenge for other computing tasks for general-purpose computing.

Limited operator set

Tensor processors do not necessarily provide enough operator support for composing general applications. To facilitate high-performance computation for AI/ML workloads, tensor processors usually only need to provide an essential set of operators, and it typically includes convolution, matrix multiplication, polling, and activation functions, etc. However, this feature limits the application domains of tensor processors toward general-

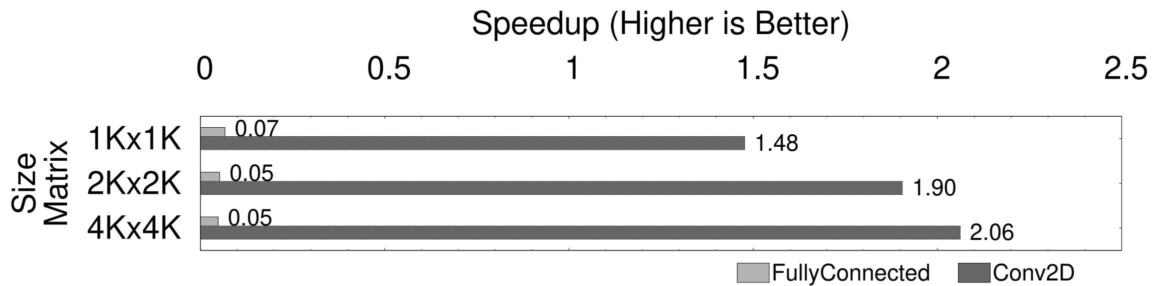


Figure 1.3: A sub-optimal performance case

purpose computing. Ultimately, the goal of general-purpose computing on tensor processors should require sufficient operator support that allows the combination or transformation of arbitrary programs. For example, the existing programming interface of NVIDIA's Tensor Cores only exposes `wmma::mma_sync();` as the core API. Although there are already some research works [42, 67, 30, 69, 82, 104, 39, 198] focused on expanding the applicability of tensor processors for certain tasks or application domains, this limitation is still challenging during the development of general-purpose computing on tensor processors.

Lack of toolchain support

The existing toolchains mainly focus on developing efficient support for AI/ML applications. Defining proper programming language with compiler support and providing comprehensive runtime support for general computing is challenging. For example, the toolchain support for Edge TPUs centers around ML models. The existing software stack requires file transformation from TensorFlow to the TFLite format and the usage of an ML-oriented C++ backend runtime system.

Sub-optimal performance

Lastly, even if all the aforementioned challenges are properly addressed, a potential general solution still needs to address the sub-optimal performance issue. Either with more operator set support or comprehensive operation combination or transformation, the composition of a new algorithm may overlook the impact on performance leveraging AI/ML-oriented operators when an intuitive methodology is chosen. For example, Figure 1.3 shows the performance comparison between using the FullyConnected operator as an intuitive option to implement GEMM and using the Conv2D operator that requires more complex transformation. Although both can achieve the same functionality, the former suffers from sub-optimal performance when the latter was not discovered. More details of this example can be found in Chapter 2 and Section 2.7.1.

This dissertation describes and evaluates all the aspects discussed above for general-purpose computing on tensor processors. Chapter 2 - 4 are organized in the conference paper format: each chapter consists of its introduction, background, problem statements, proposed framework or solution, implementation, methodology, results, related works, and conclusion sections. The organization of this dissertation is as follows.

Chapter 2 presents GPTPU: general-purposed tensor processing units, which is an open-architecture framework that enables the opportunity of general-programming leveraging tensor processors.

Chapter 3 presents SHMT: simultaneous and heterogeneous multithreading, a new programming and execution model that allows concurrent execution of heterogeneous accelerators for the same function for better utilizing the processing power within computers.

Chapter 4 presents GSLD: a sparse matrix computing library that extends the applicability of dense tensor processors for sparse matrix computation by leveraging the ”globally sparse, locally dense” insights.

Chapter 5 concludes this dissertation.

Chapter 2

GPTPU: Accelerating Applications using Edge Tensor Processing Units

Neural network (NN) accelerators have been integrated into a wide-spectrum of computer systems to accommodate the rapidly growing demands for artificial intelligence (AI) and machine learning (ML) applications. NN accelerators share the idea of providing native hardware support for operations on multidimensional tensor data. Therefore, NN accelerators are theoretically tensor processors that can improve system performance for any problem that uses tensors as inputs/outputs. Unfortunately, commercially available NN accelerators only expose computation capabilities through AI/ML-specific interfaces. Furthermore, NN accelerators reveal very few hardware design details, so applications cannot easily leverage the tensor operations NN accelerators provide.

This paper introduces General-Purpose Computing on Tensor Processing Units (GPTPU), an open-source, open-architecture framework that allows the developer and research communities to discover opportunities that NN accelerators enable for applications. GPTPU includes a powerful programming interface with efficient runtime system-level support—similar to that of CUDA/OpenCL in GPGPU computing—to bridge the gap between application demands and mismatched hardware/software interfaces.

We built GPTPU machine uses Edge Tensor Processing Units (Edge TPUs), which are widely available and representative of many commercial NN accelerators. We identified several novel use cases and revisited the algorithms. By leveraging the underlying Edge TPUs to perform tensor-algorithm-based compute kernels, our results reveal that GPTPU can achieve a 2.46× speedup over high-end CPUs and reduce energy consumption by 40%.

2.1 Introduction

The demand for AI and ML applications has exploded in recent years and the increase in AI/ML workloads has led to significant research advances in neural network (NN) accelerators, including Google's Edge Tensor Processing Units (Edge TPUs) [52] and Apple's Neural Engines [10] that are already presented as auxiliary hardware components in commodity systems. These NN accelerators' power/energy efficiency is orders-of-magnitude better than that of conventional vector processors (e.g., Graphics Processing Units [GPUs]) for the same workloads. Despite the differences among microarchitectures, most NN accelerators are essentially matrix processors that use tensors/matrices as inputs and outputs, and provide operators that facilitate NN computations.

Two decades ago, graphics processing units (GPUs) were just domain-specific accelerators used for shading and rendering. But intensive research into high-performance algorithms, architectures, systems, and compilers [144, 168, 47, 94, 112, 181, 145, 14, 186, 73] and the availability of frameworks like CUDA [121] and OpenCL [85], have revolutionized GPUs and transformed them into high-performance, general-purpose vector processors. We expect a similar revolution to take place with NN accelerators—a revolution that will create general-purpose matrix processors for a broader spectrum of applications. However, democratizing these NN accelerators for non-AI/ML workloads will require the system framework and the programmer to tackle the following issues:

(1) The microarchitectures and instructions of NN accelerators are optimized for NN workloads, instead of general matrix/tensor algebra. These auxiliary NN accelerators focus on latency per inference, but not yet on delivering computation throughput comparable to GPUs. Naively mapping conventional matrix/tensor algorithms to AI/ML operations will lead to sub-optimal performance. (2) Because many AI/ML applications are error tolerant, NN accelerators typically trade accuracy for area/energy-efficiency; when such a trade-off produces undesirable results, additional mechanisms are needed to make adjustments. (3) The programming interfaces of existing NN accelerators are specialized for developling AI/ML applications. Existing frameworks expose very few details about the hardware/software interfaces of NN accelerators, so programmers are unable to customize computation and the application can suffer from significant performance overhead due to adjusting the parameters/data bound to the supported ML models. (4) Tensor algorithms are traditionally time-consuming, so programmers have tailored compute kernels in favor

of scalar/vector processing. Such tailoring makes applications unable to take advantage of tensor operators without revisiting algorithms.

This paper bridges the gap between general-purpose programmability and domain-specific NN accelerators by presenting a full-stack system architecture that enables General-Purpose Computing on Edge TPUs (GPTPU). GPTPU tackles all the aforementioned challenges through providing a programming interface, a runtime system, compiler and libraries. With the system this paper proposes, programmers will be able to explore the enormous potential of the matrix processing model inherent in Edge TPU, a commercially available accelerator that can be part of a system-on-module (SOM) or be easily attached to various forms of computer systems. A commercialized Edge TPU can inference ML models at 4 TOPS (tera operations per second) with only 2 W of power consumption. The design that GPTPU demonstrates can also work for NN accelerators sharing similar architectures.

GPTPU provides a programming framework, including an Edge TPU-specific C/C++ extension, OpenCtpu and a runtime system. GPTPU offloads programmers from directly interacting with the accelerator's hardware to focus on the design of tensor-based algorithms for applications. OpenCtpu achieves more programming flexibility than existing domain-specific interfaces by exposing high-level tensor/matrix algebra operators (e.g., matrix multiplication) and low-level accelerator operators (e.g., convolution and matrix multiplication-accumulation) to the programmer, so programmers can design arbitrary tensor algorithms or customize operators that cannot be easily achieved using domain-specific languages.

The core of the GPTPU runtime system is our proposed *Tensorizer*, a module that dynamically evaluates input data and transforms data into ML models that the underlying Edge TPUs or other NN accelerators can efficiently perform inference operations on. Tensorizer handles quantization and calibration of input datasets and computation results, thereby minimizing the impact of limited precision on NN accelerators. The GPTPU runtime also schedules computation tasks and distributes prepared data to available NN accelerators in a manner that minimizes the data movement overhead.

Despite the Edge TPU's promising energy efficiency and recently open-sourced C++ API, documentation is vague regarding the Edge TPU's hardware/software interface and architecture. This lack of detail complicates the design of systems that fully exploit the Edge TPU's capabilities. To develop GPTPU, we measured the performance of available Edge TPU operators, reverse-engineered the Edge TPU hardware/software interface for data exchanges, and analyzed the Edge TPU architecture. We applied our understanding of the Edge TPU to optimize the backend runtime system for efficient task creation and data transformation. We then built a prototype GPTPU system with 8 Edge TPUs to allow concurrent GPTPU task execution.

We demonstrate the potential of the GPTPU system by modifying several key applications for financial computing, linear algebra, physics simulations and graph analytics. By revisiting the algorithms at the heart of these applications and using OpenCtpu, we show that GPTPU can simplify compute kernels; GPTPU preserves the nature of the application's tensor/matrix inputs and avoids explicit decompositions of datasets and algorithms into vector or scalar data. When used with the GPTPU-integrated applications, our proto-

type GPTPU system exhibits a 2.46× speedup and 40% reduction in energy consumption relative to modern CPUs.

By introducing the GPTPU system architecture, this paper makes the following contributions: (1) The paper introduces a novel full-stack system architecture to efficiently support general-purpose programming on Edge NN accelerators. (2) The paper characterizes the capabilities and previously unidentified architectural details of an inferencing hardware so that future research may exploit and optimize the GPTPU concept. (3) The paper proposes and implements Tensorizer to demonstrate the mechanism of dynamically and transparently mapping operators to NN models and Edge TPU instructions that lead to efficient use of underlying NN accelerators. (4) The paper demonstrates algorithm design for non-NN applications on NN accelerators by revisiting application algorithms to wisely use available accelerator instructions.

2.2 Background

This section briefly highlights TPU architectures and introduces alternative NN accelerators where GPTPU can potentially work.

2.2.1 TPU architecture

As most NN applications take matrix/tensor inputs and iteratively update parameters/weights from previous outcomes, the TPU microarchitecture accelerates NN tasks for modern ML applications by creating a systolic array that performs operations on the units of matrices/tensors. For inferencing tasks, the TPU treats one of the input matrices as the

trained model and the other as the samples to predict or classify. Taking matrices/tensors as the default inputs and outputs makes the TPU architecture and its corresponding execution model fundamentally different from conventional CPU/GPU architectures that compute on scalar/vector data pairs. TPUs also incorporate large on-chip memory to hold the intermediate results that later iterations reuse. Unlike conventional processors, TPUs do not contain on-chip instruction caches but simply use a CISC-style instruction-set architecture and rely on the host program to issue instructions through the system interconnect. And whereas conventional processors aim for precise computation results, TPU matrix units only support operations on a limited level of precision that is sufficient to satisfy the demands of modern ML applications while significantly reducing both TPU costs and energy requirements.

2.2.2 Edge TPU

This paper uses Edge TPUs, the trimmed-down versions of the Google Cloud TPU to build our system. Compared with Cloud versions, Edge TPUs contain smaller data memory (i.e., 8 MB).

The documented peak TOPS of Edge TPU is 4 TOPS under a 2 W TDP, while Cloud TPUs can achieve up to 90 TOPS under a 250 W TDP.

Although Google Cloud TPUs offer higher performance, we chose the Edge TPUs for the following reasons: (1) The Edge TPU hardware is publicly available, whereas Cloud TPU hardware is available exclusively through Google services; our use of Edge TPUs will therefore allow the developer and research communities to easily replicate, utilize, and optimize the system that this paper describes. (2) The current version of the Edge

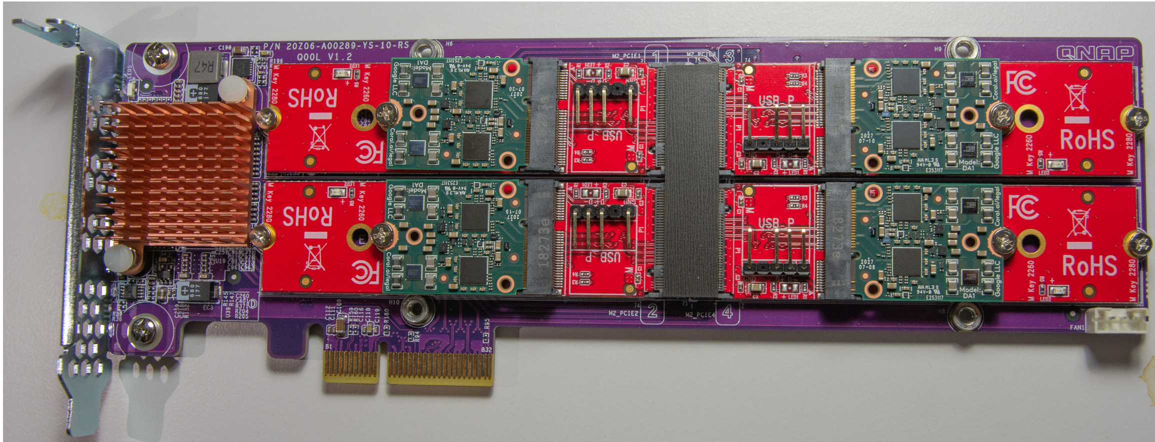


Figure 2.1: The custom-built quad-EdgeTPU PCIe card

TPU software framework has a partially open-sourced C++ backend that enables language front-end and runtime-system customization, whereas the Google Cloud TPU only provides a TensorFlow front-end without the backend source code. (3) Each Edge TPU offers better performance per watt than Cloud TPUs (i.e., 2 TOPS/W v.s. 0.36 TOPS/W) with just 2 W of power consumption and costs as little as USD 29, making a platform like GPTPU applicable to a broader range of computer systems than would be possible with the Google Cloud TPU alone.

2.3 Characterizing Edge TPUs

To directly measure the characteristics of Edge TPUs and determine their performance numbers, we built a customized machine with Edge TPUs attached. This section describes the architecture of our GPTPU testbed and reports the key performance characteristics of Edge TPUs that serve as the basis for our GPTPU system and application designs.

2.3.1 The prototype Edge TPU accelerated machine

The TPU architecture relies heavily on the CPU to distribute instructions, so our custom-built GPTPU hardware prototype aims at minimizing communication latency while efficiently using the limited system-interconnect bandwidth. To achieve this goal, the GPTPU prototype machine uses Edge TPUs in PCIe M.2 form factors; the Edge TPUs are attached directly to the PCIe system interconnect to allow lower latency and better bandwidth compared to other Edge TPU interconnect options, such as USB 3.0.

Operator	OPS (ops per second)	RPS (results per second)	Description
`conv2D`	10268.80	168240326.89	2D Convolution on a matrix
`FullyConnected`	51924.96	6646394.57	Input vector multiplies a weight matrix
`sub`	6273.28	82871343.60	Pair-wise subtraction on two matrices
`add`	6203.52	98293633.48	Pair-wise addition on two matrices
`mul`	14515.84	216469999.54	Pair-wise multiplication on two matrices
`crop`	4867.96	1562904391.76	Remove all unwanted elements outside of a sub-matrix from a given 2D matrix and return the sub-matrix
`ext`	1604.78	3637240203.38	Pad a matrix to the target dimensionality and return the padded matrix
`mean`	408.54	408.54	Count the average value of all elements in the matrix
`max`	477.08	477.08	Find the maximum value within a matrix
`tanh`	3232.31	2148232470.28	Perform tanh function on a matrix pair-wisely
`ReLu`	11194.26	4043196115.38	Leave only non-zero values on a matrix pair-wisely

Table 2.1: The maximum OPS and RPS for each Edge TPU operator/instruction

Each M.2 Edge TPU is designed to occupy only a single PCIe 2.0 lane, whereas most expansion slots that physically connect to the processor use multiple lanes. To efficiently use the precious PCIe lanes from the processor and the limited expansion slots, Figure 2.1 shows our custom-built quad-EdgeTPU PCIe expansion cards using QNAP QM2-4P-384A [137]. Each quad-EdgeTPU PCIe card contains 4× M.2 Edge TPUs with M.2 slots connected to a PCIe switch. The PCIe switch evenly divides the PCIe lanes (attached to each expansion slot) to four Edge TPUs and makes all Edge TPUs available to the host processor.

The current GPTPU hardware prototype contains an AMD Ryzen 3700X CPU with a Matisse microarchitecture that can reach a max-boost clock speed of 4.4 GHz with 32 MB LLC and 24× PCIe lanes available to all peripherals. Excluding the expansion slots used for essential peripheral devices, our hardware prototype can host 8× M.2 Edge TPUs, and each Edge TPU connects to the processor with just one hop (i.e., the PCIe switch) in the middle. The machine also contains 64 GB DDR4 main memory and an NVMe SSD. In addition to the hardware specifications, the prototype machine runs Ubuntu Linux 16.04 with kernel version 4.15.

2.3.2 Characterizing Edge TPU instructions

Due to the long interconnect latency and the absence of instruction caches on Edge TPUs, coped with the variable number of cycles and different types of input/output data resulting from the available CISC instructions, the GPTPU library, runtime system, and applications must use Edge TPU instructions wisely to achieve the best performance. The released Edge TPU performance numbers are available only in TOPS (tera operations per

second) and IPS (inferences per second). However, neither TOPS nor IPS provides sufficient insight for general-purpose software design because (1) TOPS or IPS is highly task specific, and (2) IPS is only representative for inferencing but not for other workloads [161].

We therefore use the *RPS (results per second)* as an additional metric to assess the benefits of each available Edge TPU operator/instruction. We define RPS as the amount of final result values an Edge TPU can generate within a second. We measure the OPS, RPS, and data-exchange rate of each tensor arithmetic instruction as follows: First, we begin timing the program and send the input datasets with size s to the target Edge TPU. Second, we issue and execute the same operator 10,000 times and measure the end-to-end latency (t_1) as well as the total number of result values (r_1) generated since the timer started. Third, we repeat the first and second step, but this time we execute the operator 20,000 times with the same input to get the end-to-end latency (t_2) and the total number of generated result values (r_2). Finally, we calculate the OPS of instructions/operators using Equation 2.1, their RPSs using Equation 2.2, and the data-exchange rate using Equation 2.5.

$$OPS_{operation} = \frac{10,000}{t_2 - t_1} \tag{2.1}$$

$$RPS_{operation} = \frac{r_2 - r_1}{t_2 - t_1} \tag{2.2}$$

$$\textit{Data-Exchange Rate} = \frac{s}{t_1 - (t_2 - t_1)} \tag{2.3}$$

Table 2.1 lists the RPS and the OPS of each Edge TPU instruction. The results lead to three observations on Edge TPUs. (1) `Conv2D` (convolution) achieves a very high

RPS given the high amount of operations required in equivalent CPU/GPU implementations, a hint that Edge TPU optimizes its microarchitecture for this instruction, (2) the OPS and RPS vary significantly for different types of instructions, and (3) the OPS and RPS are not strongly correlated because output varies; for example, instructions like `sub` generate outputs with the same dimensions as their inputs, but instructions like `FullyConnected` only produce vectors.

Our measurements also show that data-exchange performance does not vary among different types of instructions, but simply correlates with data size; transmitting 1 MB of data to an Edge TPU takes around 6 ms, while transmitting 8 MB of data that completely fill the on-chip memory takes 48 ms. The latency of copying data between the host main memory and Edge TPU's on-chip memory is significantly longer than any Edge TPU instruction.

2.3.3 Characterizing Edge TPU data and model formats

Edge TPU instructions ordinarily take two types of data inputs: (1) a tensor used for input datasets to be inferenced and (2) a model that the TFLite framework must generate and compile. Both types of inputs must be quantized before the host program sends them to the Edge TPU for computation. As GPTPU needs to use both types of inputs to achieve general-purpose computing, the GPTPU runtime library must translate one of the instruction inputs as a model for the Edge TPU.

The current Edge TPU toolchain only allows the user to generate models by invoking the Edge TPU compiler in the Python-based TFLite. With TFLite, translating a 2K $\times$ 2K matrix into a model takes 2.7 seconds on our testbed. This latency does not create

issues for inferencing tasks in ML applications, as inferencing tasks tend to reuse the same model for continuously changing inputs, and the overhead of creating models is amortized as input datasets scale. However, such amortization does not stand for many applications outside the ML realm. Unfortunately, neither the Edge TPU compiler code nor the Edge TPU model encoding has been released, so we have been unable to optimize the Edge TPU model-creation overhead.

To compensate for this lack of information, we reverse-engineered the Edge TPU model formats by creating models with different inputs, dimensions, and value ranges. We examined the models generated with the different inputs, and we identified the following key characteristics that allowed us to optimize the GPTPU runtime-system Edge TPU model-input instructions: (1) Models embed a 120-byte general header that allows TPUs to recognize the model-format version. The last 4 bytes of the header contain an unsigned integer describing the size of the data section. (2) Following the header, the data section contains binary-encoded 8-bit integers stored in row-major order. If the raw data values exceed the scope of 8-bit integers or are non-integers, the values must be scaled to fit in the 8-bit integer range. (3) A metadata section following the data section describes the data-section dimensions in terms of rows and columns. The metadata section also contains the scaling factor (f), a floating-point number that the compiler uses to rescale raw data into 8-bit integers; that is, an 8-bit integer value in the data section is calculated by multiplying its raw value by f. (4) The model encodes all values using little endian.

In addition to making the above observations, we determined that data dimensions do not necessarily reflect the dimensions of raw data inputs. The Edge TPU compiler adds

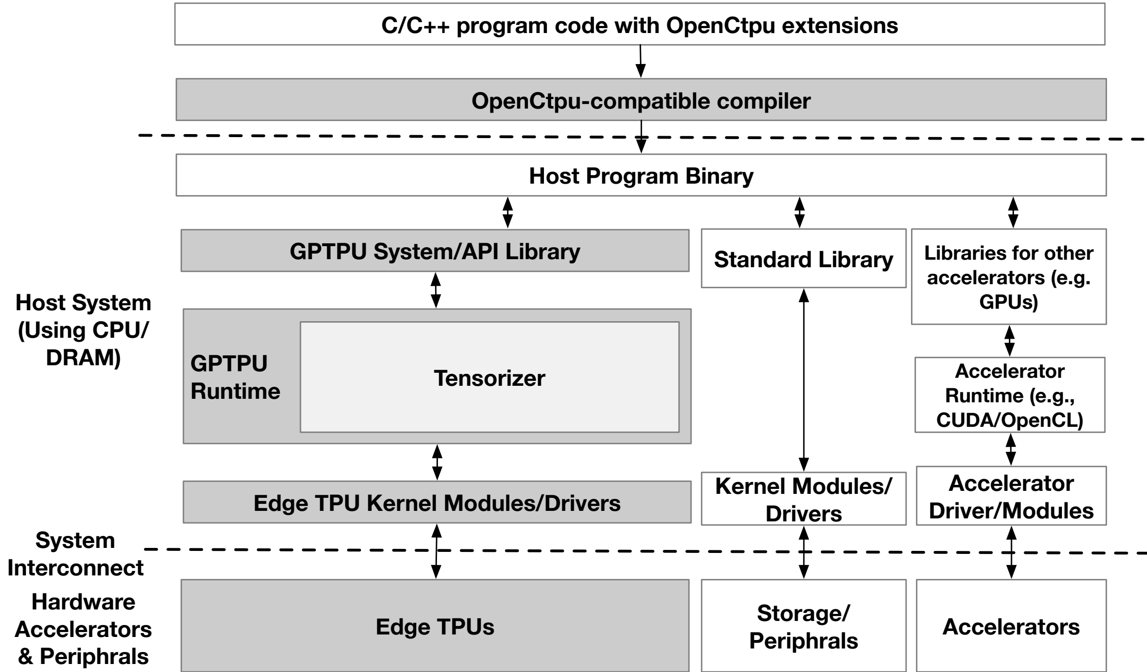


Figure 2.2: The GPTPU system overview

zero padding to unused elements (depending on the instructions) to reflect the hardware microarchitecture. Taking the most optimized instruction in Edge TPU architecture as an example, the Edge TPU compiler reshapes all input data into 128×128 sub-matrices. This implies that the Edge TPU's matrix unit is designed for computing on 128×128×8-bit matrices, in contrast to the Cloud TPU matrix unit, which is designed for 256×256×8-bit matrices.

2.4 Overview of the GPTPU System

Using insights learned from Section 2.3, this paper presents the system stack of the GPTPU framework that Figure 2.2 shows. GPTPU maintains the original heterogeneous-

computing system stack and extends the programming-language front end. GPTPU also provides a system library that can trigger the runtime system to (1) transform data, (2) schedule instructions for the underlying TPU hardware, (3) communicate with the TPU hardware, and (4) use the TPU hardware to accomplish computation tasks.

OpenCtpu serves as the programming-language front end for GPTPU. A programmer can use OpenCtpu to create a host program that describes TPU tasks and coordinates the use of heterogeneous computing resources and data exchanges in the system. A compiler supporting the OpenCtpu extensions will generate machine binaries compatible with the host CPU architecture and will generate code that transfers control to the GPTPU runtime system.

The GPTPU runtime system coordinates available TPU hardware. The runtime system schedules TPU operations from programmer-defined TPU tasks and prepares the inputs/outputs for TPU operations. Task scheduling and data preparation are left to the runtime system because doing so allows the GPTPU system to (1) adapt to changes in the underlying hardware without the need for reprogramming, (2) flexibly utilize underlying hardware resources, and (3) unburden the programmer of hardware-limitation details (e.g., data precision).

The following sections describe the design of the OpenCtpu programming-language front end (Section 2.5), the GPTPU runtime system (Section 2.6), and optimized operators/library function/applications (Section 2.7).

Synopsis	Description
`opencptu_dimension *openctpu_alloc_dimension(int dimensions, ...)`	This function allocates an `openctpu_dimension` data structure that describes the dimensionality of data in an input/output buffer. Depending on the input value of `dimensions`, the function can accept additional parameters that describe the dimensions.
`openctpu_buffer_t *openctpu_create_buffer( openctpu_dimension *dimension, void *data, unsigned flags)`	This function creates an input data buffer for TPU kernels. The pointer `dimension` provides a data structure with information about the number of data elements, the data type, and the dimensionality of the data. The pointer `data` provides the address for the raw data. The `openctpu_buffer_t` function returns a pointer to the created buffer.
`int *openctpu_enqueue(void *(*func)(void *), ...)`	This function enqueues a TPU task described in `func`. In addition to `func`, this function can accept an arbitrary number of arguments as `func` parameters. The function returns a task ID for the enqueued task.
`int *openctpu_invoke_operator(enum tpu_ops op, unsigned flags, ...)`	This function invokes a supported TPU operator (with operator arguments) and returns the operator output. The `flags` consist of parameters like the quantization method.
`int *openctpu_sync()`	This synchronization function requires all TPU tasks to complete before it returns.
`int *openctpu_wait(int task_id)`	This function blocks the calling thread until the specified task returns.

Table 2.2: Sample functions from the OpenCtpu API

2.5 OpenCtpu—The GPTPU programming interface

OpenCtpu is a C/C++ extension for general-purpose programming with GPTPU. OpenCtpu shares similarities with popular GPU programming models like CUDA [121] and OpenCL [85] in that OpenCtpu (1) places the control of application flow and device usage on the CPU-based host, (2) leverages virtual memory abstraction so that applications can specify data locations, (3) requires the programmer to explicitly manage data buffers for TPUs, and (4) provides functions that enable programmers to describe computation tasks for computation on TPUs.

A programmer can use OpenCtpu API functions and the C/C++ standard library to compose a TPU-accelerated program (see Table 2.2 for a list of representative OpenCtpu API functions). To create tasks for TPUs with the OpenCtpu API functions, a program needs to have the following: (1) kernel functions that describe the desired computation for TPUs, (2) input/output data buffers/structures for TPU kernels, and (3) enqueuing kernel functions and their inputs/outputs as tasks (OpenCtpu is similar to OpenCL in this respect). In the OpenCtpu programming model, all TPU operations within a task (i.e., an instance of a TPU kernel function) will perform in serial, but tasks can perform out of order in parallel. Therefore, the programmer may need to invoke synchronized primitives to ensure execution order and task completion.

To use Edge TPU operators in the kernel function, OpenCtpu provides an API function `openctpu_invoke_operator`. As the runtime system handles the precision, the programmer simply needs to specify the desired quantization method. In addition to

```c
#include <stdio.h>
#include <stdlib.h>
#include <gptpu.h>

// The TPU kernel
void *kernel(openctpu_buffer *matrix_a,
             openctpu_buffer *matrix_b,
             openctpu_buffer *matrix_c)
{
    // invoke the TPU operator
    openctpu_invoke_operator(conv2D, SCALE, matrix_a, \
                             matrix_b, matrix_c);
    return 0;
}

int main(int argc, char **argv)
{
    float *a, *b, *c; // pointers for raw data
    openctpu_dimension *matrix_a_d, *matrix_b_d, *matrix_c_d;
    openctpu_buffer * tensor_a, * tensor_a, * tensor_a;
    int size; // size of each dimension

    // skip: data I/O and memory allocation/initialization

    // describe a 2-D tensor (matrix) object for a
    matrix_a_d = openctpu_alloc_dimension(2, size, size);
    // describe a 2-D tensor (matrix) object for b
    matrix_b_d = openctpu_alloc_dimension(2, size, size);
    // describe a 2-D tensor (matrix) object for c
    matrix_c_d = openctpu_alloc_dimension(2, size, size);

    // create/fill the tensor a from the raw data
    tensor_a = openctpu_create_buffer(matrix_a_d, a);
    // create/fill the tensor b from the raw data
    tensor_b = openctpu_create_buffer(matrix_b_d, b);
    // create/fill the tensor c from the raw data
    tensor_c = openctpu_create_buffer(matrix_c_d, c);

    // enqueue the matrix_mul TPU kernel
    openctpu_enqueue(kernel, tensor_a, tensor_b, tensor_c);
    // synchronize/wait for all TPU kernels to complete
    openctpu_sync();

    // skip: the rest of the program
    return 0;
}
```

Figure 2.3: An OpenCtpu code sample

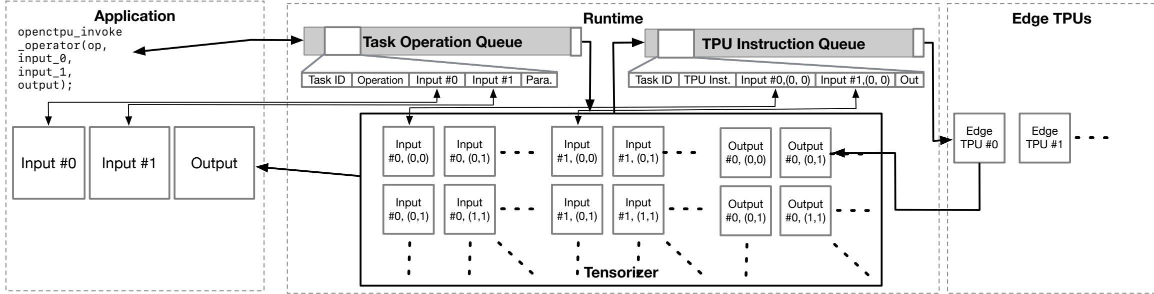


Figure 2.4: An overview of GPTPU's runtime system.

`openctpu_invoke_operator` that directly invoke Edge TPU instructions, OpenCtpu also implemented optimized overloaded operators on tensor data (e.g., matrix-add [+], matrix-sub [-], matrix-multiply [*]) to perform pair-wise matrix addition, subtraction and multiplication to further simplify programming.

The current OpenCtpu design brings several benefits to the GPTPU system. First, OpenCtpu gives the runtime system the flexibility to schedule and execute parallel tasks and to control the data movements associated with each task. Second, OpenCtpu avoids hardware complexity related to managing data consistency/coherency; OpenCtpu does this by leaving data management to software, as with GPGPU programming models. Third, OpenCtpu is designed to be complementary to existing heterogeneous computing platforms (we have verified that CUDA/OpenCL are compatible with our OpenCtpu extensions when run in the same program). We expect that CUDA/OpenCL can easily integrate our proposed extensions into their programming interface. The purpose of OpenCtpu simply serves as a transition for developers to easily exploit Edge TPU features and rethink/rewrite algorithms for applications, rather than replacing any existing heterogeneous programming standard.

Figure 3.4 shows an OpenCtpu code sample. The code contains a kernel function that uses the `conv2D` operator. Before creating a task instance from the kernel, the code must prepare two tensors, `a` and `b`, as inputs and another tensor, `c`, as the output. To describe the dimensionalities of these tensors, the program must call `openctpu_alloc_dimension` to create `openctpu_dimension` data structures for each tensor. The program can then make calls to `openctpu_create_buffer`, which contains the `openctpu_dimensions` values created for `a` and `b`, the pointers to the raw data for `a` and `b`, and the reserved data buffer for the product, `c`. To perform the `conv2D` operation, the program calls the `openctpu_invoke_operator` function, specifying `SCALE` as the quantization method for input/output data, `a` and `b` as the inputs, and `c` as the output for the Edge TPU operator (currently a one-to-one mapping to a fixed set of Edge TPU/CPU instructions). The kernel function returns when the operator is complete.

2.6 The GPTPU library and runtime system

The GPTPU runtime system receives tasks from the OpenCtpu front end, dynamically schedules tasks, and transforms input/output datasets for tasks. This section describes the design of the GPTPU runtime system.

2.6.1 Task scheduling

The GPTPU runtime task-scheduling policy is a dataflow-based algorithm on a front-end task operation queue (OPQ) and a back-end instruction queue (IQ) as Figure 2.4 highlights. An OPQ entry contains a task identification, the requested TPU operation, the

input and output locations, and parameters like the quantization method. An IQ entry contains similar items and also the actual underlying TPU instruction.

GPTPU gradually fills the OPQ during the execution of the user application. When the program calls the `openctpu_enqueue` function, the GPTPU runtime system initiates a new task ID for the invoked kernel function. The runtime system then executes the code designated by the function pointer using the set of parameters from the `openctpu_enqueue` call. The above process ends when `openctpu_invoke_operator` is called to request the involvement of a TPU operator/instruction.

The `openctpu_invoke_operator` function triggers the runtime system to create an OPQ entry with the task ID created from the current kernel function. The GPTPU runtime system then fills the rest of the queue entry with information passed to the `openctpu_invoke_operator` function. As the current OpenCtpu design serializes operators from a single kernel-function instance, kernel-function execution will be blocked until the operation finishes and each task has one operator from the `openctpu_invoke_o perator` function in the OPQ. Since OpenCtpu allows all tasks to execute in parallel, the GPTPU runtime system can issue entries in the OPQ to Tensorizer without considering their original order.

After Tensorizer optimizes, reshapes and transforms data and operations into instructions, Tensorizer divides a task into instructions in the IQ. The runtime system then schedules to the same Edge TPU if they share the same input, quantization flags, and the same task ID, but have different output locations—a scheduling approach that reduces movement overhead and the number of data transformations required. For other instruc-

tions, the GPTPU runtime system will use a first-come-first-serve policy to assign them to available Edge TPUs.

2.6.2 Tensorizer

Tensorizer is responsible for dynamic optimizations at the task level. Tensorizer transforms and optimizes programmer-requested operations into instructions, input tensors and models that enable efficient use of Edge TPUs.

Upon receiving a task from OPQ, Tensorizer first partitions the programmer-requested operation into Edge TPU instructions into sub-problems where each instruction works on its optimal data/model shapes using insights from Section 2.3.2. Tensorizer transforms the input data to minimize loss of accuracy due to the 8-bit precision of TPU matrix units for each Edge TPU operator.

Mapping operators into instructions

As OpenCtpu hides the hardware details from the programmer, programmer's tasks are agnostic to the granularity of inputs that optimize Edge TPU instructions. Tensorizer tackles this performance issue by dynamically partitioning these tasks into Edge TPU instructions working on their optimal data sizes/shapes (e.g., 128×128 matrices in most arithmetic instructions). As Edge TPU supports limited numbers of instructions/operators, we creates a set of rules that guides Tensorizer in rewriting tasks.

For pair-wise operators that calculate on pairs of values from both input matrices, including `add`, `sub` and `mul` or element-wise operators that calculate on each value of an input matrix, including `tanh` and `relu` the rule is straightforward. Tensorizer simply needs to first

divide the input data into tensors and models that contain sub-matrices with the optimal shape. Then, Tensorizer rewrites the operator into a set of Edge TPU instructions where each works on a sub-matrix or a pair of sub-matrices locating at the corresponding positions in the original inputs and collects the results in the corresponding memory locations.

For matrix-wise operators, including `mean` and `max`, Tensorizer still divides the input into sub-matrices with optimal shapes (i.e., both instructions favor 64×64 sub-matrices) and uses instructions to work on each sub-matrix. However, Tensorizer will additionally generate CPU code to aggregate the received values from results of instructions to produce the final outcome. An alternative approach is to create another sets of Edge TPU instructions and making the received values an input tensor/model to iteratively use Edge TPU to produce the result. Tensorizer does not take this approach as (1) the first round of executing `mean` or `max` instruction already shrinks the values to aggregate by a factor of 4096, and (2) the latency of moving data in the currently system architecture is significantly longer than aggregating results with CPU code.

For arithmetic operators, including `FullyConnected` and `conv2D`, Tensorizer applies mechanisms similar to the blocking algorithm for matrix multiplications [37] in rewriting tasks. If each input matrix is partitioned into $P \times Q$ sub-matrices, The resulting code will contain Edge TPU instructions that perform $P \times Q$ `FullyConnected` or `conv2D` instructions and CPU code that aggregates results into the final outcome. The CPU code only needs to add received values that requires very short latency to execute on modern processors. In addition, as CPU registers are wider than Edge TPU's data precision, aggregating results on CPU will allow the platform to reduce precision loss in results.

After rewriting operations into actual machine/accelerator code, Tensorizer will obtain the mapping between an input value and its location in the transformed tensor/model.

Data transformation

To minimize the inaccuracy of computation, Tensorizer carefully rescales values into fixed-point numbers and fill numbers into models or inference data arrays that Edge TPUs can accept. GPTPU determines the scaling factor for input datasets using (1) the sequence of operators, (2) the number of operators, and (3) the range of input data. As the data size of each Edge TPU instruction and the sequence of operators are known at runtime, the GPTPU system can estimate the number of logical arithmetic operations ($num_logical_operations$) that the instructions will generate. By discovering the maximum value (max) and the minimum (min) value of the dataset, the runtime system can estimate the range of output values and derive the model/tensor scaling factor. The general rule of the scaling factor S of an operator is

$$S = \frac{1}{max(|output_{max}|, |output_{min}|)} \tag{2.4}$$

where $output_{max}$ is the expected maximum output value and $output_{min}$ is the expected minimum output value. For most datasets, sampling is efficient enough in large datasets as previous work indicates that small subset of input data is representative for the whole dataset [93]. As GPTPU calculates S using the maximum absolute value of outputs, GPTPU prevents the case of overflow.

GPTPU applies different formulas for different types of operators. If the input data is a pair of $N \times N$ matrix, GPTPU estimates the scaling factor for each `conv2D` and `FullyConnected`, as:

$$S = \frac{1}{|max - min|^2 \times N} \tag{2.5}$$

For pair wise `add` and `sub`, GPTPU uses:

$$S = \frac{1}{2 \times |max - min|} \tag{2.6}$$

as the scaling factor. For pair wise `mul`, GPTPU uses:

$$S = \frac{1}{|max - min|^2} \tag{2.7}$$

as the scaling factor, and for other operators, GPTPU calculates the scaling factor as:

$$S = \frac{1}{|max - min|} \tag{2.8}$$

For example, consider a request that performs matrix multiplication and then pairwisely add another matrix on $N \times N$ matrices with data ranging from 0 to $n-1$. The maximum output value in the resulting matrix will be $2 \times N \times (n-1)^2$. The runtime system can choose $\frac{1}{2 \times N(n-1)^2}$ as the scaling factor.

The overhead of Tensorizer

Using the information we gained from reverse-engineering the Edge TPU model format as described in Section 2.3.3, we implemented the proposed Tensorizer to dynamically create models from arbitrary input data. The C-based Tensorizer can bring the latency of generating a model from a 2K×2K matrix down to 1.8 ms—a 1500× speedup

over the original Python-based Edge TPU TFLite compiler and shorter than the latency of data transfer. The GPTPU runtime system thus can overlap Edge TPU matrix-input data movements with Tensorizer to reduce the total latency of executing Edge TPU instructions from tasks.

2.7 Optimizing applications for GPTPU

Mapping a problem into a GPTPU application requires inputs/outputs to be transformed into tensors that Edge TPUs can operate on. Although many applications use data in tensor form, the Edge TPU instructions are optimized for NN workloads, meaning that naively applying the default tensor operators may not improve performance. Tensorizer helps to optimize performance in the task level, but using the most efficient operator for a task still requires programmer's optimization. This section describes GPTPU application design and optimization using matrix multiplication as an example.

2.7.1 General Matrix Multiply (GEMM)

To demonstrate the importance of designing algorithms to wisely use Edge TPU instructions, we explain the design of an efficient GEMM on Edge TPUs, a fundamental linear-algebra tool for matrices. GEMM takes two 2-dimensional tensors (matrices) as inputs and produces a single 2-dimensional tensor as output. We can calculate each element in the result matrix, C, obtained from a set of pairwise multiplications and accumulations from an $M \times N$ matrix, A, and an $N \times K$ matrix, B.

GEMM and the `FullyConnected` operator

The Edge TPU `FullyConnected` instruction offers an intuitive way to implement GPTPU GEMM, as the operator essentially produces a matrix-vector product.

A program can select either matrix A or matrix B and iterate through a column or row of the other matrix to produce the result, and matrix multiplication will be performed via the M or K `FullyConnected` operators.

The `conv2D` operator/instruction

Edge TPU's `conv2D` instruction can also perform multiplications and accumulations but in different orientations to derive the result. In conventional architectures, programmers implement convolutions by performing scalar-scalar or vector-vector multiplications and accumulations for higher efficiency. However, Table 2.1 shows that the RPS of convolution (i.e., `conv2D`) is 25$\times$ the RPS of matrix-vector multplications (i.e. `FullyConnected`) on Edge TPUs. Inspired by this observation, we therefore explore the implementation by changing the layout of input data and using `conv2D` to perform exactly the same number of multiplications and accumulations on the set of input numbers to leverage the high RPS of `conv2D` for a more efficient GEMM implementation.

The `conv2D` instruction takes one of its inputs as the kernel, multiplies each kernel element with an input element mapping to the corresponding location, and accumulates the result as an output element. Each `conv2D` instruction can produce a result matrix that has the same size as the non-kernel input.

Figure 2.5: The `conv2D` as implemented with stride

For an $M \times N$ input matrix, A, and an $L \times L$ kernel, B', each element in the `conv2D` $M \times N$ output matrix, C, is:

$$C_{i,j} = \sum_{q=0}^{L} \sum_{p=0}^{L} A_{i+p,j+q} \cdot B'_{p,q} (\forall\ 0 \leq i < M, 0 \leq j < N) \tag{2.9}$$

Targeting AI/ML workloads that are error tolerant (and so permit approximations), the Edge TPU `conv2D` instruction allows a programmer to assign a *stride* value (s_x, s_y) that treats inputs as groups of $s_x \times s_y$ sub-matrices and produces a corresponding result value for them.

Figure 2.5 illustrates the concept of `conv2D` with stride. We select (3, 3) as our stride, restricting `conv2D` to 9 numbers in a group; the `conv2D` operator only produces a

value for every 3 row/column elements in the abstracted outcome, as in Figure 2.5(c), from the source matrix, as in Figure 2.5(a), using the kernel in Figure 2.5(b). The final output of `conv2D` is a condensed matrix, as in Figure 2.5(d).

GPTPU uses `conv2D` and its striding feature to implement an efficient GEMM algorithm. The algorithm starts by reshaping both inputs that transform each row in the chosen source matrix into a sub-matrix whose size is determined by the selected stride (s_x, s_y). Ordinarily, both s_x and s_y are the round-up of the square root of the column dimension in the source matrix. The other input matrix serves as a list of kernels, where each kernel of size $s_x \times s_y$ contains a column from that matrix. When creating the kernels, the GPTPU GEMM algorithm fills the kernel elements to match the desired element-wise multiplications for GEMM. In other words, for a matrix with N columns and K rows, the resulting kernel matrix will contain N kernels where each kernel contains $\lceil\sqrt{K}\rceil \times \lceil\sqrt{K}\rceil$ elements. That being said, the resulting kernel matrix still contains exactly the same or similar amount of elements (i.e., $N \times (\lceil\sqrt{K}\rceil)^2$ v.s. $N \times K$) as the original input matrix. After transforming both inputs, `conv2D` iterates through all sub-matrices over each kernel with the selected stride and generates output identical to that of conventional matrix multiplication.

`FullyConnected` and `conv2D` together

Figure 2.6 shows the performance of GPTPU GEMM kernel implementations using `FullyConnected` and `conv2D` compared to the CPU baseline using OpenBLAS [192]. The `conv2D` implementation reveals a strong performance gain (a 2.06× speedup in the 4K×4K microbenchmark) over the CPU baseline. In contrast, the GPTPU GEMM im-

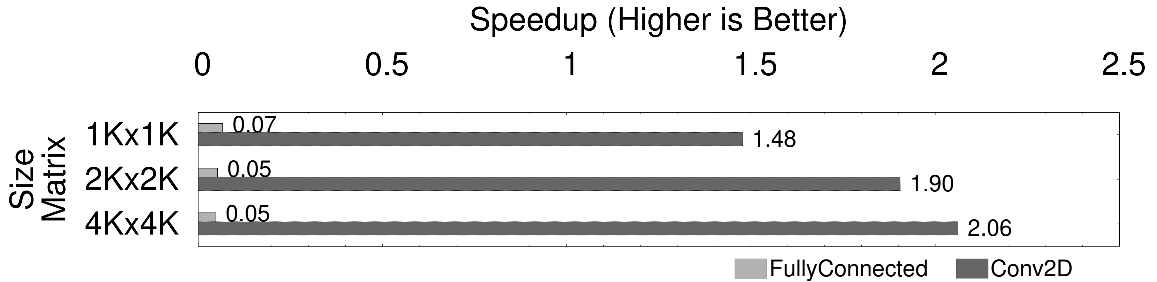


Figure 2.6: Speedup of GEMM GPTPU implementations using `FullyConnected` and `conv2D`, relative to the baseline CPU OpenBLAS implementations.

plementation cannot beat the CPU baseline without `conv2D` (i.e., when GEMM only uses `FullyConnected`).

Though the GPTPU GEMM algorithm incurs additional data-transformation overhead, GPTPU's `conv2D`-based GEMM significantly outperforms the conventional vector-product-based algorithm by 43× on our GPTPU platform. This is because the Edge TPU architecture highly optimizes `conv2D` and the favorable RPS of `conv2D` compensates for the additional overhead.

Since GEMM is a widely used, fundamental linear-algebra tool for matrices, GPTPU makes the core GEMM algorithm available as an optimized library function, `tpuGemm`, that GPTPU applications can invoke—just as CUDA invokes the cuBLAS GEMM implementation via the `cublasGemm` function [119].

2.7.2 Other applications

As with GEMM, our goal for all GPTPU applications is to utilize instructions with the highest RPS. We now summarize how we extended the GPTPU GEMM approach to other applications whose workloads we evaluate in the latter part of this paper.

PageRank

The PageRank algorithm [126] is a representative graph application. PageRank takes an adjacency matrix representing a graph as input. Both the baseline and the GPTPU implementations use the classic power method that iteratively performs matrix-vector multiplications. In contrast to CPU/GPU PageRank implementations that perform pairwise or vector-wise multiplications, the GPTPU PageRank implementation simply uses one `FullyConnected` instruction for each adjacency-matrix multiplication with a single vector.

HotSpot3D

HotSpot3D is a thermal-simulation tool for estimating the temperature of a chip made with 3D-stacking. The main algorithm gradually and iteratively updates each point on the chip, which is represented as a matrix with a weighted average of the point's closest neighbors in 8 different directions. The HotSpot3D algorithm can naturally map to `conv2d` with a 3×3 kernel without striding.

LU Decomposition (LUD)

LUD factors a matrix into a lower triangular matrix (L) and an upper triangular matrix (U) such that $L \times U$ yields the original matrix. Our GPTPU LUD implementation uses the recursive algorithm [28] via `crop`, `FullyConnected`, and `conv2D` to partition matrices and perform appropriate operations on different combinations of the partitioned matrices.

Gaussian elimination (Gaussian)

Like LUD, Gaussian is a method for solving a system of linear equations. Gaussian combines row swaps, the scalar multiplication of rows, and row additions until the lower left-hand triangular matrix contains only zeroes. For Gaussian, GPTPU uses `mul` to perform each row reduction.

Backpropagation (Backprop)

Backprop is foundational to NN supervised learning. We implemented a plain-vanilla version of Backprop to demonstrate the ML/AI-generalizable nature of GPTPU. For a feedforward NN, the GPTPU Backprop uses (1) multiple layers of `FullyConnected` and sigmoid activation functions in `ReLu`, (2) `add` for the actual backpropagation, and (3) `tpuGEMM` to derive weights for the delta matrix.

Black–Scholes (BlackScholes)

BlackScholes is a financial model for estimating the market price of stock options. GPTPU uses a ninth-degree polynomial function [3] with the `FullyConnected` instruction to compute the cumulative normal distribution function.

2.8 Experimental methodology

2.8.1 The system platform

We use exactly the same prototype machine described in Section 2.3 for all experiments performed with GPTPU.

Benchmark	Input Matrices	Data Size	Category	Baseline Implementation
Backprop	1×8K×8K	512MB	Pattern Recognition	[22, 62]
BlackScholes	1×256M×9	9GB	Finance	[183]
Gaussian	1×4K×4K	64MB	Linear Algebra	[22, 62]
GEMM	2×16K×16K	1GB	Linear Algebra	[192, 119, 32]
HotSpot3D	8×8K×8K	2GB	Physics Simulation	[22, 62]
LUD	1×4K×4K	64MB	Linear Algebra	[22, 62]
PageRank	1×32K×32K	4GB	Graph	[19]

Table 2.3: The input dataset sizes for the GPTPU benchmark applications

When performing experiments for baseline applications, we removed the TPUs from the machine.

For each application, we measured the end-to-end latency. We also measure the total system power using a Watts Up meter. When calculating energy consumption, we aggregate the total system power throughout the application execution time. On average, each active Edge TPU adds only 0.9 W to 1.4 W of power consumption, while a loaded AMD Matisse core in the GPTPU hardware prototype consumes from 6.5 W to 12.5 W. As GPTPU still relies on the CPU for the runtime system and data transformation, both CPUs and Edge TPUs can be active when running applications. The idle power of the experimental platform is 40 W, including the southbridge chip on the motherboard, NVMe-based storage devices as well as other peripherals connected to the system.

2.8.2 The baseline application implementations

For each application described in Sections 2.7, we compared our GPTPU implementations with (1) optimized CPU/GPU implementations from benchmark suites [22, 183] or (2) widely-used distributions [192, 119, 19]. Table 2.3 lists the input datasets and the

baseline implementations for each application we used in our experiments. We only select a subset of applictions from these benchmark suites because these are all applications that (a) preserve the form of matrix inputs and (b) can map their core algorithms to reasonable matrix operations. We do not expect GPTPU and Edge TPUs to be effective for applications that can only exploit vector arithmetics since Edge TPU's architecture is specialized for matrix operations. We also use Facebook's GEMM (FBGEMM) [32] for approximate computing on GEMM.

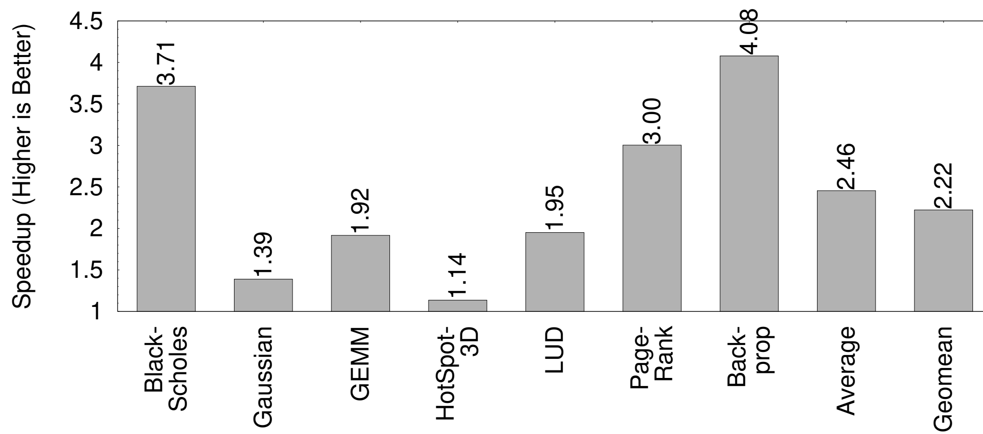
2.9 Results

This section describes the speedup, energy consumption, and accuracy observed for GPTPU when running different applications. Compared to modern CPU-based platforms running optimized code, GPTPU exhibits improved performance and significantly reduced energy needs. In addition, the GPTPU GEMM implementation yields more reliable results in approximation than a low-precision matrix-multiplication library run on a CPU.

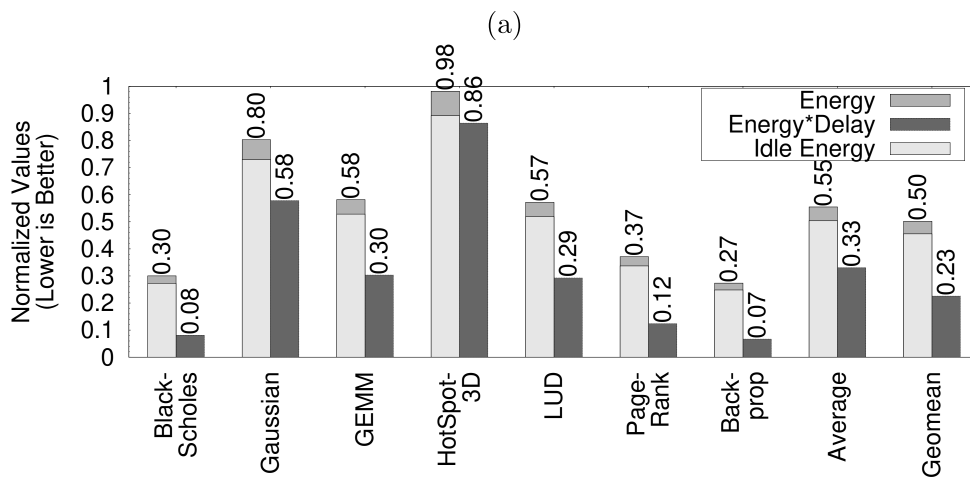
2.9.1 Single core performance: GPTPU vs. CPU

Figure 3.6 summarizes the speedup, energy consumption, and energy-delay of GPTPU-based applications. We used a single Edge TPU and a single CPU core to compare execution of workloads in our baseline tests to compare the per-core raw hardware capabilities.

Figure 3.6(a) compares end-to-end latency. The GPTPU system is, on average, $2.46\times$ faster than the CPU. For Backprop, the speedup is $4.08\times$ (not surprising given that



(a)



(b)

Figure 2.7: The application (a) speedup, (b) energy consumption, and energy-delay products for a single Edge TPU, relative to the baseline CPU implementations

the Edge TPU was originally designed for applications like Backprop). Excluding Backprop, the average speedup is still 2.19×. HotSpot3D actually experiences the least speedup with GPTPU. This is because GPTPU's HotSpot3D uses very small kernels and large inputs accompany each iteration, the data-movement overhead dominates end-to-end application latency. However, even under this scenario, GPTPU can still speed up the performance of HotSpot3D by 1.14×.

Figure 3.6(b) shows the relative energy consumption and energy-delay products for GPTPU applications vs. their CPU baseline implementations. GPTPU consumes only 5% of the active energy and only 51% of the idle energy that a CPU consumes (an energy savings of 45%), and even the worst-performing GPTPU benchmark still saves 3% overall system energy. For energy-delay products, which take both latency and energy consumption into consideration, applications run with GPTPU enjoy a 67% reduction over the baseline CPU. Excluding the top-performing Backprop, GPTPU still achieves an 40% energy savings and a 62% energy-delay improvement.

GPTPU sacrifices accuracy—but only to a limited degree. Table 2.4 measured the mean absolute percentage error (MAPE) and the root mean square error (RMSE) between the GPTPU and CPU application implementations using the default dataset from the benchmark and our randomly generated datasets with various ranges of values in their inputs. The MAPE is always less than 1% across all applications, regardless their ranges of input values. The average MAPE is 0.26%–0.33%. The largest RMSE we measured was an acceptable 0.98%. In some cases, the GPTPU results in higher error rates in compute on default datasets than on synthetic inputs with larger data ranges. This is because the

Benchmark	default	$-2^7 \leq x < 2^7$	$-2^{15} \leq x < 2^{15}$	$-2^{31} \leq x < 2^{31}$
Backprop	0.12%	0.17%	0.10%	0.11%
Blackscholes	0.18%	0.18%	0.18%	0.18%
Gaussian	0.00%	0.00%	0.00%	0.00%
GEMM	0.89%	0.90%	0.90%	0.90%
HotSpot	0.50%	0.49%	0.46%	0.46%
LUD	0.00%	0.00%	0.00%	0.00%
PageRank	0.61%	0.73%	0.73%	0.73%
Average	0.33%	0.35%	0.34%	0.34%

(a)

Benchmark	default	$-2^7 \leq x < 2^7$	$-2^{15} \leq x < 2^{15}$	$-2^{31} \leq x < 2^{31}$
Backprop	0.14%	0.17%	0.12%	0.12%
Blackscholes	0.33%	0.33%	0.33%	0.33%
Gaussian	0.00%	0.00%	0.00%	0.00%
GEMM	0.98%	0.91%	0.91%	0.91%
HotSpot	0.64%	0.64%	0.59%	0.59%
LUD	0.00%	0.00%	0.00%	0.00%
PageRank	0.41%	0.91%	0.91%	0.91%
Average	0.41%	0.42%	0.41%	0.41%

(b)

Table 2.4: The (a) MAPEs and (b) RMSEs for GPTPU applications

Range of Values		0–2	0–4	0–8	0–16	0–32	0–64	0–128
Speedup over FBGEMM		1.26	1.27	1.28	1.22	1.28	1.27	1.28
RMSE	FBGEMM	0.00	0.00	0.00	0.00	0.47	0.87	0.97
	TPUGEMM	0.00	0.00	0.00	0.00	0.00	0.00	0.01

Table 2.5: The speedup and RMSE for GPTPU's GEMM library function relative to FBGEMM

input values of synthetic datasets are typically normally distributed but the real, default datasets are not always normally distributed.

2.9.2 GPTPU-GEMM vs. 8-bit CPU GEMM

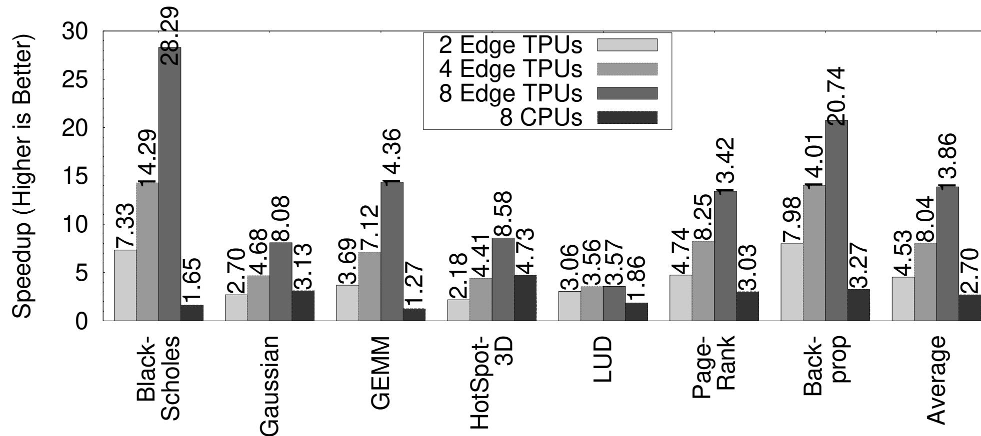
GPTPU allows single–Edge TPU performance to surpass single-CPU-core performance. That being said, the Edge TPU uses low-precision data types, whereas the baseline CPU implementations do not. To account for this difference when using approximate computing with the CPU cores, we compared the GPTPU implementation running with the state-of-the-art FBGEMM low-precision CPU matrix-multiplication library that intensively uses the latest AVX instructions to support 8-bit operations [32]. We did not include other workloads in this part as other workloads do not have implementations optimized for 8-bit CPU operations.

Table 2.5 shows the results for GPTPU's GEMM vs. FBGEMM using 1024×1024 matrices with positive integers and maximum input values ranging from 2 to 128 (we chose this data size only to accommodate FBGEMM's limitations). As Figure **??**(a) shows, GPTPU-GEMM consistently outperforms FBGEMM on high-performance CPU cores with 1.22× to 1.28× across all configurations. However, when the maximum matrix-entry value

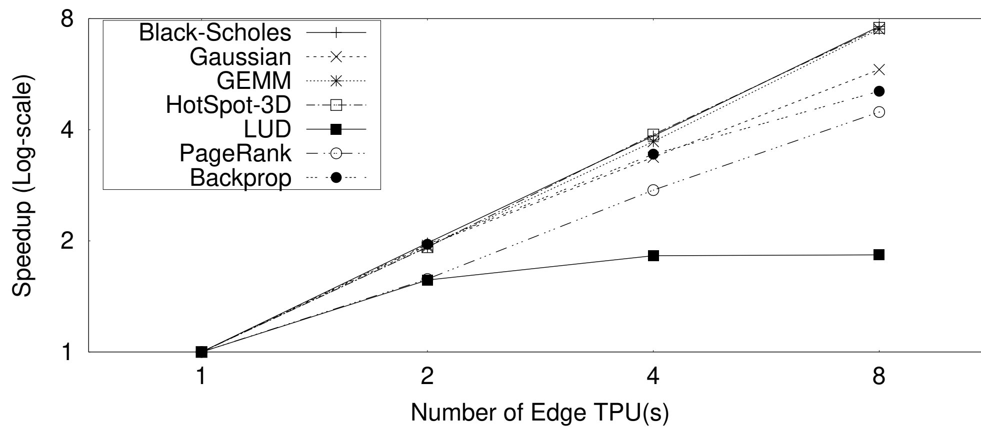
exceeds 16, FBGEMM's RMSE is poor as Table 2.5 presents, reaching 47% when the largest value within the dataset is 32. Furthermore, the FBGEMM RMSE goes as high as 97%, meaning that most result values are not convincing when the largest value is 128. In contrast, GPTPU-GEMM's RMSE is always less than 1% (0.82% when maximum value is 128). This is because FB's GEMM targets at error-tolerant ML applications but does not handle overflow cases. However, the performance evaluation indicates that even if the CPU baseline uses 8-bit operations, GPTPU-GEMM is faster.

2.9.3 Parallel processing with multiple Edge TPUs

The GPTPU runtime system uses a task queue that allows multiple Edge TPUs to process tasks in parallel. Even without programmer's explicit partitioning of tasks, Tensorizer also automatically generates parallel tasks from the user code. Figure 2.8(a) shows the speedup of adding more Edge TPUs into our system, without modifying the user code, compared with the single-core CPU baseline. With 8 Edge TPUs that consume similar active power as a single RyZen core, GPTPU achieves an average 13.86× speedup. In constrast, the 8-core, OpenMP-based CPU implementations can only achieve 2.70× speedup over the baseline. Figure 2.8(b) further shows log-scale performance with up to 8 Edge TPUs running GPTPU tasks, compared with single Edge TPU. The linear plots reveal good performance scaling for 6 out of 7 applications when the GPTPU runtime system executes tasks in parallel. The only exception is LUD, which already partitions matrices into four sub-matrices for computation using matrix-wise operators, making it difficult for Tensorizer to scale the performance in only one of the four partitions.



(a)



(b)

Figure 2.8: Performance scaling for multiple Edge TPUs

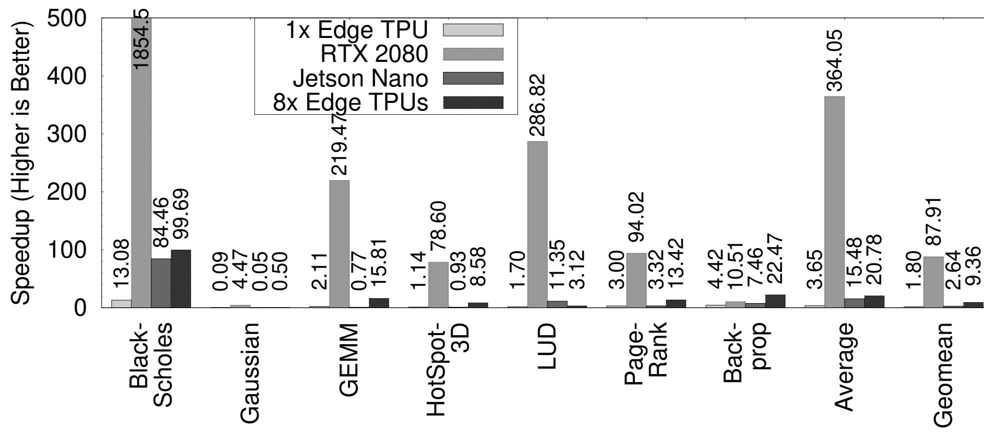
	Cost	Power Con.	Comment
Single Edge TPU	USD 24.99	2 W	
RTX 2080	USD 699.66	215 W	Now USD 1399
Jetson Nano	USD 123.99	10 W	
8× Edge TPU	USD 159.96	16 W	Using 4× dual Edge TPU modules

Table 2.6: The cost and power consumption of hardware accelerators that we compared in this work

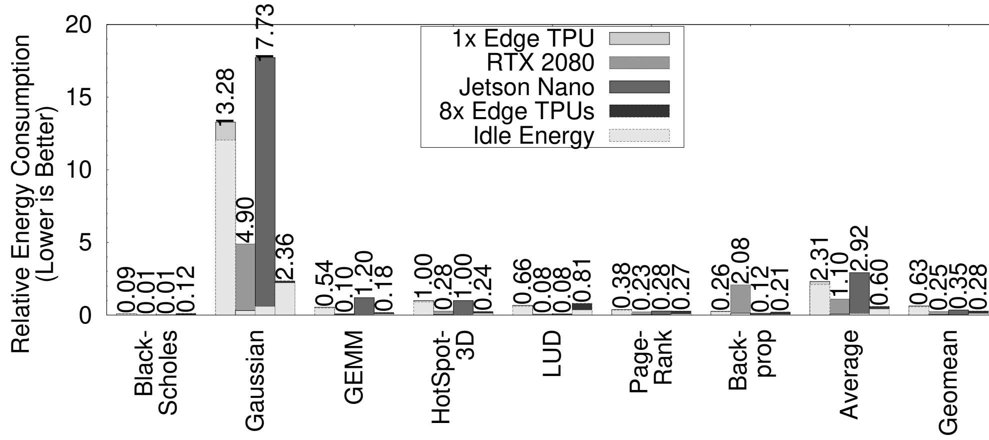
2.9.4 Comparison with GPUs

Because an increasing number of workloads leverage GPU parallelism, we compared the GPTPU to NVIDIA's high-end Turing-architecture-based GTX 2080 and NVIDIA's embedded Jetson Nano platform. Table 2.6 lists the cost and power consumption of evaluated GPUs along with Edge TPUs. Due to the limitation of Jetson Nano's available memory capacity, we have to scale down the input datasets of Blackscholes, Gaussian, GEMM, LUD and PageRank by 25% to 50% to not crash the GPU kernel. Figure 2.9(a) compares the performance for the RTX 2080 and Jetson Nano, using a single Ryzen 3700X CPU core as the baseline, for Rodinia benchmark applications and GEMM using cuBLAS. We enabled RTX-2080's 16-bit ALUs for Gaussian, HotSpot3D, Backprop and Tensor Cores in 8-bit mode for GEMM. The GTX 2080 GPU is 364× faster than a CPU core and 69× faster than the Edge TPU. The embedded GPU on Jetson Nano is still 15× faster than a CPU core and 2.30× faster than an Edge TPU on average. However, with 8× Edge TPUs, the GPTPU can outperform the CPU core by 3.65× and Jetson Nano by 2.48×.

Figure 2.9(b) compares the energy consumption of evaluated platforms. Including idle energy, the 8×-Edge TPU system is the most energy-efficient as the platform can save



(a)



(b)

Figure 2.9: The relative (a) performance and (b) energy for GPTPU with 1× and 8× Edge TPUs vs. the GTX 2080 GPU and Jetson Nano

energy by 40% from the CPU baseline but achieve reasonable speedup. In constrast, the GTX 2080 platform consumes 9% more energy than the CPU baseline. Even though the idle power of the Jetson nano development kit is simply 0.5 W, Jetson nano is still more energy-consuming than GTX 2080 due to the limited speedup.

If we only consider the active power consumption to exclude the factor of various idle power in different system settings, the GTX 2080 consumes 14× the energy of 1× Edge TPU on average, due to the GPU's 195× average active power consumption compared with the Edge TPU, translating to 4.96× worse energy-delay than the baseline. Jetson Nano consumes 23.55× more energy than 1× Edge TPU, making the energy-delay of nano 15.54× worse than 1× Edge TPU. 8×-Edge TPU system consumes just 75% more active energy than 1× Edge TPU, even though the active power consumption is almost 8× of a single Edge TPU. With 8× Edge TPUs, GPTPU offers even better energy-delay (i.e., 46% lower) than the baseline. This result shows that GPTPU offers better energy-efficiency than the current GPU-based solution on embedded/edge platforms.

2.10 Related work

Neural processing units (NPUs) [42, 184] work by using pre-trained models that predicts the outcome of code blocks and map the user program to these models. The GPTPU-based approach is fundamentally different from approaches that rely on the acceleration of approximate programs via NPU in three important ways: (1) GPTPU can accelerate any user-defined algorithm by mapping tensor/matrix operations to supported operators, whereas NPUs can only accelerate a limited set of algorithms that match previ-

ously trained NN models. (2) GPTPU can leverage the Edge TPU microarchitecture and NN hardware to implement exact tensor/matrix operations for applications, whereas NPUs use NNs to produce approximate results for applications. (3) GPTPU can achieve the desired level of precision by iteratively computing on different portions of raw input numbers, whereas NPUs are always limited by the approximate outcomes of NN models.

ASICs can be used like TPUs to accelerate NN applications, as can existing fine-tuned architecture components. Industry data centers [20, 58, 135] take advantage of heterogeneous hardware components by using different processors and reconfigurable hardware for different ML tasks. EFLOPS [36], Richins et. al. [142], and FlexTensor [194] optimize algorithms and task allocations for network traffic in data-center-scale edge computing or single-server computing to reduce infrastructure costs. Language frameworks like Approx-HPVM [147] and ApproxTuner [148] further helps programmer to estimate and optimize the loss of accruacy in ML workloads. The GPTPU framework is orthogonal to the aforementioned research because GPTPU is compatible with existing heterogeneous computing platforms; Edge TPUs can function as complementary hardware accelerators within the system. Ultimately, emerging tensor-processing hardware will inspire the development of related algorithms and associated software [27, 64, 30]. We have seen work extending the application of TPUs to medical image processing [105]. We expect GPTPU can further facilitate this trend. GPTPU can exist in parallel to such future research and potentially extend newly developed algorithms to work in additional application domains.

This paper does not focus on sparse matrices, as many NN accelerators implicitly optimize for sparse matrices. Examples include SCNN [129], SparTen [50], Sparch [191],

Scalpel [185], SIGMA [136], Cambricon-X [189], Bit-Tactical [33], Bit-Pragmatic [2], OuterSPACE [127], Laconic [149], Bit Fusion [150], Sparse Tensor Core [196], PermDNN [34], Park et al. [130], Song et al. [156], and Rhu et al. [141].

2.11 Conclusion

This paper presents GPTPU to bridge the gap between NN accelerators and general-purpose programming. By reverse engineering the commercially available, low-profile NN accelerator, the Google Edge TPU, to uncover important architectural characteristics and the data-exchange protocol, we implement an efficient runtime system, including Tensorizer that dynamically optimizes data layout and instructions, as the GPTPU platform's backend. Using the GPTPU platform and the derived performance numbers, we re-designed the algorithms for a set of important, non-AI/ML related applications. The prototype GPTPU system exhibits a 2.46× speedup over modern high-end CPUs with 40% energy reduction. Though single Edge TPU performance is not yet competitive with high-end GPUs, but the strong scalability of multiple Edge TPUs reveals the potential of future extensions of this line of accelerators. As the demand of ML applications keep growing, we expect manufacturers to keep advancing the microarchitecture of ML accelerators for higher performance and energy-efficiency. GPTPU thus represents an important exploration of general-purpose computing on NN accelerators and is complementary to existing work. The insights presented in this paper will also help extend the range of NN accelerator applications as well as guiding the algorithm design and code optimization for future NN accelerators.

Chapter 3

SHMT: Simultaneous and Heterogenous Multithreading

The landscape of modern computers is undoubtedly heterogeneous, as all computing platforms integrate multiple types of processing units and hardware accelerators. However, the entrenched programming models focus on using only the most efficient processing units for each code region, underutilizing the processing power within heterogeneous computers.

This paper *simultaneous and heterogenous multithreading* (SHMT), a programming and execution model that enables opportunities for "real" parallel processing using heterogeneous processing units. In contrast to conventional models, SHMT can utilize heterogeneous types of processing units concurrently for the same code region. Furthermore, SHMT presents an abstraction and a runtime system to facilitate parallel execution. More importantly, SHMT needs to additionally address the heterogeneity in data precision that

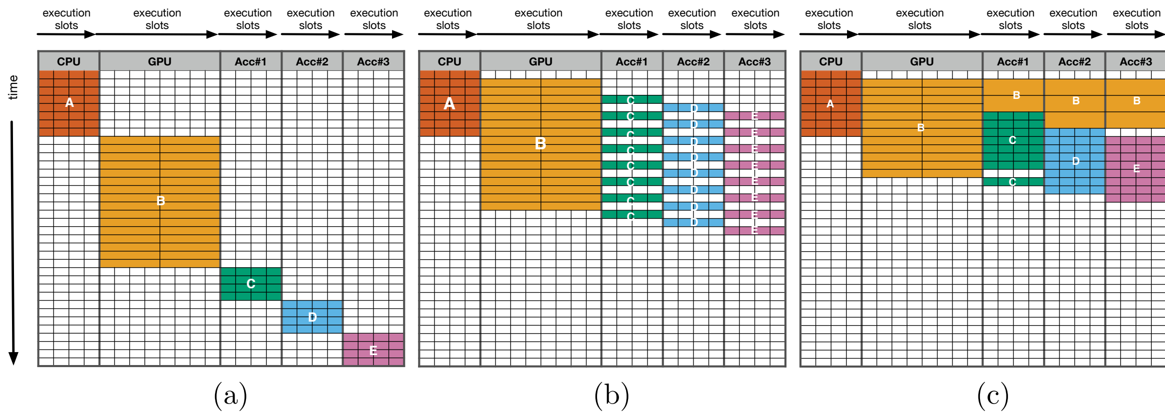


Figure 3.1: The execution model of (a) conventional heterogeneous computers (b) conventional heterogeneous computers with software pipelining, and (c) SHMT.

various processing units support to ensure the quality of the result.

This paper implements and evaluates SHMT on an embedded system platform with a GPU and an Edge TPU. SHMT achieves up to 1.95× speedup and 51.0% energy reduction compared to GPU baseline.

3.1 Introduction

The integration of graphics processing units (GPUs) and hardware accelerators for artificial intelligence (AI) and machine learning (ML) or Digital Signal Processing (DSPs) brings heterogeneous computing models into all types of modern computers, ranging from wearable devices, mobile phones, and personal computers to data center servers. Famous, commercialized examples include Tensor Cores (TCs)[123, 122] or Ray Tracing Cores (RT Cores)[18] on NVIDIA GPUs, Tensor Processing Units (TPUs) on Google Cloud servers [81,

80, 78], Neural Engines on Apple's iPhones [11], Edge Tensor Processing Units (Edge TPUs) on Google Pixel Phones. Through implementing more efficient architectures processing models for target applications domains, heterogeneous computing resources help address the issue that general-purpose CPUs alone can not afford the desired performance for modern workloads, including artificial intelligence (AI), machine learning (ML), reality, or gaming applications.

Recent research projects have proved that many co-processors and hardware accelerators can perform the same functions at similar orders of magnitude [65, 30, 69, 104, 103, 67, 95, 39, 42], despite their differences in processing models and design agendas. Theoretically, the system can simultaneously use these heterogeneous processors to maximize throughputs and minimize latency and energy consumption. However, conventional programming frameworks, including domain-specific languages, can only delegate a code region exclusively to one kind of processor, leaving other computing resources idle without contributing to the current function [1, 121, 85].

This paper presents SHMT, simultaneous and heterogeneous multithreading, to evaluate the performance and tackle the challenges of simultaneously using heterogeneous computing resources of a heterogeneous computer system. Unlike conventional programming and execution models that focus on using the most efficient computing resources and exploiting homogeneous parallelism within the identified type of computing resources for each function, SHMT can break up the computation from the same function to multiple types of computing resources and exploits heterogeneous types of parallelism in the meantime.

Figure 3.1 illustrates the advantage of SHMT against the conventional execution model. Figure 3.1 assumes a program containing five primary functions, A to E, and five computing resources, including CPUs, GPUs, and three accelerators. Figure 3.1(a) presents the execution flow in conventional programming models that delegate the function to the most efficient processing units. Though conventional models can exploit parallelism within the same type of processors, conventional models still let other resources idle or make no progress to the current program. The program seems to use multiple types of hardware concurrently through programming techniques like software pipelining. Figure 3.1(b) assumes the program can progress with partial results and pipeline the execution of different functions on different hardware units. However, as each function takes a different amount of time to generate partial results, the imbalance of execution can still lead to waste. SHMT, as Figure 3.1(c) depicts, allows function B to use GPUs and other accelerators. As a result, SHMT can significantly improve hardware utilization and lead to better end-to-end latency and energy consumption.

Enabling SHMT is challenging in the following aspects. First, as heterogeneous computing resources use diverse programming models (e.g., vector processing in GPUs and matrix processing in Tensor Cores), SHMT must present some mechanism that can describe and divide equivalent operations and data on different computing resources. Second, unlike traditional programming systems that delegate each code region to one type of hardware, SHMT must be able to coordinate the execution on heterogeneous hardware efficiently. Finally, as various hardware units deliver results at different levels of quality, SHMT must assure the outcome without incurring significant overhead.

The SHMT framework proposed three components to address the challenges above. First, SHMT promotes a set of virtual operations (VOPs) and High-Level Operations (HLOPs) as an intermediate between programming languages and hardware instructions/operations to facilitate task matching and distribution. Second, SHMT presents a runtime system that dynamically adjusts the workloads on various hardware units to maximize hardware efficiency while allowing flexibility in scheduling policies. Finally, SHMT presents a low-overhead scheduling policy that considers both results and performance.

This paper develops the proposed SHMT framework on an embedded system platform equipped with a multi-core ARM processor, an NVIDIA GPU, and an Edge TPU. SHMT achieves up to 3.92× speedup and 2.07× on average. With our proposed quality assurance mechanisms, SHMT still achieves 1.95× speedup on average. SHMT also reduces energy consumption by 51%.

In presenting SHMT, this paper makes the following contributions.

- SHMT presents a new parallel programming and execution model that distinguishes itself from prior work as SHMT uses heterogeneous hardware concurrently to accomplish parallel tasks from the same piece of code.

- SHMT evaluates and demonstrates the potential of leveraging hardware using heterogeneous programming models using a real system platform.

- SHMT presents an abstraction and mechanisms to coordinate concurrent execution on heterogeneous hardware components.

- SHMT proposes a low-overhead mechanism and scheduling policy to ensure the quality of results.

3.2 Background and motivation

In modern heterogeneous computers, two technology trends make sense of SHMT: first, the ubiquitous adoption of hardware accelerators. Second, the abilities of hardware accelerators to applications beyond their original target domains. However, before SHMT, no existing work tried to have multiple types of accelerators collaborate on the same code region. This section describes the technology trends and the potential of SHMT.

3.2.1 Modern heterogeneous components

As Dennard scaling slows, the integration of domain-specific hardware accelerators becomes universal. Most computer systems nowadays contain the following domain-specific hardware accelerators.

Graphics processing unit (GPU) Despite the broad spectrum of applications, GPUs are initially accelerators for computer graphics. The nature of pixel rendering algorithms makes vector processing architecture using the single instruction multiple data (SIMD) paradigm the best fit for the target domain. Modern GPU architectures natively support computation in single precision (FP32) but also provide half-precision (FP16) [62] for AI/ML applications.

AI/ML accelerators AI/ML accelerators have become popular in all types of computer systems to tackle the rapidly growing demand for AI/ML workloads and offer better energy efficiency and offloading CPUs/GPUs for other workloads. As modern AI/ML models intensively use matrix algebra, most AI/ML accelerators tailor their internal architectures with circuits specialized for matrix operations. Google's Edge TPUs, data center TPUs,

and NVIDIA's Tensor Cores [123, 122] are all hardware implementations of frequently used matrix operations in AI/ML workloads.

Most AI/ML applications are error-tolerant. As a result, the hardware design can further improve performance, power consumption, and area-efficiency through approximate computing and reduce data precisions. The early version of Edge TPUs supports only INT8 precision support and thus can deliver more compelling performance per Watt than the data center TPUs (2 TOPS/W v.s 0.36 TOPS/W for Cloud TPUs). NVIDIA's tensor cores only natively support half-precision and Bfloat16 (BF16).

Other accelerators Computer systems have a long history of adopting digital signal processors (DSPs) back in the 1970s. DSPs have again become popular as strong demands in high-bandwidth communication, teleconferencing, media streaming, and creating visual and audio inputs/outputs for AI/ML applications. The hardware logic may implement mathematical operations to support Fast Fourier transforms (FFTs) or finite impulse response (FIR) filters. As image data contain three bands of 8-byte color descriptions, most image DSPs only support computation in 24-bit [124, 8]. Google Visual Core's Image Processing Unit implements stencil operations in 16-bit. However, as many DSP applications have strong connections with AI/ML applications and rely on similar mathematical functions, SHMT can easily extend the support to DSPs. Ray Tracing is another emerging type of accelerator that simulates the behavior of lights in the real world to fulfill the demand for virtual reality and gaming applications. Modern ray-tracing cores implement logics for bounding volume hierarchy (BVH) tree traversal [18].

3.2.2 Generalization of domain-specific accelerators

Broadening the application of domain-specific accelerators has two different approaches. First, use the mathematical functions in DSAs to perform the equivalent operation in an out-of-domain application. The other approach is to reduce the out-of-domain problem as a problem inside the accelerator's target domain. This section will introduce the recent advances in both directions on emerging hardware accelerators besides GPUs.

Using mathematical functions in DSAs

As most hardware accelerators are accelerators for key mathematical operators, the programmer can change the program implementations to invoke an accelerator's hardware operations directly. This approach typically relies on support from appropriate hardware/software interfaces and general-purpose programming frameworks. Examples include CUDA and OpenCL, which promote general-purpose computing on GPUs (GPGPUs). In the context of modern AI/ML accelerators, NVIDIA exposes the MMA instruction support in Tensor Cores through the **wmma** interface and cuBLAS library functions. Recent research projects, including TCUSCAN [30], TCUDB [69], and RQTPU [65] demonstrate the use of matrix multiplications on Tensor Cores to accelerate database query operations like reduction, scan, and join. Besides AI/ML workloads, Google also demonstrates matrix multiplication in TPUs to accelerate Fourier Transform [104, 39] and facilitate MRI image reconstruction [103]. GPTPU [67] reverse-engineered the Edge TPU compiler and built a tensor operator-based programming framework for Edge TPU to accelerate Rodinia benchmark applications [22].

Reducing the original problem to the accelerator's target domain

The other approach to using domain-specific accelerators is to reduce the problem as one in the accelerator's target domain. In contrast to the method in Section 3.2.2, this approach requires less programming language or ISA support in exposing the internal hardware features to programmers.

Neural Processing Units (NPUs) [42, 5, 110, 101] follow this route to solve general-purpose problems using NN accelerators. NPUs leverage universal the approximation theorem [29] in approximating any given problem/algorithm as an NN model, and thus the process of solving the original problem becomes an instance of NN inference. In this paper, we intensively used NPUs as our solutions for Edge TPU implementations, as implementing the concept of NPUs can make more efficient use of AI/ML accelerator hardware. RTNN [198] also follows the same direction but with RT Cores as the target domain-specific accelerator. RTNN formulates the tree-based neighbor search algorithms on the BVH tree, thus enabling the BVH traversal function on RT Cores.

3.2.3 Potential and challenges of SHMT

With existing efforts of general-purpose computing on hardware accelerators, multiple types of accelerators can perform the same function with compelling performance. Figure 3.2 compares the performance of running the core kernel function in ten applications using their NPU implementations on Edge TPU against their state-of-the-art GPU implementations on the GPU of Jetson Nano. If we offload all kernels to Edge TPU, the performance is 5% slower than GPUs on average. The average theoretical speedup from

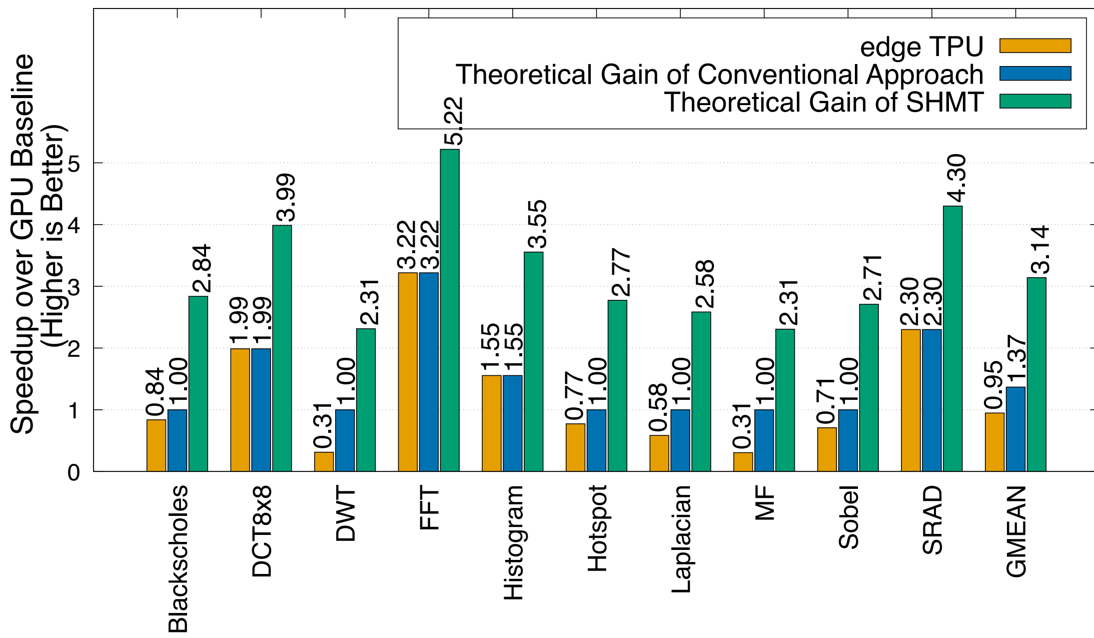


Figure 3.2: The potential speedup of SHMT (and Edge TPU) relative to GPU-only implementation

conventional approaches that delegate kernels to the best-performing accelerator is 1.37×. Using the performance number we gathered from running experiments using GPUs or Edge TPUs, we derived the theoretical performance gain of SHMT and presented the numbers in Figure 3.2. By carefully finding the optimal planning of using GPUs and Edge TPU simultaneously to share the computation from the same application kernel and ignoring all data exchange/transformation overhead, the average speedup is 3.14×.

However, a system must tackle the following challenges to enable the simultaneous use of multiple types of hardware accelerators in accomplishing the computation for a compute kernel. First, as each hardware accelerator has its unique programming interface and execution model, without appropriate system supports, the programmer needs to figure out the equivalent set of operations on various accelerators and manually create multiple threads that map each partition of computation to different hardware and handle the data exchange/synchronization. Second, as the microarchitecture and execution model of each hardware accelerator differs, the relative performance ratio and data exchange overhead among hardware accelerators change as data sizes or system dynamics change. Therefore, even if the programmer can partition computation to simultaneous threads working on different data partitions, the resulting program is not always optimal for the underlying hardware or cannot guarantee speedup. Finally, unlike homogeneous hardware components that accept data in the same representation and deliver the result with the same accuracy, heterogeneous hardware components accept data and deliver results in different formats and accuracies. As a result, carelessly using heterogeneous hardware components simultaneously can lead to unwanted execution results.

3.3 SHMT

In response to the challenges of supporting SHMT, we developed a system architecture consisting of three main components. First, SHMT defines an extensible set of hardware-independent virtual operations (VOPs) that allows heterogeneous hardware to interact with SHMT software as an intermediate. Second, an SHMT runtime system that performs the low overhead task scheduling to manipulate the use of heterogeneous hardware. And finally, runtime mechanisms to ensure the quality of results. This section will overview the proposed framework and present our proposed policies and mechanisms in each component.

3.3.1 Overview

Figure 3.3 presents the overview of SHMT. SHMT abstracts its subsystem as a virtual hardware computing resource offering a rich set of virtual operations (VOPs) that allows a CPU program to "offload" computation to this virtual hardware device. The compiler or the programmer can use VOPs to describe the desired computation for SHMT. As the adoption of domain-specific languages (e.g., Tensorflow or PyTorch) using standard libraries and accelerated libraries (e.g., cuBLAS, cuDNN) in modern programming languages, we expect the frontend authoring languages of user programs to remain the same. Most changes should only occur at the library level.

During the program execution, the runtime system, which acts as the "driver" of SHMT's virtual hardware, dynamically parses the VOPs and gauges the ability of hardware resources to make scheduling decisions. The runtime system divides a VOP into one or more

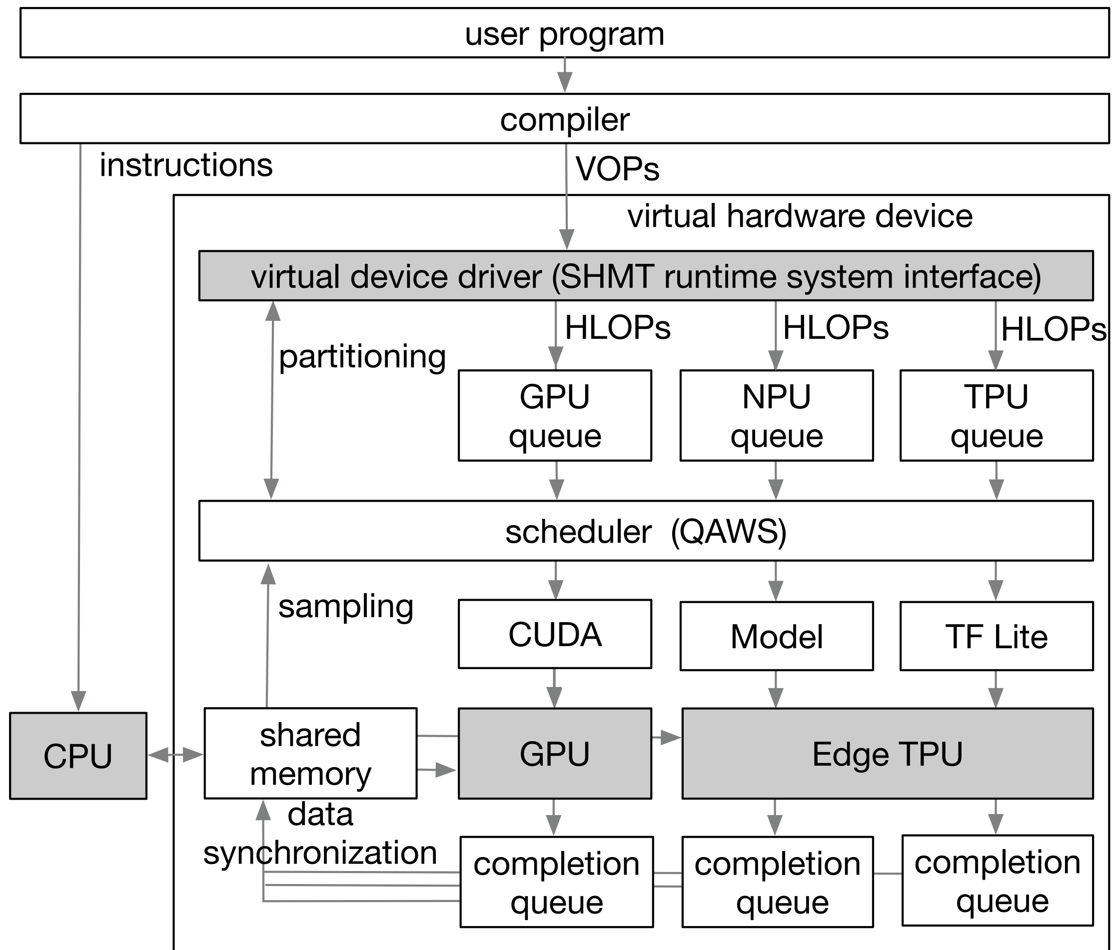


Figure 3.3: SHMT overview

high-level operations (HLOPs) to simultaneously use multiple hardware resources. Each HLOP is a basic scheduling identity in SHMT and performs a partition of computation for a VOP. The implementation of each HLOP typically maps to a set of hardware operations and functions on the target hardware resource. Finally, the runtime system assigns these HLOPs to the task queues of the target hardware. As HLOPs are also hardware-independent, the runtime system can still adjust the task assignment if necessary.

As VOPs and HLOPs provide flexibility in scheduling, SHMT's runtime system can easily integrate scheduling policies to improve performance. This paper presents a quality-aware work-stealing (QAWS) scheduling policy that has low execution overhead but helps to maintain quality and balance the workload.

Figure 3.4 provides an overview from the programmer's perspective. We envision the programming interface for general application programmers to remain the same. The application programmer can still use domain-specific function calls or library functions at a high level. In Figure 3.4, the application programmer invokes the general matrix multiplication (GEMM) functions that TensorFlow provides (i.e., `tf.matmul`). Most application programmers will be unaware of the following change at the language runtime level: the TensorFlow implementation of `tf.matmul` calls the `shmt:: matmul()` function that SHMT provides to the system programmer to invoke the VOP of GEMM. The SHMT internal implementation of `shmt::matmul()` will then analyze and decompose the GEMM VOP into HLOPs, where each HLOP is a native implementation of a chunk of GEMM computation on the dedicated hardware resource.

Figure 3.4 presents the programming model of SHMT. In summary, we have limited the programming efforts as we tried to present an almost identical programming interface to most programmers. The implementation of HLOPs also leverages existing support without burdening most system engineers.

3.3.2 Virtual Operations (VOPs) and High-Level Operations (HLOPs)

SHMT tackles the challenge of the heterogeneity from execution models and data formats using VOPs and HLOPs. VOPs define the available computation that SHMT can provide to the program and HLOPs define available operations in the underlying hardware that SHMT can leverage. In SHMT model, HLOPs without data dependency can execute simultaneously, regardless of the actual hardware performing the computation.

Virtual operations (VOPs)

VOPs in SHMT is a set of definitions describing available operations that SHMT's underlying hardware can support. VOPs help to abstract the whole SHMT subsystem as a single but powerful accelerator from the software's perspective. The SHMT subsystem is a big umbrella covering all computing resources that SHMT can use to exercise sub-tasks from VOPs simultaneously.

Table 3.1 lists the VOPs that our prototyping SHMT system supports. As SHMT focuses on the simultaneous use of multiple types of computing resources, our current list covers the most frequently implemented supported computation in hardware accelerators. In our current list, these VOPs can either use an element-wise vector processing model or

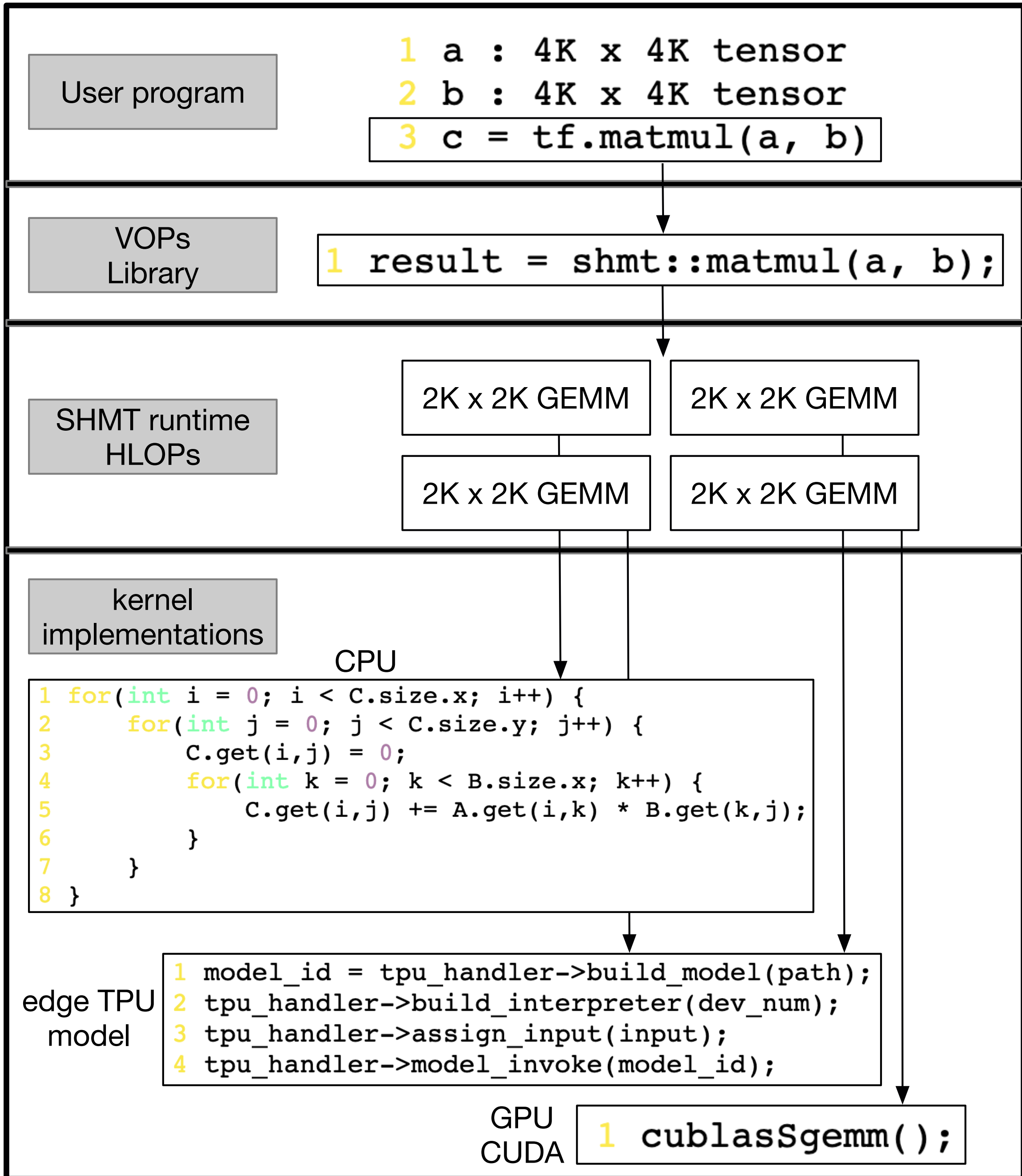


Figure 3.4: The SHMT's programming model

vector		tiling
add	reduce_sum	conv
log	relu	DCT8x8
max	rsqrt	FDWT97
min	sqrt	FFT
multiply	sub	GEMM
parabolic_PDE	tanh	Laplaican
reduce_average		Mean_Filter
reduce_hist256		Sobel
reduce_max		SRAD
reduce_min		stencil

Table 3.1: The VOPs list in either vector or matrix tiling processing model types.

a tile-wise matrix processing model to partition and parallelize the computation without violating the correctness.

High-level operations (HLOPs)

An HLOP in SHMT defines a subset of a VOP operation that an underlying hardware computing device can support. An HLOP shares the same opcode as the supporting VOP. However, unlike a VOP with no assumption/restriction on the input/output data sizes, an HLOP defines the data sizes/granularities and the data types a hardware device can support. For each VOP, SHMT's runtime system will dynamically partition computation tasks and data into HLOPs and assign each HLOP to an underlying hardware device using the data sizes and the parallelization model.

If the target device provides native support for an HLOP, the HLOP's implementation for such devices can directly invoke the hardware command. For example, as edge TPU implements convolution 2D in hardware, the edge TPU's HLOP implementation simply invokes the corresponding hardware function. Otherwise, the HLOP can still use multiple

hardware operations to accomplish the desired computation on optimal data sizes. For example, the convolution 2D implementation on a GPU will internally become a series of vector operations within the HLOP implementation. For NPUs, the implementation makes an inference through a pre-trained model that approximates the result of convolution 2D.

3.3.3 SHMT's runtime system

In actual system implementation, the SHMT's runtime system is a kernel driver of a virtual device. The virtual device driver accepts VOPs as a subset of its commands and partitions VOPs into HLOPs on the target hardware devices. SHMT's runtime system also provides interfaces for more advanced scheduling policies. SHMT's kernel driver maintains a pair of queues for each SHMT-compatible hardware resource; one serves as the incoming queue and the other as the completion queue. Upon the initialization of the SHMT system, each hardware resource's driver is responsible for providing SHMT with its list of available HLOPs operations and their implementations.

HLOP distribution

For each VOP that SHMT receives, the runtime system figures out available hardware resources to perform the VOP, gathers the information regarding the parallelization method and data partitioning, and consults the scheduler for the task mapping on hardware resources. If the scheduler suggests a plan, the runtime system realizes the plan by partitioning the VOP into HLOPs on devices at supported data sizes. As SHMT supports a limited number of parallelization models, the runtime system can apply the template for each parallelization model for dataset partition, aggregation, and synchronization. SHMT

assigns an HLOP to the target device by sending the HLOP to the device's incoming queue. A thread monitoring the queue will work with the target device's kernel module and execute the HLOP implementation whenever the device is available. Once the HLOP finishes, the thread will move the task to a completion queue that SHMT runtime system can later dequeue and use for data aggregation and synchronization purposes.

Data distribution and transformation

In modern heterogeneous systems, hardware accelerators are typically separated intellectual property cores or chips that communicate with the main CPU cores through the system interconnect. Like the idea of processor caches, most hardware accelerators also own their private device memory to facilitate the execution of operations. As each device's HLOP accepts fix-sized, fix-shaped data, SHMT's runtime system creates memory operations using similar arguments as the implementation of CUDA's `cudaMemcpy2D` that takes the starting address of the source data structure and use the element size, dimensions of each input partition to calculate the effective addresses of source and target data locations that each HLOP uses. The runtime system will schedule the data movement using the effective addresses between the system's shared main memory and each device's memory location after assigning an HLOP.

Most hardware accelerators optimize their computation models and architectures for targeted application domains, thus supporting limited data precisions. Suppose the scheduling policy determines the use of a target hardware resource as appropriate despite the potential loss of accuracy. In that case, the runtime system will perform data type casting through the desired quantization method before distributing the input data. When

the device finishes computation, the runtime system is again responsible for restoring the result to the data precision that the application desires.

3.3.4 The basic work-stealing scheduler

Work-stealing is the basic scheduling policy that SHMT uses as the policy best balances the workload among scheduling targets with various performances. The scheduler makes an initial plan by partitioning datasets evenly based on the parallelization model of the scheduling VOP and assigning each data partition as well as the computation associated with that partition to a target computing resource. The runtime system will generate and enqueue the HLOPs corresponding to the computation for each partition to the target hardware's incoming queue. When an HLOP completes, the runtime system also reports to the scheduler.

As heterogeneous computing systems share and synchronize data at the system's main memory level, each input and output data partition should be larger than and be multiples of the main memory page size whenever possible. For example, using the most frequently used 4KB page size, each partition of floating-point data inputs in the vector processing model should contain at least 1,024 consecutive elements, and a matrix tile should be at least 1,024 $\times$1,024 sized. Partitioning data at larger than page-sized granularities can make more efficient use of memory bandwidth and avoid redundant page accesses and write amplification issues.

When the workload is imbalanced, that is, the incoming queue of a hardware device has more pending items than others, the scheduler informs the runtime system to withdraw HLOPs associated with unprocessed data partitions from the current assignment

and reassign the HLOPs to the hardware with the most empty queue. The granularities can mismatch between different devices, so the runtime system may need to further fuse or partition HLOPs.

3.3.5 Quality-Aware Work-Stealing (QAWS) policy

This paper proposes an exemplary quality-aware work-stealing (QAWS) scheduling policy to demonstrate the effect of a first-level quality control mechanism in the SHMT scheduler and the flexibility of SHMT in changing scheduling policies. As the microarchitecture of application-specific hardware accelerators aims to provide just enough result quality for the target workloads, most hardware accelerators, especially those targeting AI/ML workloads, do not support the precision modes for exact, general-purpose computing. Without any quality control mechanism, naively using hardware accelerators as other general-purpose processors can lead to unwanted computation results.

The design of QAWS aims at ensuring the results' quality of critical data regions while maintaining low computation overhead in scheduling. For each input data partition, QAWS samples the data to determine the criticality and assigns computation to a device accordingly. We leverage the experience from prior works that consider critical regions as data partitions with the widest value distributions. In this paper, we examined two policies using sampled criticalities.

1. **Device-dependent limits** This policy determines the scheduling on a device using device-dependent limits. Each computing device has a set of acceptable hardware

Algorithm 1 Device Limitation

Input: P, limits

1: $N \leftarrow |\mathbf{P}|$
2: $M \leftarrow |\mathbf{limits}|$
3: $|\mathbf{Q}| \leftarrow N$
4: **for** $i \leftarrow 0$ to N **do**
5: $\quad s \leftarrow sampling_module(\mathbf{P}_i)$
6: $\quad \mathbf{Q}_i \leftarrow M - 1$ ▷ your default choice
7: $\quad$ **for** $j \leftarrow 0$ to M **do**
8: $\quad\quad$ **if** $s < \mathbf{limits}_j[0]$ **then**
9: $\quad\quad\quad \mathbf{Q}_i \leftarrow \mathbf{limits}_j[1]$
10: $\quad\quad\quad$ *break*
11: **return Q**

limits based on the supporting data precision and accuracy. QAWS assigns only data inputs lower than the criticality limits to that computing resource. In the case of work stealing, QAWS only allows a device to steal HLOPs from another device with the same or a lower hardware limit.

2. **Application-dependent top−K% criticality** This policy ranks the criticality within a window of data partitions and schedules top-K% partitions to the most accurate device, second-L% to the second-most accurate device, and so on. The threshold values of K and L are application-dependent. The programmer or the library composer should provide, along with each VOP, indicating the percentage of data inputs that

Algorithm 2 Top-K Criticality

Input: $\mathbf{P}$, K, W

1: $N \leftarrow |\mathbf{P}|$

2: $|\mathbf{Q}| \leftarrow N$

3: $|window[]| \leftarrow W$

4: **for** $i \leftarrow 0$ to N **do**

5: $\quad window[i\%W] \leftarrow sampling_module(\mathbf{P}_i)$

6: $\quad$ **if** $(i\%W == W - 1)$ *or* $(i == N - 1)$ **then**

7: $\quad\quad sort(window)$

8: $\quad\quad$ **for** $j \leftarrow 0$ to W **do**

9: $\quad\quad\quad \mathbf{Q}_i \leftarrow (j < K)\ ?\ 0 : 1$

10: **return Q**

are generally critical to results in this library function or the application. In the case of work stealing, QAWS only allows a device with higher accuracy to steal HLOPs from another device with the same or a lower accuracy.

Algorithm 1 and Algorithm 2 explain the algorithmic details of how QAWS assigns computation to a device for two options: (1) device-dependent limitation and (2) application-dependent top-K criticality, respectively. In Algorithm 1, **P** is an array of input partitions, and **limits** is an array of paired numbers - the limitation number of a device and the index of the corresponding device queue. **limits** is sorted by the first index in descending order. In Algorithm 2, the two additional inputs other than **P**, K, and W, are the threshold value of top-K and window size W, respectively. Any given K has to be

Algorithm 3 The striding sampling

Input: $\mathbf{D}$, N, s

1: $|\mathbf{S}| \leftarrow N$

2: **for** $i \leftarrow 0$ to N **do**

3: $\quad \mathbf{S}_i \leftarrow \mathbf{D}[i * s]$

4: **return S**

smaller than the W. And the result array **Q** from each algorithm is an array of queues' index numbers each HLOP assigns to. For example, in the case of only GPU and Edge TPU queues present in a SHMT system, the GPU queue has an index value of 0, and the Edge TPU queue has an index value of 1.

The mechanism that SHMT uses to determine the criticality leverages the insight of canary input from the input responsiveness approximation (IRA) technique [93]. IRA technique proposes and proves that the computation result using canary input, a small set of input data, can effectively approximate the overall computation quality. However, the complete IRA technique requires actual computations on canary inputs that incur significant performance overhead at the scheduler's level. Therefore, SHMT only performs the input evaluation from IRA and determines the criticality of an input data partition using two metrics, data range (i.e., maximum and minimum values) and standard deviation within the region.

As faithfully scanning through the input region increases the computation overhead, SHMT proposes sampling. We examined three different sampling mechanisms in this

Algorithm 4 The uniform random sampling

Input: $\mathbf{D}$, N

1: $|\mathbf{S}| \leftarrow N$

2: **for** $i \leftarrow 0$ to N **do**

3: $\mathbf{S}_i \leftarrow \mathbf{D}[random()]$

4: **return S**

Algorithm 5 The reduction sampling

Input: $\mathbf{D}$, s

1: $\mathbf{S} \leftarrow []$

2: $\mathbf{dims} \leftarrow dimension(\mathbf{D})$

3: **for** $i0 \leftarrow \mathbf{dims}_0$ with step size s **do**

4: **for** $i1 \leftarrow \mathbf{dims}_1$ with step size s **do**

5: ...

6: $\mathbf{S}.append(\mathbf{D}[i0, i1, ...])$

7: **return S**

paper. Algorithms 3, 4, and 5 summarize the three sampling methods - striding, random, and reduction - QAWS uses, respectively. The $\mathbf{D}$ of all options is the input data partition, the s for Algorithms 3 and 5 is a step size, and the N for Algorithms 3 and 4 is the desired number of samplings.

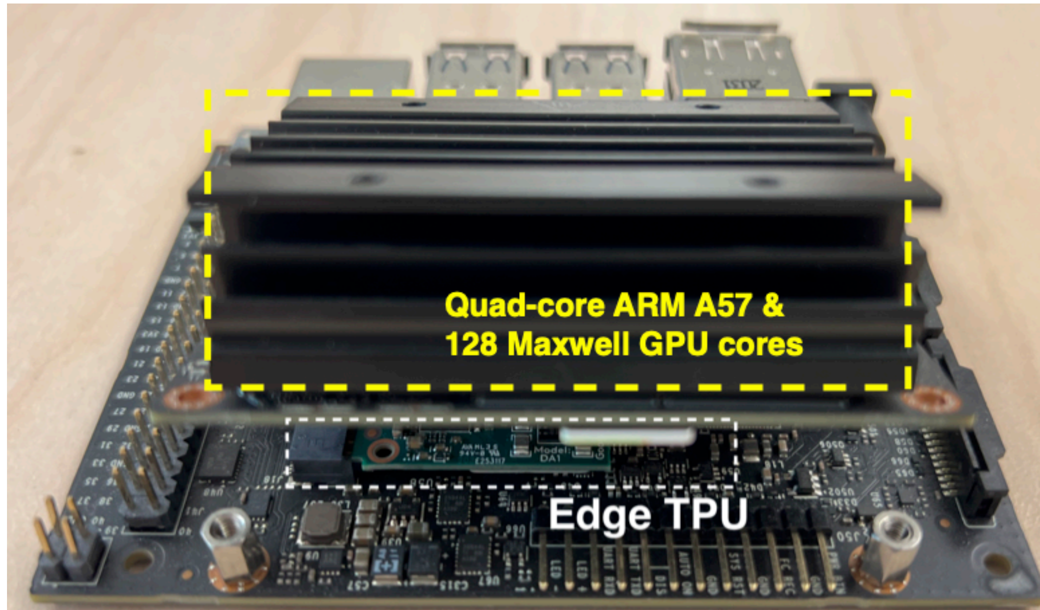


Figure 3.5: The SHMT prototype platform

3.4 The SHMT system prototype

This paper evaluates SHMT using a custom-built prototype with real hardware components and applications. This section describes the hardware/software system architecture and the method of incorporating NPUs into this platform.

3.4.1 The system assembly

This paper built an exemplary SHMT prototype using NVIDIA's Jetson Nano and Google's Edge TPU. Figure 3.5 shows the photo of the assembled system. The Jetson Nano module contains a quad-core ARM A57 processor and 128 Maxwell GPU cores. We connect an Edge TPU to the system via the m.2 slot on the back of the Jetson Nano processor/GPU module. The three types of processing units, CPU, GPU, and Edge TPU exchange data through the on-board PCIe interface. The prototype system contains 4 GB

64-bit LPDDR4 interface at 25.6 GB/s as the main memory. The system's main memory hosts the share data among CPU, GPU, and Edge TPU. Edge TPU additionally contains 8 MB device memory. The system assembly runs an Ubuntu Linux 18.04 with NVIDIA's customized 4.9.253-tegra kernel. We implemented the virtual SHMT hardware device as a dynamically loadable kernel module.

We built the prototype using selected components and believe that this prototype is representative of most use cases for the following reasons. First, the processing power and the available types of processors and accelerators of this system platform resemble the hardware components that modern smartphone or mobile devices [54], allowing this platform to assess the real performance of using SHMT on these scenarios. Second, the ratio of computing power between Maxwell GPUs and Edge TPUs (472 GLOPS v.s. 4 TOPS) resembles those on data center servers (67 TFLOPS FP32 of A100 and 275 TFLOPS of TPUv4) [123, 79], allowing this platform to assess the relative performance of SHMT on cloud servers. Finally, the most important reason is the availability of the hardware components and customizing the software stack. As SHMT requires changes in kernel modules, the evaluation platform must allow full control for experimental purposes. However, Google only provides access to data center TPUs through their cloud platforms without permitting the creation of customized system modules. We can only use the commercially available Edge TPUs to build the prototype platforms. Building SHMT using widely available hardware components will also enable broader applications to the proposed framework.

3.4.2 NPU implementations

Edge TPU can either serve as a matrix function accelerator as Section 3.2.2 describes or implement an NPU as Section 3.2.2 describes. In the case of using Edge TPU as matrix accelerators, we leverage existing library to implement corresponding HLOPs [67].

Edge TPU can naturally implement the concept of NPU as the target application of Edge TPU is inferencing NN models. Each HLOP of Edge TPU using NPU mode is a pre-trained model for the HLOP. Based on the microarchitecture of Edge TPUs, these HLOP-NN models should (1) use multilayer perceptrons (MLPs) with convolution and dense operators and *sigmoid* or *relu* as activation functions and (2) be the first found and the simplest topology whenever the learning curve of a full precision TensorFlow model training significantly improves throughout topology searching. We use the following steps to construct an NPU model on Edge TPU.

1. Construct the training and validation datasets by running the target algorithm/function using high-performance CPU/GPU platforms with randomly-generated input data and collecting the output.

2. Train the NPU-HLOP model using high-performance CPU/GPU platforms.

3. Perform post-training quantization for the trained model into an Edge TPU-compatible model using TensorFlow Lite (TFLite) and *edgetpu_compiler* [53].

Benchmark	Category	Baseline Implementation
Blackscholes	Finance	CUDA Examples [121]
DCT8x8	Image Processing	CUDA Examples [121]
DWT	Signal Processing	Rodinia 3.1 [22]
FFT	Signal Processing	CUDA Examples [121]
Histogram	Statistical	Opencv 4.5.5 [16]
Hotspot	Physics Simulation	Rodinia 3.1 [22]
Laplacian	Image Processing	Opencv 4.5.5 [16]
Mean Filter(MF)	Image Processing	Opencv 4.5.5 [16]
Sobel	Image Processing	Opencv 4.5.5 [16]
SRAD	Medical Imaging	CUDA Examples [121]

Table 3.2: Table of benchmarks

4. Test the Edge TPU-compatible model with validation dataset again. If the Edge TPU-compatible model's accuracy is significantly lower than its version on the high-performance platform, we will enable quantization-aware training mode to re-train the model with weights in 8-bit precisions.

3.5 Results

SHMT with QAWS achieves 1.95× speedup compared with the case where we can only offload computation to the fastest accelerator. As SHMT leverages low-power hardware accelerators to assist the program execution along with GPUs, SHMT reduces the energy consumption by 51.0%. This section describes the speedup, quality, and energy consumption of SHMT when running various applications using the prototype Section 3.4 presents.

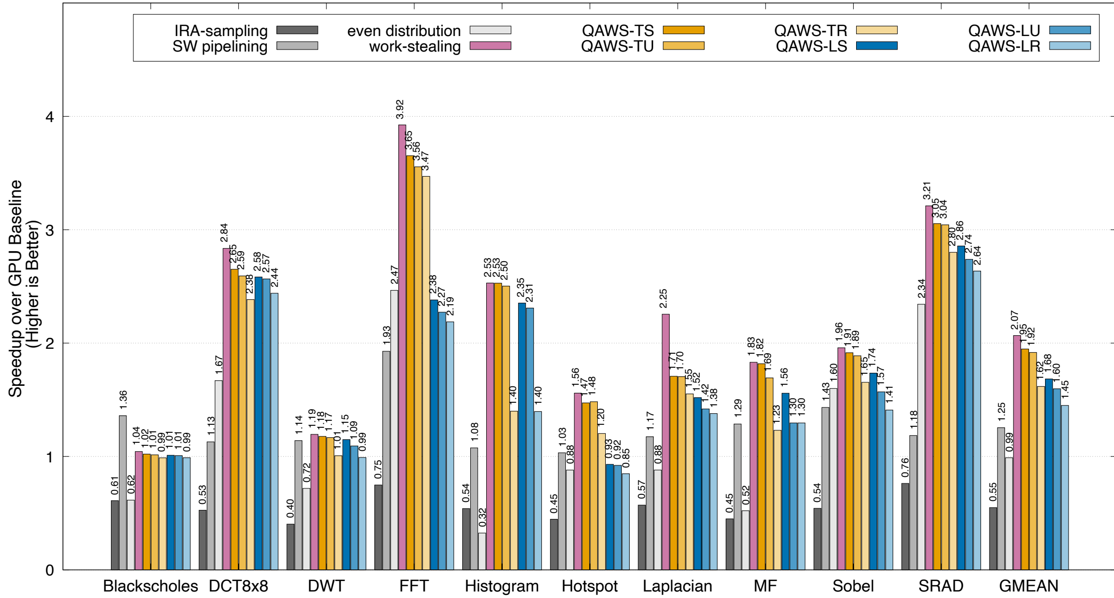


Figure 3.6: The application speedup relative to the baseline GPU implementations

3.5.1 Benchmark applications

Table 3.2 lists the benchmark applications this paper uses to evaluate SHMT and the sources of their baseline GPU implementations. We select these applications as these applications have both high-performance GPU and NPU implementations that we can gather from public code repositories through our best-effort search. In addition, these applications cover multiple application domains, including image processing, signal processing, physics simulation, medical imaging, and finance. Without otherwise mentioned, the default input data size for each benchmark contains 8192×8192 randomly generated floating-point numbers.

3.5.2 Speedup of end-to-end latency

Comparing the end-to-end latency of SHMT with optimized baseline GPU implementations, SHMT with the best-performing QAWS policy achieves 1.95× speedup. Figure 3.6 illustrates the speedup of SHMT with various scheduling policies. In Figure 3.6 and the following sections, we denote the variation of QAWS results as *QAWS-XY* where ***X*** stands for hardware assignment policies using (1) Device **L**imitation or (2)**T**op-K methods, and **Y** stands for the sampling method, either (1)**S**tridding, (2)**U**niform random sampling or (3)**R**eduction.

We also include two policies that do not consider the quality of results, the even distribution, and the work-stealing, as references. Naively distributing HLOPs evenly between the GPU and the Edge TPU would make the performance bounded by the slower hardware and result in performance loss in 6 out of ten benchmark applications where Edge TPU's implementations are slower in our case. In contrast, work-stealing can achieve 2.07× speedup on average as work-stealing adjusts the workloads based on the consumption rate of HLOPs, allowing faster hardware to perform more HLOPs and slower hardware as an auxiliary device supporting the parallel execution with the least amount of idleness. The performance work-stealing policy also represents the optimal speedup of SHMT without considering result qualities.

All QAWS policies in this paper sample and adjust workload distributions on top of the basis of work-stealing. The speedup that Figure 3.6 reports for each policy already includes the sampling overhead. Among all SHMT policies with quality control mechanisms, *QAWS-TS* performs the best and achieves 1.95× speedup compared with the

GPU baseline on average. *QAWS-TU* seconds at 1.92× average speedup. Compared with the two policies *QAWS-LU* and *QAWS-LS* that use the same sampling mechanism with initial queue assignment policy using device limitations, the performance of *QAWS-TS* and *QAWS-TU* reveals that "Top-K" is more suitable for performance-critical workloads. Compared with device limitations, the rank-based approach in Top-K may increase the amount of data partitions that Edge TPU can perform in our platform since Edge TPU can still work on some data partitions with wider value ranges or variances than its hardware limitation. Regardless of using Top-K or device limitations, reduction performs the worst due to the relatively higher sampling overhead. As each SHMT policy implements a subset of IRA-sampling [93], Figure 3.6 also includes that policy as another baseline. Implementing the full features of IRA-sampling will result in a 45% slowdown and render SHMT unusable.

Figure 3.6 also includes the performance of optimized GPU implementations with software pipelining as another reference design. Software pipeline can only achieve 1.25× speedup. For compute-intensive workloads, software pipelining cannot compete with SHMT. Software pipelining is effective for Blacksholes and MF as computation becomes relatively minor in these applications. However, this is not a limitation of SHMT. SHMT can potentially parallelize the data preprocessing part to further speed up these applications if appropriate hardware and algorithm exist.

3.5.3 Quality of QAWS policies

This section evaluates the quality of results for all proposed QAWS policies and their sampling mechanisms. We quantitatively measure result qualities using Mean Absolute Percentage Error (MAPE) and structural similarity index measure (SSIM). The exper-

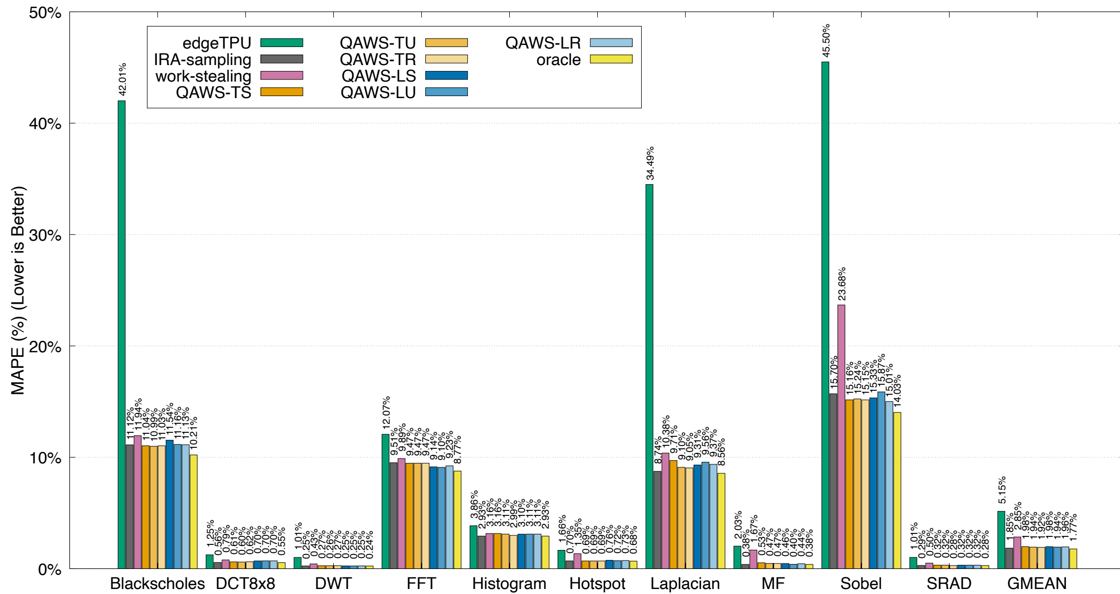


Figure 3.7: The Mean Absolute Percentage Errors (MAPEs) for SHMT applications

imental result shows all proposed policies can effectively improve the quality of results to a similar level.

Figure 3.7 shows the MAPEs of all QAWS mechanisms. In addition to SHMT policies and the baseline IRA-sampling mechanisms, we create an "oracle" scenario where we manually identify critical input data regions and assign HLOPs accordingly without considering the performance. If the program can only use less precise Edge TPUs, the MAPE is 5.15% on average. With careful manual optimizations, the MAPE of the Oracle assignment is 1.77%. The MAPE of the baseline IRA-sampling is 1.85%. Without using any QAWS policies, the pure work-stealing approach can deliver the result with an average MAPE of 2.85%.

For the proposed QAWS policies, the MAPEs of all policies are lower than 2% on average, close to the Oracle assignment and IRA-sampling. Furthermore, the difference in MAPEs between the QAWS policy with the lowest and the QAWS policy with the highest end-to-end latency is a marginal 0.07%, implying that a high-overhead sampling mechanism is overkill for most cases.

Due to the various result distributions of each application, the MAPEs across different applications vary significantly. For example, resulting images of edge detection type of applications, *Sobel filter*, and *Laplacian*, contain vast amounts of near-zero values representing non-edge areas. Thus, any moderately approximated non-edge value will contribute a much higher percentage of the error rate to the overall MAPE. The limitation in dealing with close-to-zero is a well-known issue of MAPE [88].

To more effectively evaluate the quality of results in image data containing near-zero values, we introduced SSIM as an additional metric. SSIM is a measure that predicts perceived visual quality, and an SSIM score of more than 0.95 is the generally agreed threshold of very good quality. We use SSIM for the six image-related workloads, *DCT8x8*, *DWT*, *Laplacian*, *Mean Filter*, *Sobel filter*, and *SRAD*. Figure 3.8 presents the SSIMs of these applications. All QAWS policies can maintain higher than 0.97 SSIMs as the average SSIM results across these applications are 0.9916, 0.9924, 0.9949, 0.9873, 0.9829, and 0.9798 for QAWS-TS, QAWS-TU, QAWS-TR, QAWS-LS, QAWS-LU, and QAWS-LR, respectively. All QAWS policies can achieve SSIM results close to the oracle of 0.9957, especially the top-K QAWS policies. This set of experiments again shows that using high-overhead mechanisms is not necessary in most cases.

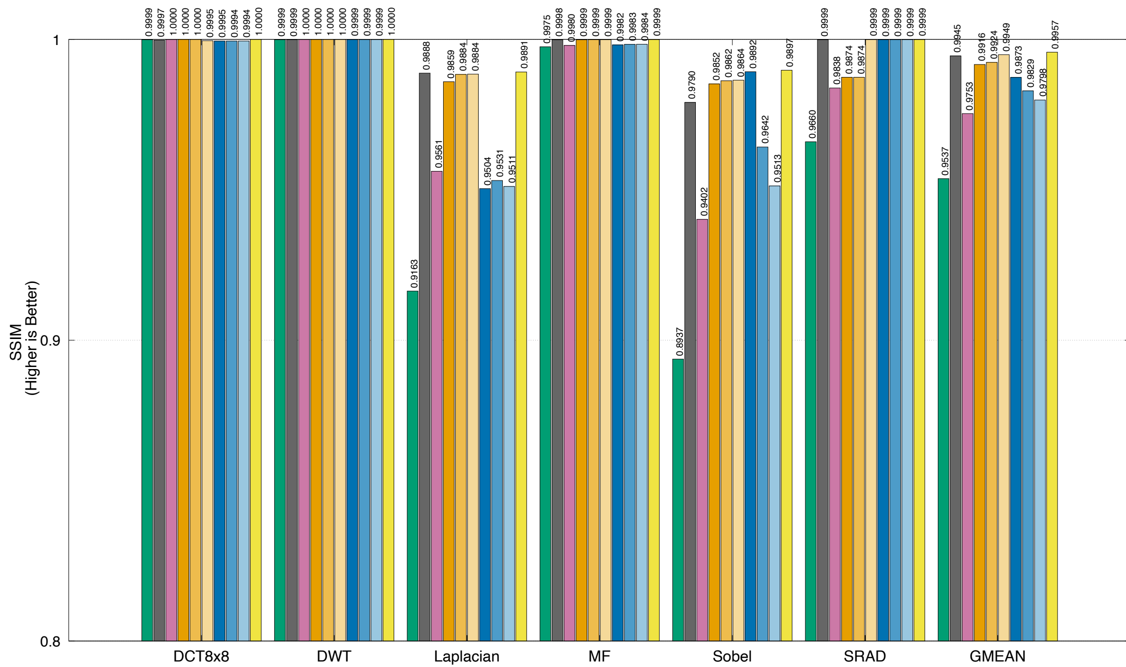


Figure 3.8: The Structural Similarity Index Measures (SSIMs) for image-related SHMT applications

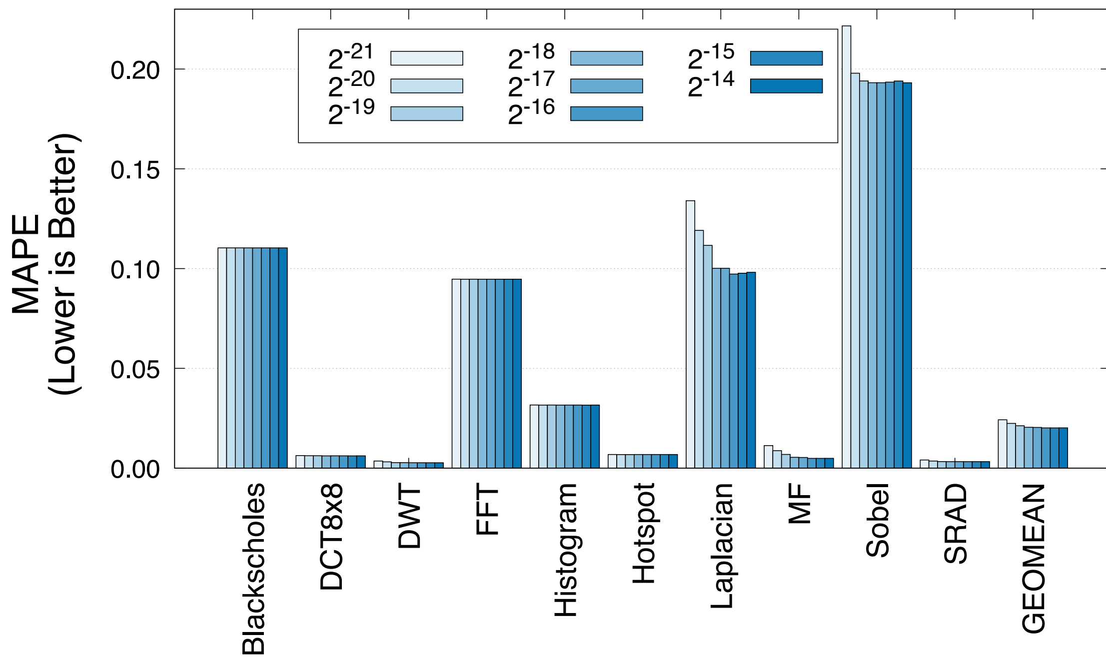
Since all QAWS policies deliver a similar level of result qualities but *QAWS-TS* obtains the best performance compared with all QAWS policies, in the rest of the paper, we use *QAWS-TS* by default.

3.5.4 QAWS sampling rate

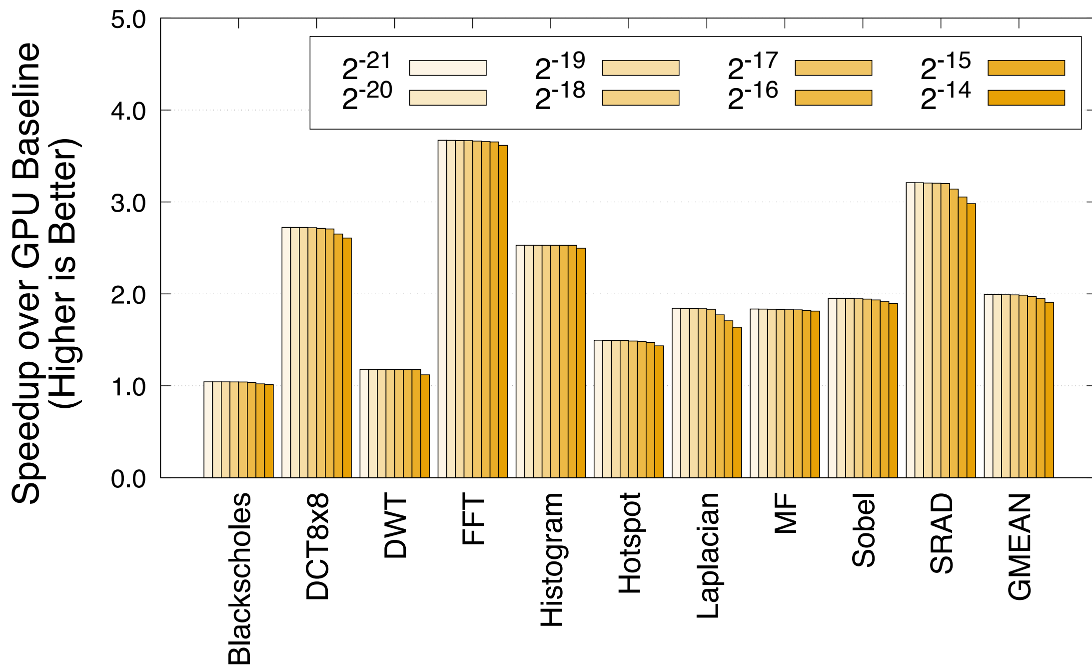
The number of samples during each sampling phase is another parameter that helps optimize the sampling overhead. Figure 3.9(a) and Figure 3.9(b) show the speedup and MAPEs when the sampling rate (the portion as samples from the raw datasets) of our best-performing QAWS-TS changes, respectively. A sampling rate of 2^{-14} means we select 256 samples from a 2048×2048-sized input. We changed the sampling rate from 2^{-21} to 2^{-14}. As SHMT's policy already reduces the post-processing after each sample, QAWS-TS achieves competitive performance regardless of the sampling rate. However, the MAPEs decrease monotonically until the sampling rate reaches 2^{-15}. The result suggests that the sampling rate of 2^{-15} can generate significant enough input samples for QAWS policies without sacrificing performance gain considerably.

3.5.5 Energy consumption

By reducing the total execution time and offloading computation to a lower-power-consuming Edge TPU, SHMT has a strong potential for energy saving. We connect the power source of the prototype through a power meter and collect the periodical measurements from the meter. Figure 3.10 reports the breakdown of energy consumption of both GPU baseline and SHMT with QAWS-TS. The same figure also shows the relative



(a)



(b)

Figure 3.9: (a) Quality v.s. QAWS sampling rates, (b) Speedup v.s. QAWS sampling rates

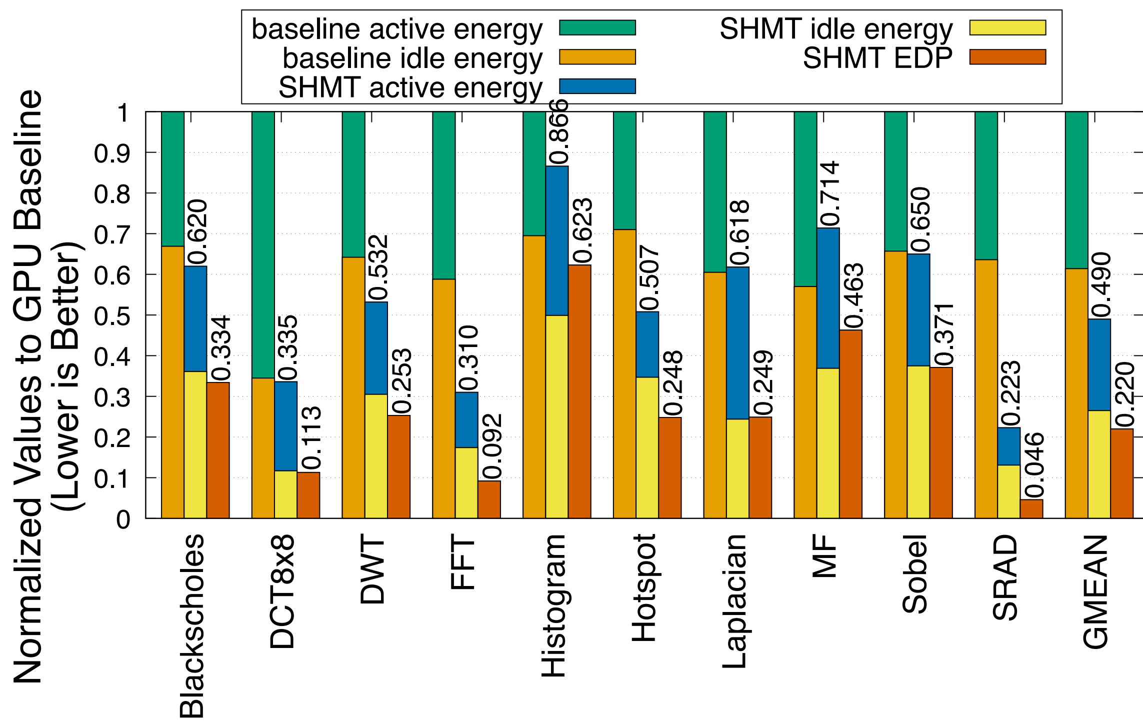


Figure 3.10: Energy consumption and energy-delay products (EDP)

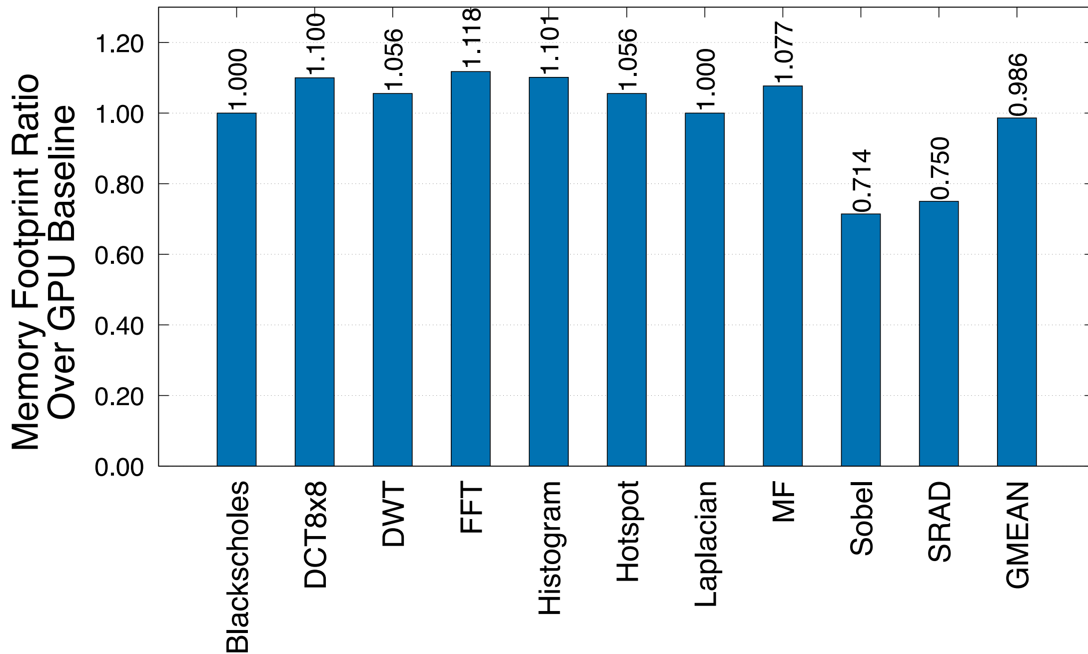


Figure 3.11: SHMT's Memory Footprint (Normalized to GPU Baseline)

energy-delay products (EDP) of SHMT with QAWS-TS, compared against the GPU baseline. SHMT with QAWS-TS reduces energy consumption and EDP by 51.0% and 78.0% on average, respectively.

The peak power consumptions of three cases including (1) platform idling, (2) GPU baseline, and (3) the SHMT with QAWS-TS are 3.02 watts, 4.67 watts, and 5.23 watts, respectively. Although SHMT with QAWS-TS reaches higher peak power since both GPU and Edge TPU are functioning during runtime than GPU baseline, on average, the 51.0% energy reduction of SHMT with QAWS-TS comes from the 1.95× speedup that reduces the period consuming the power with 5.23 Watt at peak.

Benchmark	Communication Overhead(%)	Benchmark	Communication Overhead(%)
Blackscholes	0.77%	DCT8x8	0.89%
DWT	0.66%	FFT	1.03%
Histogram	0.47%	Hotspot	1.04%
Laplacian	0.49%	MF	0.67%
Sobel	0.79%	SRAD	0.59%
GMEAN	0.71%		

Table 3.3: Communication Overhead

3.5.6 Memory and communication overhead

Figure 3.11 presents the total memory footprint when running benchmark applications at each process's virtual memory abstraction level. As the specialized logic in Edge TPUs provides more accelerated functions in hardware, Edge TPUs require less system memory than equivalent implementations on GPUs. For example, the buffers in Edge TPU processing elements can replace the memory in storing the intermediate results of vector products that GPUs require. As a result, the memory footprint of SHMT counter-intuitively reduces for applications with significant amounts of HLOPs on Edge TPUs, despite the additional buffers for inputs to Edge TPUs.

Table 3.3 describes the communication overhead resulting from the nature of peripheral devices like Edge TPUs. The computing resources in SHMT only spend about or less than 1% of the time, with 0.71% on the geomean average, waiting for data exchanges for the following reasons. (1) The parallel programming model of SHMT promotes data-parallel algorithms like matrix semiring tiling ones that implicitly have low data exchanges among parallel chunks of computing. (2) The computation time is relatively longer on each processing resource than the data exchange time, allowing mechanisms like double buffering

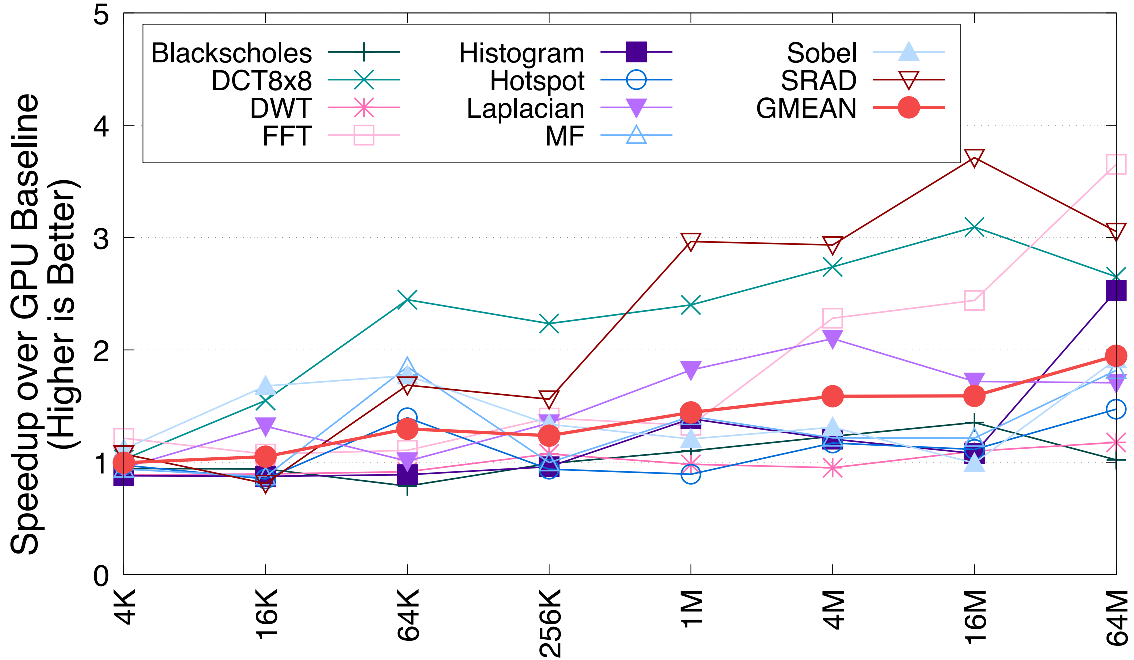


Figure 3.12: Speedup v.s. problem sizes

to hide the latency. (3) The amount of HLOPs from each application allows the SHMT runtime system to easily oversubscribe available processing resources and cover the latency of data exchange.

3.5.7 Discussion on SHMT's limitation

Figure 3.12 shows the speedups of SHMT under QAWS-TS variation when problem sizes of benchmarks vary. Within the tested problem size interval, from 4K to 64M, the speedup increases as the problem size increases. We did not go beyond 64M as the working set size of GPU kernels in some applications will surpass the physical memory limitation and crashes, not the limitation of SHMT. SHMT is more effective for larger problem sizes as larger problem size provides more parallelism among HLOPs for various devices.

Reviewing the result in Section 3.5.6 presents, SHMT does not lead to significant memory and communication overhead if we can leverage the embarrassingly data-level massive parallelism as the applications we demonstrated in this paper. Therefore, the adoption of SHMT simply helps the system to enjoy more parallel processing resources to tackle larger problem sizes without significantly further burdening the system. In other words, the limitation of SHMT is not the model itself but more on whether the programmer can revisit the algorithm to exhibit the type of parallelism (e.g., matrix tiling [190, 109, 146]) that makes SHMT easy to exploit.

3.6 Related work

Existing runtime for parallel programming on heterogeneous systems. Popular domain-specific languages, including TensorFlow [1] and Pytorch [133], allow the automatic delegation of domain-specific functions to one particular accelerator. Suppose the back-end implementation of functions can exploit parallelism among the delegated type of accelerators. In that case, these frameworks can concurrently execute pieces of computation on multiple devices but the same type. These frameworks can also employ pipeline parallelism to overlap different domain-specific functions with concurrency. However, none of the existing domain-specific language frameworks can employ multiple types of accelerators simultaneously in the manner that SHMT can perform. IR-level optimizations like XLA [51], or model-level optimizations like TVM [23] and AutoTVM [24] do not consider the simultaneous use of heterogeneous devices but can only optimize for a single type of device for each code region.

Heterogeneous programming frameworks like OpenCL [85] allow programmers to compose a single code version but generate binary running on multiple hardware devices. However, the OpenCL does not generate code that can simultaneously execute on heterogeneous devices. Though programmers can use OpenCL or other alternatives to create programs running in SHMT model manually, the resulting program still lacks scheduling flexibility and quality assurance.

OpenMP [21] provides an automatic parallel programming model that enables multithreading execution on homogeneous multi-processors. Through adding pragmas, OpenMP can exploit data-level parallelism and create homogeneous threads. SHMT can leverage the identified data-level parallelism and create parallel execution using HLOPs to make use of multiple types of hardware. However, without the abstraction and mechanisms that SHMT framework presents, existing homogeneous programming frameworks cannot take advantage of the presence of heterogeneous hardware.

Existing task distribution solutions for heterogeneous systems utilizing multiple accelerators in the system use the following methods.

(1) Partitioning one application and mapping the partitions onto multiple accelerators of the same type (such as GPUs) in the computer system for concurrent execution [84, 7, 132, 26, 113, 83, 117, 188, 182, 139, 153, 41, 197, 100]. Some works extend the same method to computer clusters such as distributed deep learning training/inferencing [43, 92, 77, 71, 66, 55], federated learning [174, 59, 165, 89], decentralized ad-hoc computing [40], inter-datacenter scheduling [151], and scalable computing on supercomputer environments [166, 152]. Al-

though these methods can achieve higher performance with parallel execution of multiple devices of the same type, they do not consider the simultaneous use of the other types of heterogeneous accelerators on the same system as SHMT does.

(2) Extending method (1) to multiple accelerators with different configurations or versions [113, 154, 193, 15, 162, 25] but still falling into the same type. HDA [91] can configure multiple heterogeneous dataflow accelerators for different neural-network layers where each only differs from others with different PE configurations and connecting topology. HASCO [178] can efficiently generate systolic array architectures with different configurations for executing various tensor computation kernels. Again works using this method do not overcome the challenge of programming model discrepancy among devices. SHMT presents a parallel programming and execution model addressing this challenge.

(3) Allowing limited concurrent usage of multiple types of accelerators only when the task execution triggers multiple types of dedicated functions at the same time [90, 171, 13, 163]. However, the behavior of the program's execution flow and the diverse characteristics of dedicated functions mapping to DSAs limit the simultaneous level of heterogeneous execution. Whereas SHMT provides a machine-independent programming model for task partitions such that the concept of SHMT can achieve higher hardware utilization and allow broader applicability for accelerators.

Heterogeneous computing for AI/ML workloads. The high computing demands of AI/ML workloads motivating the development of AI/ML accelerators provoke many performance optimization techniques that utilize heterogeneous accelerators. Examples are (1) tensor tiling [76, 194, 195], (2) pipelining [177, 48], (3) operation fusing [4, 118],

(4) neural architecture searching (NAS) [164, 98, 97, 99, 167], and (5) model quantization/compression [46, 169, 160, 74, 159, 6, 56, 170, 9]. Essentially, these techniques reconsider the computational graphs of AI/ML workloads for better workload-to-hardware matchings that exploit parallelisms. SHMT is orthogonal to these techniques as SHMT allows extensions upon these software-based optimizations that explore opportunities enabling intra-kernel concurrent utilization on multiple heterogeneous accelerators.

Existing quality assurance policies rely on several methods including (1) taking advantage of the precision-tolerable characteristic of workloads themselves like data precision adaptation on AI/ML models [169, 6, 56], (2) providing numerical composition solutions to increase resulting precision such as iterative refinement [57] and extended precision [45], or (3) performing mixed-precision computation [108, 38, 86, 180, 172] or providing multi-resolution data [70] to adjust overall required quality according to needs. Existing approximated techniques include loop perforation [96] and numerical approximation. Another example is IRA [93] which uses canary inputs to dynamically select the most effective approximation technique for speedup before target output quality (TOQ) violation happens.

SHMT is orthogonal to these quality assurance policies as our QAWS policies are low-overhead sampling methods without actual function execution runs. As long as any aforementioned policy has low-overhead and can avoid using application-specific prior knowledge to assure quality, they can substitute QAWS as a replaceable module.

This work needs additional quality assurance simply because the hardware performs approximate computing rather than the limitation of the concept SHMT itself. Con-

ventional homogenous simultaneous multithreading hardware does not need to cope with quality assurance. In contrast, SHMT has to ensure quality because of the potential precision mismatch of underlying architectures.

3.7 Conclusion

Modern computer systems are already heterogeneous and consist of several types of hardware architectures. Conventional execution models usually under-utilize these hardware devices by only offloading certain workloads that depend on the kernel's characteristics and performance requirements.

This paper presents SHMT, a framework for heterogeneous systems to enable a simultaneous and heterogeneous execution scheme. SHMT automatically partitions given VOPs of a workload into HLOPs to allow concurrent execution of these sub-kernels on heterogeneous devices. By integrating the concept of neural generalization, SHMT enables devices such as Edge TPU that have limited programming capabilities to contribute their computational powers. Also, QAWS policy mitigates the precision mismatch issue from accelerators with low data precision causing the potential result quality degradation. Throughout the low-overhead re-scheduling behavior of QAWS introduced on HLOPs, SHMT achieves less than 2% MAPE error across applications on average via prioritizing tasks over criticality. Also, SHMT achieves 1.95× speedup and 51.0% energy reduction by enabling simultaneous and heterogeneous execution of architectures compared to GPU baseline.

Chapter 4

GSLD: a Globally Sparse Locally Dense Matrix Computing Library

Sparsity is a ubiquitous data characteristic of real-world matrix-based computational applications across numerous scientific and engineering domains. Also, sparsity is a phenomenon of data layout from the coordinate system's perspective, and skipping computation and data accessing for zero-based no-effect equations such as `x*0=0` inside an application effectively and efficiently is challenging. Existing sparse research works focus on efficiently skipping zero values in the matrix and designing optimized data flow. However, an optimized data flow of sparse computation and capturing the particular sparsity patterns of an application are hard to design at the same time. Alternatively, relaxing the skipping for zero values in sparse matrices yields a different direction of optimization opportunity. We observed a significant amount of datasets have the "globally sparse, locally dense" (GSLD) property. Leveraging the insight, we propose a new sparse computing library that more in-

telligently uses dense matrix processors and scalar cores. Our Intel AMX-based prototype achieves 2.03× times speedup compared to a high-end CPU baseline.

4.1 Introduction

Sparsity is a ubiquitous data characteristic of real-world applications across numerous scientific and engineering domains. Sparsity is also a phenomenon of a set of data points given upon perspectives such as a coordinate system and ordering definition in dimensions.

Existing matrix accelerators such as NVIDIA's TesnorCore Units (TCU), Google's Tensor Processing Units (TPU), and Intel's Advanced Matrix Extensions (AMX), are specialized accelerators for processing dense matrices efficiently on prioritized computing kernels such as general matrix multiply (GEMM) for artificial intelligence (AI) and machine learning (ML) applications. On the other hand, general-purpose hardware architectures such as central processing units (CPUs) and general-purpose graphic processing units (GPGPUs) don't natively support sparse-aware computing, which leads to the `x*0=0` performance-improving opportunity unexploited.

Existing sparsity-aware hardware accelerators exploit either sparse acceleration features (SAFs), dataflow of target applications, or both dimensions, to achieve optimal performance in real-time. Sparseloop [176] defines SAF in three terms: representation format, gating, and skipping. Representation format refers to the choice of a sparse format storing non-zero values and the associated location information. Gating refers to the idleness of the storage and/or computing units to avoid ineffectual operations. Skipping refers to the exploitation of such ineffectual operations by avoiding spending cycles.

However, two challenging aspects of existing sparse computing scenarios remain under-exploited. (1) One single computing kernel in different application domains, sometimes even in the same domain, may need to compute on different types of sparse patterns of input matrices, where no single SAF is a one-design-fits-all choice in terms of performance. Similarly, different computing kernels may share the same sparse matrix as inputs, where both have to consider their dataflows of computations separately in their sparse acceleration solutions. (2) Sparse format transformation overhead and the re-design of both dataflow and SAF associated with the format are expensive. Sparse computations are fundamentally optimization works. The level of optimization of a narrow range of applications is extremely high as the exploitations are close to the theoretical optimal such as GEMM computation. On the other hand, sparse computing solutions usually under-exploit generalizability and leave the selection for users.

In response to the design challenges, we proposed GSLD – a sparse computing library that relaxes the optimization goal by trading off optimized SAF with broader applicability of the library's usage domains. From a high-level perspective, GSLD does not differentiate between dense and sparse matrix computation and thus relieves the choice burn of the users from selecting a proper computing library and storage format that together affect the computing performance, and thus complex the real-world design decisions in practice. Based on the observation that real-world sparse data are typically "globally sparse, locally dense", GSLD exploits such property and intelligently leverages dense matrix processors for locally dense sub-matrix computation along with the assistance of other scalar cores in a system.

GSLD indifferences dense and sparse matrices by adopting a sparsity threshold checking that allows not-so-sparse sparse computation using a dense library. Based on the rule of thumb that a sparse library will only start to gain performance benefits when the sparsity of the input data is lower than 1%, which comes from the experiment result of SIMD2 [190], GSLD focuses on optimizing sparser sparse computing tasks by capturing the GSLD property. We also observe that real-world sparse patterns are not random, and one of the common sparse patterns is diagonally dominant. Combining these two observations, the GSLD design includes a diagonal-dominant-based compressed sparse row (CSR) sparse format for the sparser sparse computations that address the design challenges of sparse computations.

We built a prototype GSLD library by integrating Intel's AMX extension cores to evaluate GSLD's performance compared to the best-performing sparse library in terms of end-to-end latency. GSLD demonstrates that programmers can simply replace their matrix computation by integrating the GSLD library as the computing kernels. The experimental result shows that GSLD achieves 2.03× times speedup.

In presenting GSLD, this paper makes the following contributions.

- GSLD presents a new matrix computing library that eases the implementation burden of optimizing sparse computation with complex design choices such as sparse format and data flow design.

- GSLD evaluates and demonstrates the potential of relaxing the traditional optimization goals of sparse computation and providing broader application domains for users.

- GSLD proposes a new sparse format that captures one of the commonly seen sparse patterns that is diagonally dominant.
- The prototype implementation of GSLD shows 2.03× times end-to-end latency speedup by integrating Intel's AMX extension cores for computing locally dense sub-matrices along with the assistance of other scalar cores.

4.2 Background and motivation

This section describes the background and motivation of GSLD. By investigating the real-world sparse data, we observed the GSLD property and commonly seen sparse patterns that serve as the backbone motivation of the proposed GSLD matrix computing library.

4.2.1 Observation on real-world sparse data

Sparsity is a general terminology describing any matrix or tensor-based data that has a certain amount of zero values within the layout of the grid cells. The sparse pattern of a sparse matrix describes the organization of such zero values in the layout, and the pattern of such locations is a critical factor affecting the level of optimization a potential sparse library can achieve. Depending on the associated computation, the theoretical optimization in the performance of a sparse matrix computation can reduce the theoretical time complexity by orders of magnitude. For example, although the time complexity of a naive general matrix multiply (GEMM) implementation is in the order of 3, a sparse corresponding case that is also called sparse GEMM (spGEMM), can be as low as in the order of 2 when the number

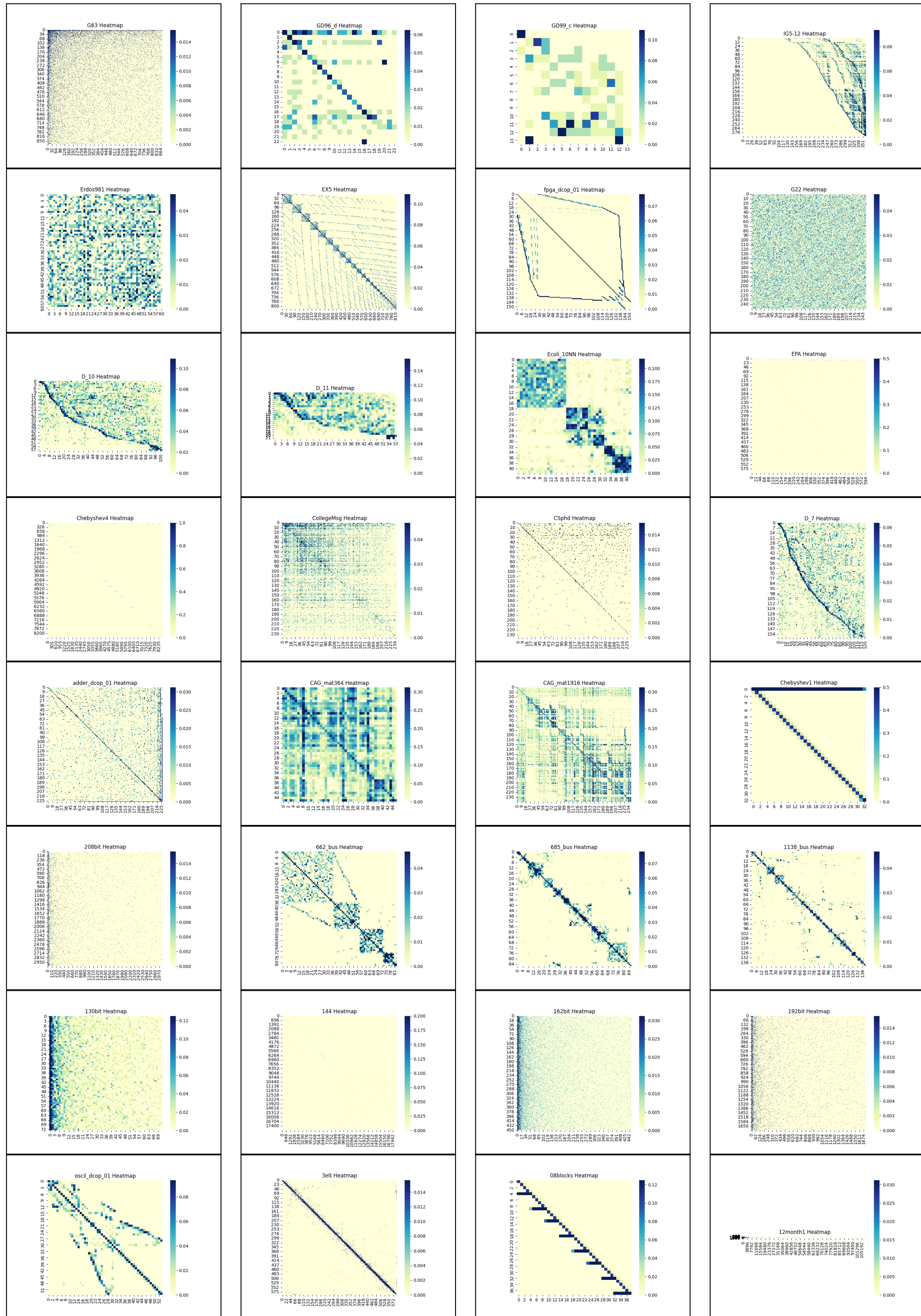


Figure 4.1: 32 representative real-world sparse matrices.

of non-zero values is proportional to the dimension size of the matrix. In the extreme case, a spGEMM with the input sparse matrix having only one non-zero value will have a constant time complexity. The critical challenge of sparse computation optimization is that any other scenarios between the two extreme cases are hard to achieve, and providing a general solution for different cases is even more challenging.

To better understand the sparse characteristics of real-world sparse data, we explored the SuiteSparse Collection [31] sparse data in full scale. Figure 4.1 shows 32 representative sparse matrices from the SuiteSparse Collection out of 2,878 effective samples. By using a k-medoids clustering algorithm with a customized matrix-based similarity metric, these 32 selected samples as cluster centers represent some of the important and common characteristics. Based on the sampling result, we can see that 16 samples have prominent diagonal dominant non-zero value patterns. Also, there are other common patterns such as uniform distribution and hybrids.

4.2.2 The GSLD property

Beyond visualization, we further investigated all effective samples at full scale and identified the GSLD property. To identify the density properties in both global and local scales, we defined both the typical global density and non-zero block density for each, respectively. A non-zero block of a sparse matrix is a local square region that contains at least one non-zero value. Because different sparse matrices have different sparse patterns, we classified them into four main categories including (1) dense-ish, (2) sparse and diagonal dominant, (3) sparse and symmetric, and (4) sparse and others.

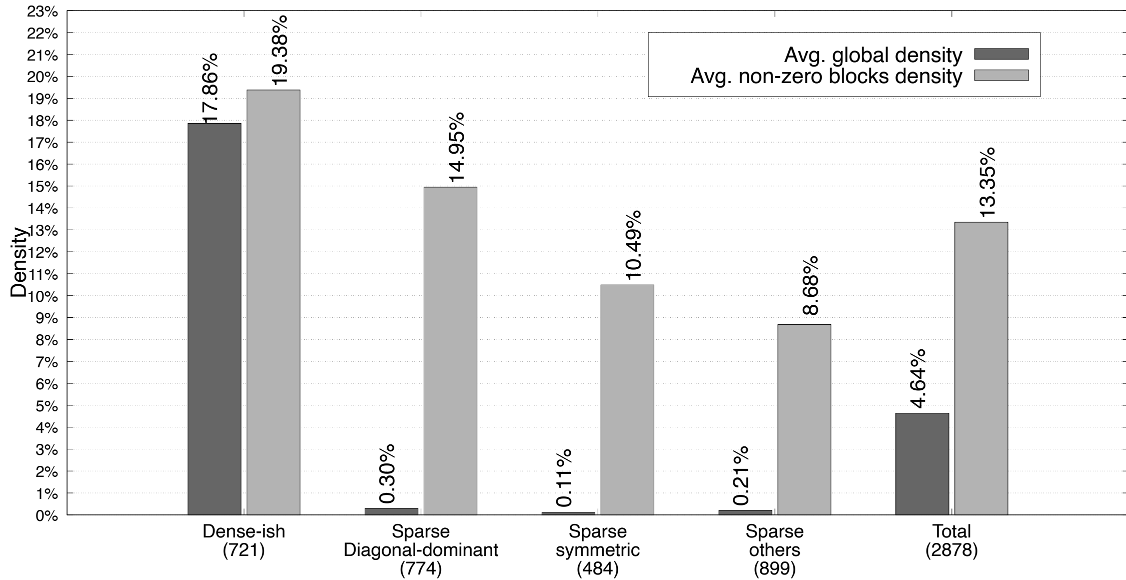


Figure 4.2: The Globally Sparse, Locally Dense (GSLD) property (number of samples)

We proposed a heuristic classifying method as the following describes. First, the method will check the global density of a given matrix. If the density of the matrix is larger than the threshold, which is 0.01 in our case, then the matrix is dense-ish. Otherwise, the method will further check if the matrix is diagonally dominant. A sparse matrix is diagonal dominant if one of the following conditions is satisfied. (1) The diagonal blocks have 50% density on average, or (2) at least 50% of the non-zero values of the sparse matrix located within the diagonal blocks region. Notice that two factors determine the region area: the block size of each block and the duplication number of a diagonal block in a horizontal direction. The latter is to capture more non-zero values for the cases in which diagonal patterns were wider visually. The last check is to determine if the matrix is symmetric.

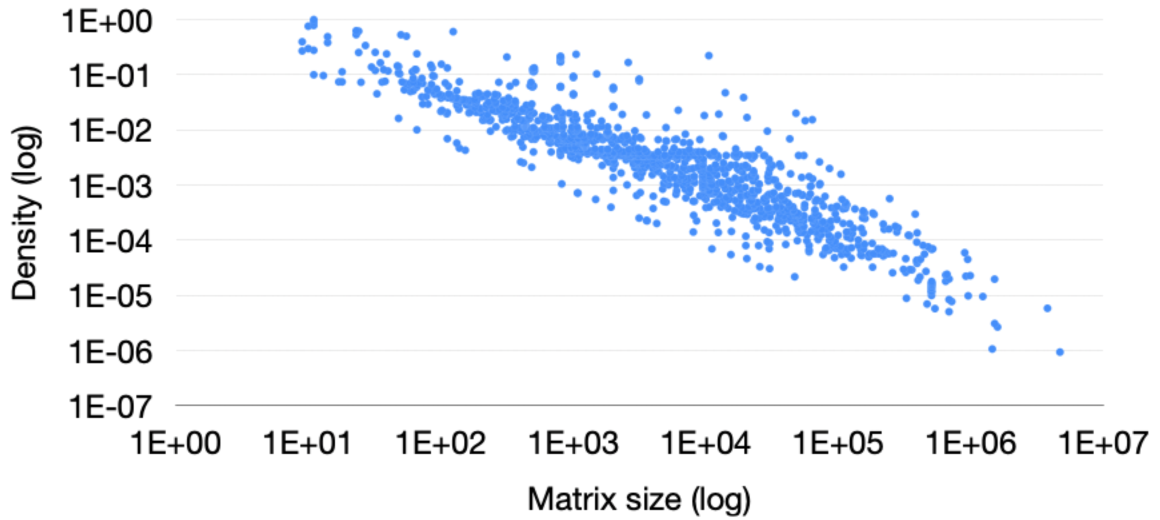


Figure 4.3: Globally Sparse

Figure 4.2 summarizes the global density and local density of samples. By summarizing the average results, we can see that all three sparse type categories have significant GSLD properties. The average global densities are all significantly lower than the average non-zero block densities. Compared to the dense-ish type one, the global to non-zero block density ratios of the three sparse types are all about 50 times larger. We can see that for a matrix that is sparse enough, the GSLD property will be prominent. Notice that it is not obvious to identify the GSLD property if we only look at the total case. Also, we denoted the number of samples examined for each type in the graph. And each category has a comparable number of samples.

Figure 4.3 and Figure 4.4 further show the GSLD property in visualization at full-scale. Figure 4.3 demonstrates the scatter plot of all effective samples with the matrix sizes as the x-axis, and the densities as the y-axis. We can see that the density range has

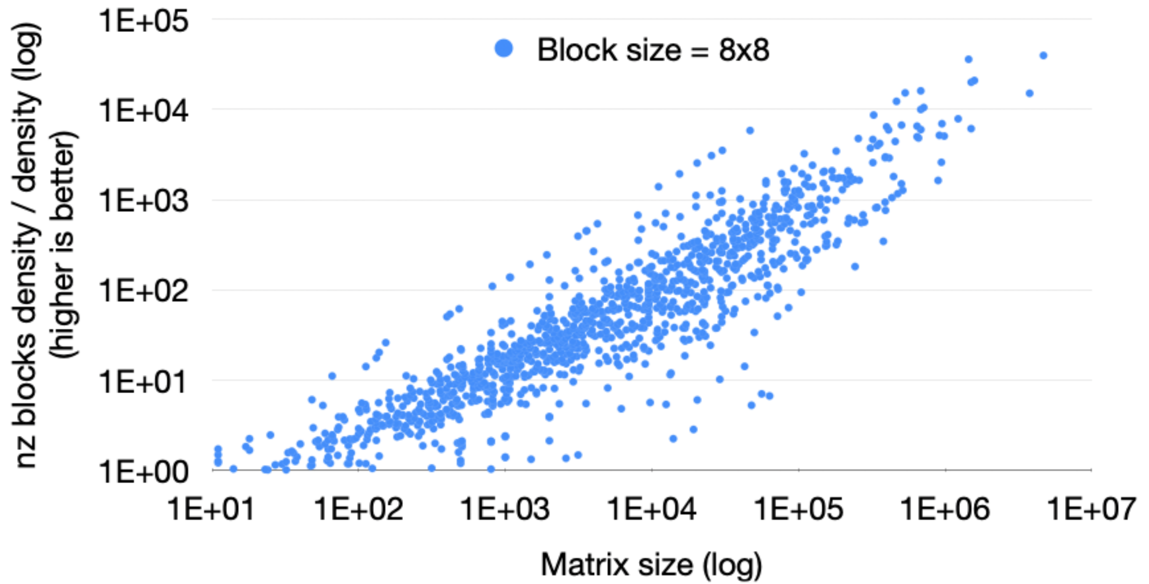


Figure 4.4: Locally Dense

wide coverage from 10 to the order of 0 to 10 to the order of -6, and real-world sparse matrices tend to be more sparse when the matrix size is larger. Figure 4.4, on the other hand, visualizes how dense local regions of a sparse matrix can be compared to global density. The local-to-global density ratio at the y-axis represents how strong the GSLD property a sparse matrix has. We also observe the trend that a larger sparse matrix tends to have stronger GSLD property. Table 4.1 summarizes more detailed matrix characteristics including the number of application domains of each type.

block size = 8x8		Dense	Sparse - Diagonal dominant	Sparse - symmetric	Sparse - others	Total
percentage of samples (out of 2878)		25.052%	26.894%	16.817%	31.237%	100%
density	average	17.861%	0.295%	0.112%	0.209%	4.638%
	geomean	3.011%	0.133%	0.028%	0.101%	0.238%
NZ block density	average	19.376%	14.955%	10.485%	8.683%	13.352%
	geomean	13.333%	9.659%	7.549%	6.823%	8.783%
number of NZ blocks / number of blocks	average	38.351%	3.316%	1.765%	3.275%	11.819%
	geomean	29.021%	1.061%	0.451%	1.406%	2.302%
number of unique matrix domains		50	56	38	52	89

Table 4.1: Sparse matrix charecterization

4.3 Overview of GSLD

The proposed GSLD matrix computing library follows the two design principles to address the issues mentioned in Section 4.1.

A unified matrix acceleration library interface

The existing division between dense and sparse computing scenarios separates the use of the underlying kernel library. Also, the existing interfaces of each are independent such that the additional arguments related to sparse format differentiate the interface usage of dense and sparse ones. GSLD aims to blur the separation and provide a unified matrix library interface for matrix computation tasks in either situation.

An ideal unified matrix computing library not only keeps the interface standard but also guarantees performance for various sparse input matrices. We propose GSLD that in theory compromises achieving optimal performance for every possible sparse scenario but provides a unified interface. GSLD captures the globally sparse and locally dense properties of sparse matrices in general, and yet the design of GSLD effectively achieves performance gain on average because of the following reasons: (1) Disgarding the entrenched differentiation rule of dense and sparse matrices, but only separate them with different implementations based on the consideration of performance instead of sparsity. (2) Using only one single static sparse format that approximately yet effectively captures the commonly seen sparse pattern in exchange for avoiding the design of domain-specific data flow optimization techniques.

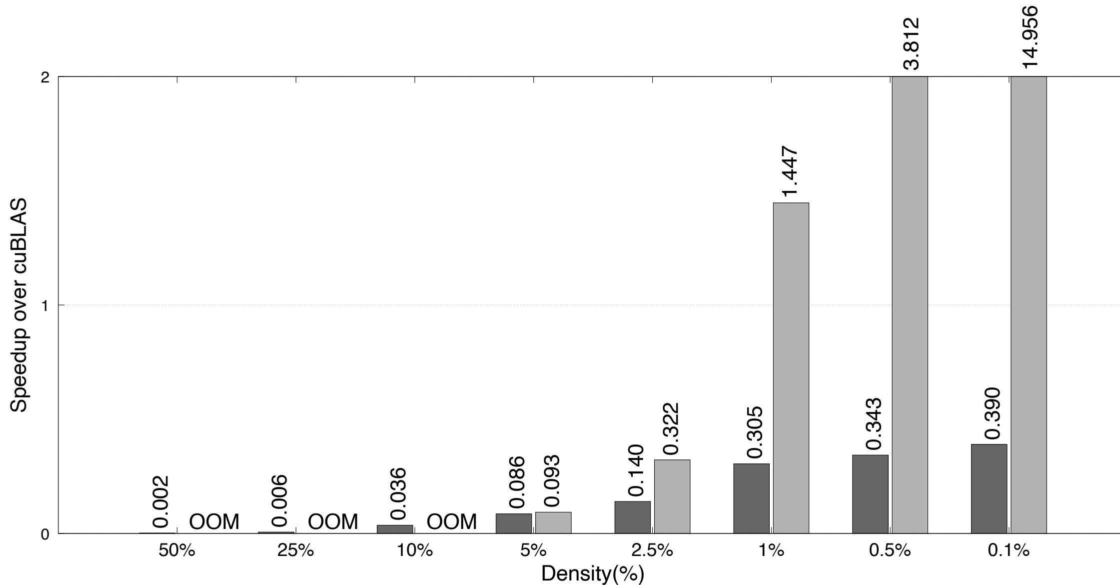


Figure 4.5: Performance of sparse matrix multiplication

Figure 4.5 shows the rule of thumb regarding how to determine a performance implementation of dense or sparse matrix computation, and it originates from SIMD2 [190]. In this graph, we can see that a spGEMM sparse implementation will start to outperform a dense counterpart when the sparsity is lower than 1% for the 4K by 4K size case. In other words, a dense cuBLAS implementation will keep performance for sparse matrix multiplication for sparse matrices that are not-so-sparse. Additionally, a sparse implementation will have an out-of-memory runtime issue in the case that the density is relatively higher. This is because using a sparse storage format, which is CSR in this case, for a dense-ish matrix requires more memory space due to the additional coordination-related information compared to simply using a naive dense storage format.

A standard sparse storage format

Determining an efficient sparse format for sparse computation in terms of both memory requirement and runtime performance is challenging. Also, users ultimately do not need to care about the design choice of the sparse format as long as a library can guarantee performance matrix computation. The sparse format will thus become a less meaningful factor to expose to the users. To achieve the goal of the first design principle, GSLD incorporates a proposed sparse storage format that statically captures commonly seen sparse patterns in exchange for chasing optimal data flow in runtime. Although counterintuitive in the first place, GSLD's sparse format at the maximum level captures the sparse pattern in general which implies a corresponding static data flow is efficient enough.

In GSLD, we proposed a diagonal dominant-based CSR sparse format as an optimized option for sparse format implementation.

4.4 GSLD implementation

This section describes the implementation details of GSLD library in three different aspects: (1) The core design algorithm, (2) the proposed diagonal-dominant CSR sparse format, and (3) the sparse computation of applications.

4.4.1 The core design algorithm

Algorithm 6 describes the core design algorithm of GSLD in a high-level perspective. To align with the design principle mentioned in Section 4.3, the algorithm takes matrix inputs in either dense or any level of sparse. Depending on the result of density

Algorithm 6 The core design algorithm of GSLD

Input: input matrix in either dense or sparse

1: **if** $density > threshold$ **then**

2: $\quad Call\ dense\ implementation.//ex : cublasSgemm();$

3: **else**

4: $\quad Call\ sparse\ implementation.//ex : gsld :: spmv();$

thresholding, GSLD will invoke a proper internal matrix computation implementation. For dense-ish cases, GSLD adopts existing high-performance dense implementation based on the rule of thumb demonstrated in Figure 4.5 and mentioned in Section 4.3. For sparse cases, GSLD will invoke the actual underlying sparse library implementation that provides a set of BLAS-like matrix kernels including sparse matrix-vector multiplication (spMV).

4.4.2 The proposed diagonal-dominant CSR sparse format

The sparse part of the library implementation includes a special sparse format that captures the commonly seen diagonal dominant pattern. Figure 4.6 illustrates the proposed sparse format. The format stores all diagonal blocks in dense format regardless of how dense the local block may be and stores the rest of the non-zero values in CSR format. This sparse format avoids recording additional coordination-related information for most of the dense blocks to save space. Although this is an approximate design for capturing sparsity patterns, the static nature of a sparse format allows for simpler sparse implementation without chasing the complex data flow of specific tasks and inputs.

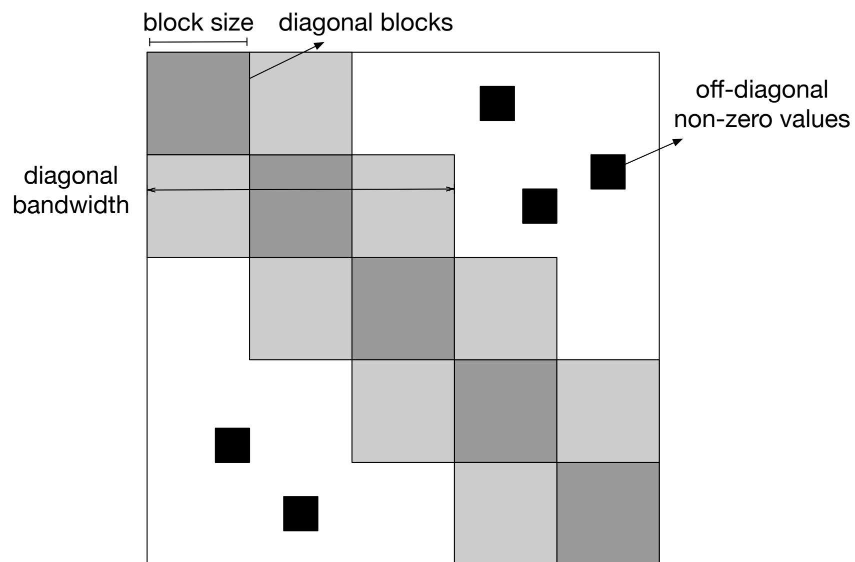


Figure 4.6: The proposed diagonal-dominant + CSR sparse format (diablock+CSR)

Dataset(number of samples)	Application	Kernel
Graph(547)	PageRank	spMV
Linear programming(338)	Karmarkar's algorithm	spMV
Structural programming(292)	linear system solving	iterative - spMV
Circuit(263)	linear system solving	iterative - spMV
CFD(184)	linear system solving	iterative - spMV
Chemistry(137)	linear system solving	iterative - spMV
DNN intermediate layers (Flexagon) [111]	DNN/ML (Flexagon) [111]	spMspM
OuterSPACE(20) [127], MatRaptor(14) [158], Gamma(31) [187], ExTensor(11) [61]	multiply the same sparse matrix by itself	spMspM

Table 4.2: Sparse matrix computations

4.4.3 The sparse computation of applications

We summarize some of the most common applications used for the matrices we profiled including the SuiteSparse Collection, and Table 4.2 shows the mapping from samples, applications, and the essential kernel. General sparse computation has a very broad application domain and computational characteristics. However, we observed that many applications share common kernels, and spMV is one of the most used kernels in applications.

4.5 Experimental methodology

This section describes the system configuration and applications we evaluated for GSLD library.

Experimental platform

We used a 64-core Intel XEON 6530 processor with a clock rate of up to 4 GHz. The processor has the Advanced Matrix Extensions (AMX) co-processor that is capable of computing dense matrix multiplication. We installed Ubuntu 24.04 (Linux kernel version 6.8.0-11-generic). For the GPU baseline part of the experiment, we additionally have an NVIDIA RTX 3090 GPU with 24 GB of device memory and CUDA 12.3.

Applications

For comparison, we implemented the single-CPU version of spMV computation by the PageRank application in Graphit [17]. For the multiple-CPU version, we additionally

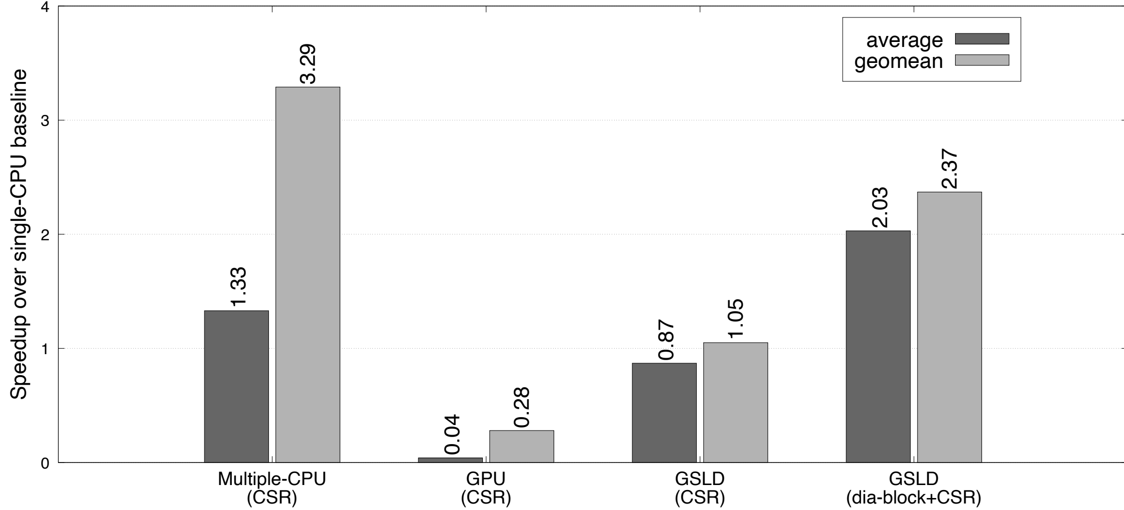


Figure 4.7: GSLD - spMV end-to-end Speedup

enabled the OpenMP [21] feature of the baseline. For the GPU baseline, we used the spmv_csr kernel of the cuSPARSE [120] GPU library.

4.6 Results

This section describes the speedup, performance result of full-scale sampling, and memory space-saving comparison against the scenario where a matrix computation uses a dense storage format. Compared to modern CPU-based platforms running optimized sparse matrix computation, GSLD exhibits improved performance and controlled memory save requirement. In addition, the GSLD in full-scale testing result shows a reliable lower bound performance guarantee for various real-world sparse data inputs.

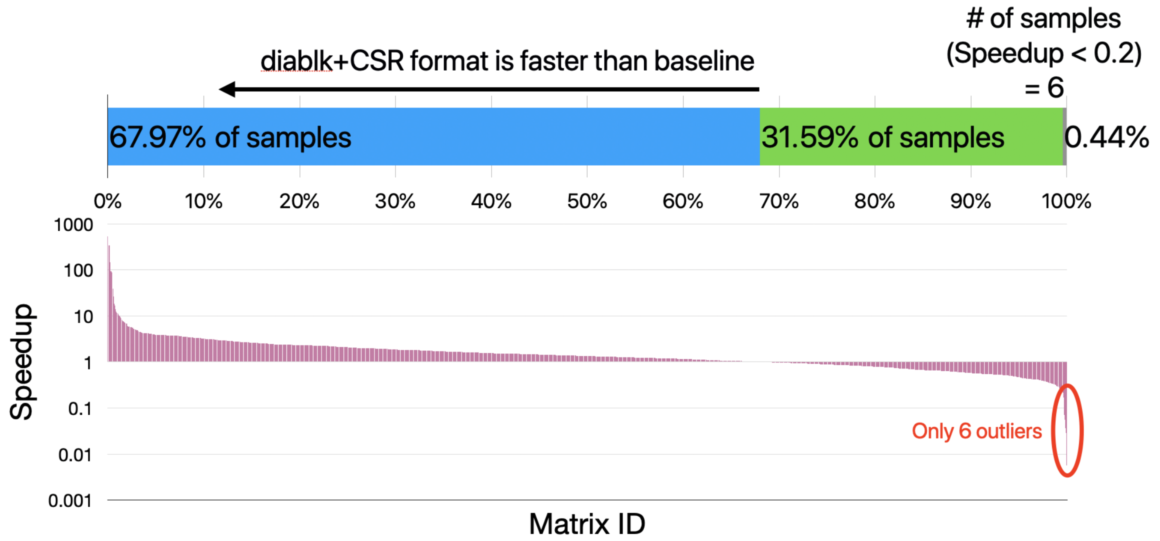


Figure 4.8: Full-scale testing

4.6.1 Speedup of end-to-end latency

Figure 4.7 shows the end-to-end latency result of the spMV kernel. We evaluated a few versions of sparse implementation including single-CPU as the baseline, multiple-CPU, GPU, the GSLD with Intel AMX kernel, and additionally the same GSLD implementation with the proposed diagonal dominant CSR sparse format. On either average or geomean of a total of 197 samples included, multiple-CPU version has significant end-to-end speedup, and GPU ones suffer from data movement as the main reason for the slowdown. As for our GSLD implementation in standard CSR format, the geomean result is slightly better than the baseline. However, the GSLD with diagonal dominant plus CSR sparse format achieves 2.03× speedup on average.

4.6.2 Full-scale testing on real-world samples

Beyond the selected samples, we also tested the spMV kernel on all sparse matrices from the SuiteSparse Collection to understand the possible worst-case performance of the GSLD library. In our experiment, as shown in Figure 4.8, around 2 thirds of the samples have speedup improvement over a single-CPU baseline. Among the rest of the cases, most of them are at least 1 fifth of the CPU baseline performance except for only 6 outliers. The proposed diagonal dominant plus CSR format is a beneficial option with a reasonable lower bound for almost all of the cases. We do not observe a significant correlated factor for the performance trend as the correlation coefficients between performance and factors including sparsity and matrix size are all close to zero. The correlation coefficients between performance and matrix size, density, sparse pattern, and the number of non-zero values are 0.00388, 0.00196, 0.00766, and -0.0460, respectively.

4.6.3 Diagonal area density

Figure 4.9 shows the average density of the diagonal area defined by the proposed diagonal dominant plus CSR sparse format. Due to the diagonal-dominant nature of the diagonal-dominant type of samples, the average and geomean diagonal area densities are 44.53% and 28.95%, respectively. Both results indicate that the proposed sparse format captures the sparse patterns with a regional density significantly larger than the density threshold based on the rule of thumb mentioned in Section 3.3.1. The sparsity capturing contributes to the performance of utilizing dense matrix processors for those diagonal blocks in diagonal.

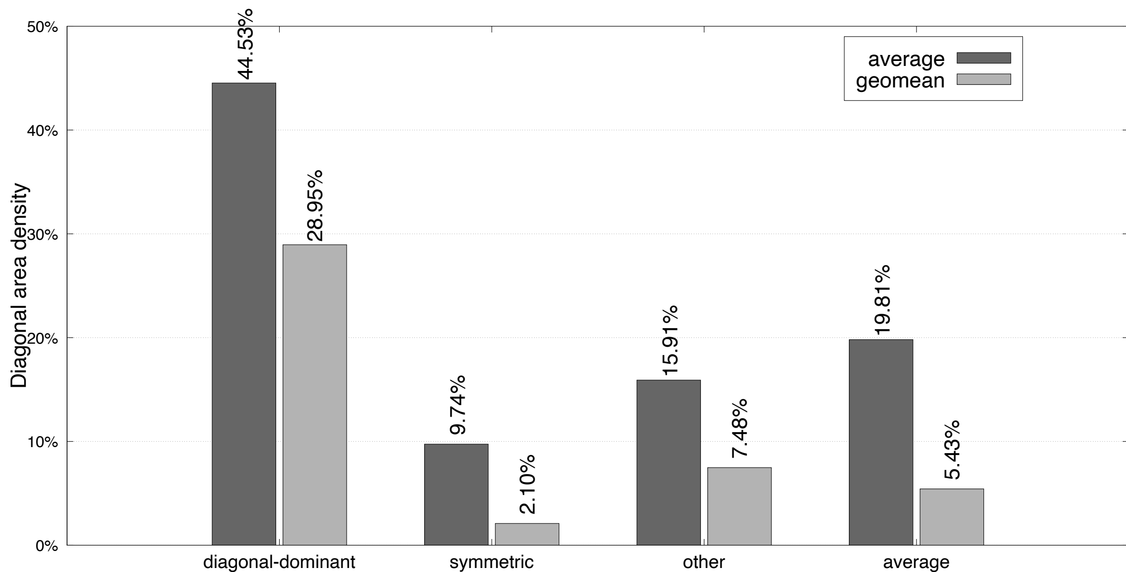


Figure 4.9: Diagonal area density

4.6.4 Memory space saving of sparse formats

Figure 4.10 shows the memory space requirement of various sparse matrix formats. By varying the block size of the proposed format from zero, which means the typical CSR format toward larger block sizes, we can see that a larger block size requires more memory size as we can expect. However, when we compare the relative sizes against dense counterparts, the diagonal-dominant+CSR format still has quite significant savings. The block size is a critical factor of the format and we selected the moderate 32 in size in the experiment. For the dense-ish category of results, the memory saving is significantly less than other types since the amount of coordination-related information undermines the expected memory saving of sparse formats as the saving is at the magnitude of 10 to the power of 2. There is one only memory-saving overturn that happens when using a diagonal domi-

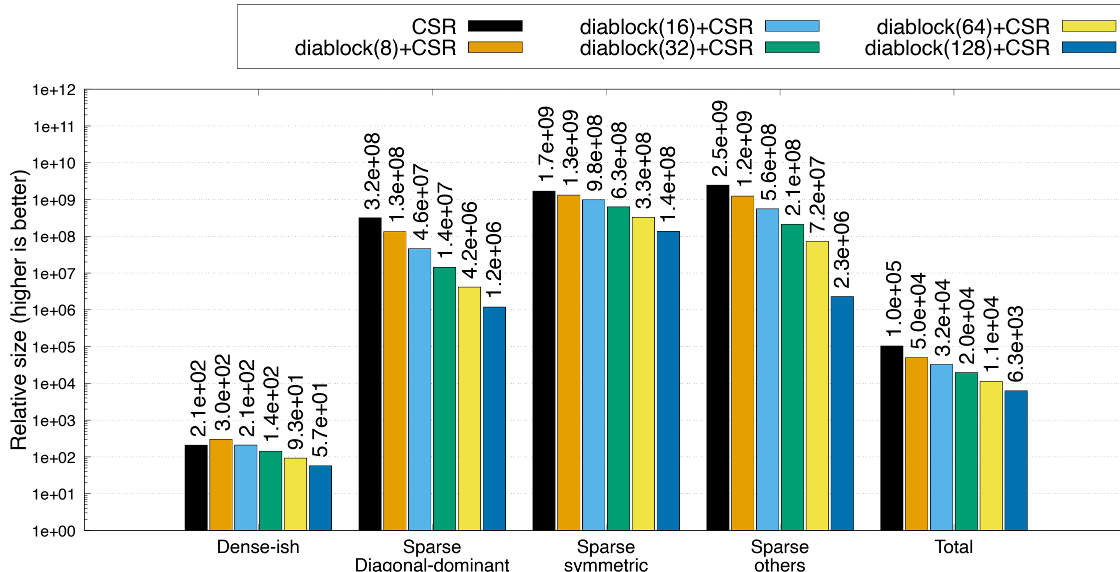


Figure 4.10: Memory space saving of sparse formats

nant plus CSR sparse format with a block size equal to 8 compared to naive CSR format. For other types, the proposed format still maintains at least 1 tenth of the memory saving compared to the naive CSR format.

4.7 Related work

Many prior works focus on designing sparse accelerators for optimizing sparse computation. Examples are ExTensor [61], SIGMA [136], SpArch [191], MatRaptor [158], Tensaurus [157], TensorDash [106], GoSPA [35], ALRESCHA [12], Sparseloop [176], Sextans [155], SPAGHETTI [63], SparseCore [140], TeAAL [115], Sparse Abstract Machine [68], Sparse-TPU [60], Flexagon [111], Cerberus [72], Symphony [134], and more [173, 102, 125]. Capstan [143] took a similar approach to our work as it targets both dense and sparse tensor applications. However, our work does not propose new hardware architecture but leverages

existing dense matrix processors in a system to achieve the same. VEGETA [75] extends the ISA of CPUs for sparse tile computation with explicit programming interface support for AI workloads, while our work is sparsity-agnostic and aims for general computations. Other than AI/ML tasks, some works focus on general computations. Spatula [44] proposes a hardware accelerator for sparse matrix factorization. Also, due to the memory-bound characteristics of sparse computation, some works focus on offloading computation toward memory or storage, examples are SpaceA [179], SparseP [49].

4.8 Conclusion

Sparsity is one of the most important characteristics of data in many general computing workloads. However, optimizing the sparse computation associated with the specific sparse patterns of input data and desired data flow toward optimal performance is challenging and usually domain-specific.

This paper presents GSLD, a matrix computing library for general sparse matrix computation that leverages the "globally sparse, locally dense" property of sparse data in general and proposes a diagonal-dominant CSR sparse storage format that explores a trade-off opportunity between chasing the optimal performance for domain-specific sparse computation and extending the usage spectrum of the sparse matrix computing library. Leveraging the insight, GSLD can intelligently use dense matrix processors and scalar cores for spare computation. Our Intel AMX-based prototype implementation achieves 2.03× speedup compared to the CPU baseline.

Chapter 5

Conclusions

Modern computer systems are increasingly heterogeneous, integrating various hardware accelerators due to the end of Dennard Scaling. These domain-specific accelerators, including GPUs, TPUs, and NPUs, meet the growing computational demands of AI and ML applications. Despite differences in their microarchitectural designs, tensor processors, which focus on efficient matrix-based computations, play a crucial role in enhancing system performance. This dissertation introduces a new programming paradigm that leverages tensor processors for general-purpose computing beyond their traditional AI and ML domains.

Firstly, GPTPU, mentioned in Chapter 2, proposes a general programming framework that additionally explores the usage opportunity of tensor processors in a computer system for general applications beyond AI and ML. With the help of the tensorizer, GPTPU demonstrates the mechanism of dynamically and transparently mapping AI/ML-specific operators that lead to efficient use of the underlying tensor processors. The GPTPU prototype system achieves $2.46\times$ speedup and 40% energy reduction compared to modern CPUs.

Secondly, SHMT, mentioned in Chapter 3, proposes a new programming model and execution model that provides "real" parallel processing using heterogeneous processing units for the same user function. SHMT presents two abstractions, VOPs and HLOPs along with mechanisms that coordinate concurrent execution on heterogeneous hardware components. The prototype implementation of SHMT archives 1.95× speedup with the proposed quality assurance mechanisms and also achieves 51.0% energy reduction.

Lastly, GSLD, mentioned in Chapter 4, is a new matrix computing library that accommodates either dense or sparse input matrix data. GSLD leverages a critical insight that sparse matrix data are generally "globally sparse, locally dense" and is a sparsity-agnostic matrix library that intelligently uses dense matrix processors and scalar cores. The Intel AMX-based prototype implementation achieves 2.03× speedup compared to the CPU baseline.

Beyond the proposed research works, we believe that more work can be done to realize the future of general-purpose computing using tensor processors. The heterogeneity of tensor processors should not be an obstacle to the computation in future computer systems but is an opportunity for designing a more powerful computer system with better performance for more general computing workloads.

Bibliography

[1] 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, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, 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, 2015. Software available from tensorflow.org.

[2] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 382–394, New York, NY, USA, 2017. Association for Computing Machinery.

[3] K Aludaat and Moh'D Alodat. A note on approximating the normal distribution function. *Applied Mathematical Sciences (Ruse)*, 01 2008.

[4] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '16.

[5] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose code acceleration with limited-precision analog computation. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture*, ISCA '14.

[6] Sam Amiri, Mohammad Hosseinabady, Simon McIntosh-Smith, and Jose Nunez-Yanez. Multi-precision convolutional neural networks on heterogeneous hardware. In *2018 Design, Automation Test in Europe Conference Exhibition*, DATE '18.

[7] Sam Amiri, Mohammad Hosseinabady, Andres Rodriguez, Rafael Asenjo, Angeles Navarro, and Jose Nunez-Yanez. Workload Partitioning Strategy for Improved Par-

allelism on FPGA-CPU Heterogeneous Chips. In *2018 28th International Conference on Field Programmable Logic and Applications*, FPL '18.

[8] Analog Devices, Inc. Analog devices' processors and dsp.

[9] Renzo Andri, Beatrice Bussolino, Antonio Cipolletta, Lukas Cavigelli, and Zhe Wang. Going Further With Winograd Convolutions: Tap-Wise Quantization for Efficient Inference on 4x4 Tiles. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*, MICRO '22.

[10] Apple. Small chip. Giant leap.

[11] Apple Inc. Apple M1, 2020.

[12] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. Alrescha: A lightweight reconfigurable sparse-computation accelerator. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 249–260, 2020.

[13] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. REVAMP: A Systematic Framework for Heterogeneous CGRA Realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22.

[14] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, 2008.

[15] Saambhavi Baskaran, Mahmut Taylan Kandemir, and Jack Sampson. An architecture interface and offload model for low-overhead, near-data, distributed accelerators. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*, MICRO '22.

[16] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[17] Ajay Brahmakshatriya, Emily Furst, Victor A. Ying, Claire Hsu, Changwan Hong, Max Ruttenberg, Yunming Zhang, Dai Cheol Jung, Dustin Richmond, Michael B. Taylor, Julian Shun, Mark Oskin, Daniel Sanchez, and Saman Amarasinghe. Taming the zoo: The unified graphit compiler framework for novel architectures. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 429–442, 2021.

[18] John Burgess. Rtx on—the nvidia turing gpu. *IEEE Micro*, 40(2):36–44, 2020.

[19] Carl Yang and Aydin Buluc and Yangzihao Wang and John D. Owens. GraphBLAST, 2019.

[20] 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. IEEE, 2016.

[21] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., 2001.

[22] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization*, IISWC '09.

[23] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18.

[24] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS '18.

[25] Xinyu Chen, Yao Chen, Feng Cheng, Hongshi Tan, Bingsheng He, and Weng-Fai Wong. ReGraph: Scaling Graph Processing on HBM-enabled FPGAs with Heterogeneous Pipelines. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*, MICRO '22.

[26] Yujeong Choi and Minsoo Rhu. PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units. In *2020 IEEE International Symposium on High Performance Computer Architecture*, HPCA '20.

[27] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA):123:1–123:30, October 2018.

[28] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[29] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 1989.

[30] 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.

[31] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1), 2011.

[32] Daya S Khudia and Protonu Basu and Summer Deng. Open-sourcing FBGEMM for state-of-the-art server-side inference.

[33] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. Bit-Tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 749–763, New York, NY, USA, 2019. Association for Computing Machinery.

[34] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan. PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 189–202, 2018.

[35] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. Gospa: An energy-efficient high-performance globally optimized sparse convolutional neural network accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1110–1123, 2021.

[36] J. Dong, Z. Cao, T. Zhang, J. Ye, S. Wang, F. Feng, L. Zhao, X. Liu, L. Song, L. Peng, Y. Guo, X. Jiang, L. Tang, Y. Du, Y. Zhang, P. Pan, and Y. Xie. EFLOPS: Algorithm and system co-design for a high performance distributed training platform. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 610–622, 2020.

[37] Jack J Dongarra and Danny C Sorensen. Linear algebra on high performance computers. *Applied mathematics and computation*, 20(1-2):57–88, 1986.

[38] F. Fernandes dos Santos, C. Lunardi, D. Oliveira, F. Libano, and P. Rech. Reliability Evaluation of Mixed-Precision Architectures. In *2019 IEEE International Symposium on High Performance Computer Architecture*, HPCA '19.

[39] 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 '21.

[40] Janick Edinger, Martin Breitbach, Niklas Gabrisch, Dominik Schäfer, Christian Becker, and Amr Rizk. Decentralized Low-Latency Task Scheduling for Ad-Hoc Computing. In *2021 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '21.

[41] Venmugil Elango. Pase: Parallelization Strategies for Efficient DNN Training. In *2021 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '21.

[42] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12.

[43] Yuping Fan, Zhiling Lan, Paul Rich, William Allcock, and Michael E. Papka. Hybrid Workload Scheduling on HPC Systems. In *2022 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '22.

[44] Axel Feldmann and Daniel Sanchez. Spatula: A hardware accelerator for sparse matrix factorization. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 91–104, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. EGEMM-TC: Accelerating Scientific Computing on Tensor Cores with Extended Precision. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21.

[46] Boyuan Feng, Yuke Wang, Tong Geng, Ang Li, and Yufei Ding. APNN-TC: Accelerating Arbitrary Precision Neural Networks on Ampere GPU Tensor Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21.

[47] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE micro*, 28(4):13–27, 2008.

[48] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo. LEO: Scheduling Sensor Inference Algorithms across Heterogeneous Mobile Processors and Network Resources. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16.

[49] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Sparsep: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(1), feb 2022.

[50] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '19. Association for Computing Machinery, 2019.

[51] Google Inc. XLA: Optimizing Compiler for Machine Learning, 2014.

[52] Google LLC. Coral M.2 accelerator datasheet, 2019.

[53] Google LLC. edgetpu compiler, 2020.

[54] Google LLC. Google Pixel 6a.

[55] Xiuxian Guan, Zekai Sun, Shengliang Deng, Xusheng Chen, Shixiong Zhao, Zongyuan Zhang, Tianyang Duan, Yuexuan Wang, Chenshu Wu, Yong Cui, Libo Zhang, Yanjun Wu, Rui Wang, and Heming Cui. ROG: A High Performance and Robust Distributed Training System for Robotic IoT. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*, MICRO '22.

[56] Cong Guo, Chen Zhang, Jingwen Leng, Zihan Liu, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. ANT: Exploiting Adaptive Numerical Data Type for Low-bit Deep Neural Network Quantization. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*, MICRO '22.

[57] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '18.

[58] 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.

[59] Chaoyang He, Murali Annavaram, and Salman Avestimehr. Group Knowledge Transfer: Federated Learning of Large CNNs at the Edge. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20.

[60] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. Sparse-tpu: adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[61] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 319–333, New York, NY, USA, 2019. Association for Computing Machinery.

[62] Nhut-Minh Ho and Weng-Fai Wong. Exploiting half precision arithmetic in Nvidia GPUs. In *2017 IEEE High Performance Extreme Computing Conference*, HPEC '17.

[63] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. Spaghetti: Streaming accelerators for highly sparse gemm on fpgas. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 84–96, 2021.

[64] 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.

[65] 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.

[66] Xueyu Hou, Yongjie Guan, Tao Han, and Ning Zhang. DistrEdge: Speeding up Convolutional Neural Network Inference on Distributed Edge Devices. In *2022 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '22.

[67] Kuan-Chieh Hsu and Hung-Wei Tseng. Accelerating Applications Using Edge Tensor Processing Units. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21.

[68] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. The sparse abstract machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 710–726, New York, NY, USA, 2023. Association for Computing Machinery.

[69] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. TCUDB: Accelerating Database with Tensor Processors. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22.

[70] Yu-Ching Hu, Murtuza Taher Lokhandwala, Te I., and Hung-Wei Tseng. Dynamic Multi-Resolution Data Storage. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '19.

[71] Zhiming Hu, Ahmad Bisher Tarakji, Vishal Raheja, Caleb Phillips, Teng Wang, and Iqbal Mohomed. DeepHome: Distributed Inference with Heterogeneous Devices in the Edge. In *The 3rd International Workshop on Deep Learning for Mobile Systems and Applications*, EMDL '19.

[72] Soojin Hwang, Daehyeon Baek, Jongse Park, and Jaehyuk Huh. Cerberus: Triple mode acceleration of sparse matrix and vector multiplication. *ACM Trans. Archit. Code Optim.*, 21(2), may 2024.

[73] Thomas B Jablin, Prakash Prabhu, James A Jablin, Nick P Johnson, Stephen R Beard, and David I August. Automatic cpu-gpu communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 142–151, 2011.

[74] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '18.

[75] Geonhwa Jeong, Sana Damani, Abhimanyu Rajeshkumar Bambhaniya, Eric Qin, Christopher J. Hughes, Sreenivas Subramoney, Hyesoon Kim, and Tushar Krishna. Vegeta: Vertically-integrated extensions for sparse/dense gemm tile acceleration on cpus. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 259–272, 2023.

[76] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems*, MLSys '19.

[77] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20.

[78] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*, ISCA '21.

[79] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings, 2023.

[80] 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. In *Communications of the ACM*, 2020.

[81] 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, pages 1–12, New York, NY, USA, 2017. ACM.

[82] Rabimba Karanjai, Sangwon Shin, Xinxin Fan, Lin Chen, Tianwei Zhang, Taeweon Suh, Weidong Shi, and Lei Xu. Tpu as cryptographic accelerator, 2023.

[83] Liu Ke, Udit Gupta, Mark Hempstead, Carole-Jean Wu, Hsien-Hsin S. Lee, and Xuan Zhang. Hercules: Heterogeneity-Aware Inference Serving for At-Scale Personalized Recommendation. In *2022 IEEE International Symposium on High-Performance Computer Architecture*, HPCA '22.

[84] Hamidreza Khaleghzadeh, Ravi Reddy Manumachu, and Alexey Lastovetsky. A Hierarchical Data-Partitioning Algorithm for Performance Optimization of Data-Parallel Applications on Heterogeneous Multi-Accelerator NUMA Nodes. *IEEE Access*, 2020.

[85] Khronos Group. OpenCL.

[86] Daya S Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke. Rumba: An online quality management system for approximate computing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture*, ISCA '15.

[87] H. Kim, J. Sim, Y. Choi, and L. Kim. NAND-Net: Minimizing computational complexity of in-memory processing for binary neural networks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 661–673, 2019.

[88] Sungil Kim and Heeyoung Kim. A new metric of absolute percentage error for intermittent demand forecasts. *International Journal of Forecasting*, 2016.

[89] Young Geun Kim and Carole-Jean Wu. AutoFL: Enabling Heterogeneity-Aware Energy Efficient Federated Learning. In *54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21.

[90] Anish Krishnakumar, Samet E. Arda, A. Alper Goksoy, Sumit K. Mandal, Umit Y. Ogras, Anderson L. Sartor, and Radu Marculescu. Runtime Task Scheduling Using Imitation Learning for Heterogeneous Many-Core Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[91] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous Dataflow Accelerators for Multi-DNN Workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture*, HPCA '21.

[92] Matthias Langer, Zhen He, Wenny Rahayu, and Yanbo Xue. Distributed Training of Deep Learning Models: A Taxonomic Perspective. In *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[93] Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16.

[94] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[95] 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 '21.

[96] Shikai Li, Sunghyun Park, and Scott Mahlke. Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18.

[97] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning. In *Annual Conference on Neural Information Processing Systems*, NIPS '21.

[98] Ji Lin, Wei-Ming Chen, John Cohn, Chuang Gan, and Song Han. MCUNet: Tiny Deep Learning on IoT Devices. In *Annual Conference on Neural Information Processing Systems*, NIPS '20.

[99] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-Device Training Under 256KB Memory. In *Annual Conference on Neural Information Processing Systems*, NIPS '21.

[100] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. VELTAI: Rowards High-Performance Multi-Tenant Deep Learning Services via Adaptive Compilation and Scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22.

[101] Atieh Lotfi, Abbas Rahimi, Hadi Esmaeilzadeh, and Rajesh K Gupta. SqueezCL: Squeezing OpenCL kernels for approximate computing on contemporary GPUs. In *Workshop on Approximate Computing*, 2015.

[102] H. Lu, M. Matheson, V. Oles, A. Ellis, W. Joubert, and F. Wang. Climbing the summit and pushing the frontier of mixed precision benchmarks at extreme scale. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*, pages 1123–1137, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

[103] 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 '20.

[104] 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 '21.

[105] Chao Ma, Thibault Marin, TJ Lu, Yi fan Chen, and Yue Zhuo. Accelerating mri reconstruction on tpus. 2020.

[106] Mostafa Mahmoud, Isak Edo, Ali Hadi Zadeh, Omar Mohamed Awad, Gennady Pekhimenko, Jorge Albericio, and Andreas Moshovos. Tensordash: Exploiting sparsity to accelerate deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 781–795, 2020.

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

[108] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. In *International Conference on Learning Representations*, ICLR '18.

[109] Mehryar Mohri. Semiring Frameworks and Algorithms for Shortest-Distance Problems. *Journal of Automata, Languages and Combinatorics*, 2002.

[110] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA '15.

[111] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 252–265, New York, NY, USA, 2023. Association for Computing Machinery.

[112] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317, 2011.

[113] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20.

[114] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20.

[115] Nandeeka Nayak, Toluwanimi O. Odemuyiwa, Shubham Ugare, Christopher Fletcher, Michael Pellauer, and Joel Emer. Teaal: A declarative framework for modeling sparse tensor accelerators. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 1255–1270, New York, NY, USA, 2023. Association for Computing Machinery.

[116] Duy Thanh Nguyen, Tuan Nghia Nguyen, Hyun Kim, and Hyuk-Jae Lee. A high-throughput and power-efficient fpga implementation of yolo cnn for object detection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(8):1861–1873, 2019.

[117] Daniel Nichols, Aniruddha Marathe, Kathleen Shoga, Todd Gamblin, and Abhinav Bhatele. Resource Utilization Aware Job Scheduling to Mitigate Performance Variability. In *2022 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '22.

[118] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21.

[119] NVIDIA. cuBLAS, 2019.

[120] NVIDIA. cusparse, 2024.

[121] NVIDIA Corporation. CUDA C programming guide v6.0, 2014.

[122] NVIDIA T4 TENSOR CORE GPU, 2019.

[123] NVIDIA A100 Tensor Core GPU Architecture, 2020.

[124] NXP Semiconductors N.V. NXP MSC8154E Quad-Core DSP with Security.

[125] Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal C. Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. Accelerating sparse data orchestration via dynamic reflexive tiling. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 18–32, New York, NY, USA, 2023. Association for Computing Machinery.

[126] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[127] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736, 2018.

[128] J. F. Palmer. The intel 8087 numeric data processor. In *Managing Requirements Knowledge, International Workshop on*, page 887, Los Alamitos, CA, USA, may 1980. IEEE Computer Society.

[129] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 27–40, New York, NY, USA, 2017. Association for Computing Machinery.

[130] E. Park, D. Kim, and S. Yoo. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 688–698, 2018.

[131] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 688–698, 2018.

[132] Alberto Parravicini, Arnaud Delamare, Marco Arnaboldi, and Marco D. Santambrogio. DAG-based Scheduling with Resource Sharing for Multi-task Applications in a Polyglot GPU Runtime. In *2021 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '21.

[133] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*. NIPS '19.

[134] Michael Pellauer, Jason Clemons, Vignesh Balaji, Neal Crago, Aamer Jaleel, Donghyuk Lee, Mike O'Connor, Anghsuman Parashar, Sean Treichler, Po-An Tsai, Stephen W. Keckler, and Joel S. Emer. Symphony: Orchestrating sparse and dense tensors with hierarchical heterogeneous processing. *ACM Trans. Comput. Syst.*, 41(1–4), dec 2023.

[135] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, June 2014.

[136] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects

for DNN training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70, 2020.

[137] QNAP. QM2 Expansion Card (Add M.2 SSD Slots), 2020.

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

[139] Kiran Ranganath, Joshua D. Suetterlein, Joseph B. Manzano, Shuaiwen Leon Song, and Daniel Wong. MAPA: Multi-Accelerator Pattern Allocation Policy for Multi-Tenant GPU Servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21.

[140] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. Sparsecore: stream isa and processor specialization for sparse computation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 186–199, New York, NY, USA, 2022. Association for Computing Machinery.

[141] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 78–91. IEEE, 2018.

[142] D. Richins, D. Doshi, M. Blackmore, A. Thulaseedharan Nair, N. Pathapati, A. Patel, B. Daguman, D. Dobrijalowski, R. Illikkal, K. Long, D. Zimmerman, and V. Janapa Reddi. Missing the forest for the trees: End-to-end AI application performance in edge data centers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 515–528, 2020.

[143] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. Capstan: A vector rda for sparsity. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1022–1035, New York, NY, USA, 2021. Association for Computing Machinery.

[144] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[145] Shane Ryoo, Christopher I Rodrigues, Sam S Stone, Sara S Baghsorkhi, Sain-Zee Ueng, John A Stratton, and Wen-mei W Hwu. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, 2008.

[146] Stanislav G. Sedukhin and Marcin Paprzycki. Generalizing Matrix Multiplication for Efficient Computations on Modern Computers. In *Parallel Processing and Applied Mathematics*, 2012.

[147] Hashim Sharif, Prakalp Srivastava, Muhammad Huzaifa, Maria Kotsifakou, Keyur Joshi, Yasmin Sarita, Nathan Zhao, Vikram S. Adve, Sasa Misailovic, and Sarita Adve. Approxhpvm: A portable compiler ir for accuracy-aware optimizations. 3(OOPSLA), 2019.

[148] Hashim Sharif, Yifan Zhao, Maria Kotsifakou, Akash Kothari, Ben Schreiber, Elizabeth Wang, Yasmin Sarita, Nathan Zhao, Keyur Joshi, Vikram S. Adve, Sasa Misailovic, and Sarita Adve. Approximating APSP without Scaling: Equivalence of Approximate Min-plus and Exact Min-Max. In *Symposium on Principles and Practice of Parallel Programming*, PPoPP 2021, 2021.

[149] S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos. Laconic deep learning inference acceleration. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 304–317, 2019.

[150] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh. Bit Fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775, 2018.

[151] Jiuchen Shi, Jiawen Wang, Kaihua Fu, Quan Chen, Deze Zeng, and Minyi Guo. QoS-awareness of Microservices with Excessive Loads via Inter-Datacenter Scheduling. In *2022 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '22.

[152] Siddharth Singh and Abhinav Bhatele. AxoNN: An asynchronous, message-driven parallel framework for extreme-scale deep learning. In *2022 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '22.

[153] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen. HyPar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 56–68, 2019.

[154] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture*, HPCA, '20.

[155] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '22, page 65–77, New York, NY, USA, 2022. Association for Computing Machinery.

[156] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li. Prediction based execution on deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 752–763, 2018.

[157] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '20, pages 689–702. IEEE Press, 2020.

[158] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780, 2020.

[159] Pierre Stock, Angela Fan, Benjamin Graham, Edouard Grave, Rémi Gribonval, Herve Jegou, and Armand Joulin. Training with Quantization Noise for Extreme Model Compression. In *International Conference on Learning Representations*, ICLR '21.

[160] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi (Viji) Srinivasan, and Kailash Gopalakrishnan. Ultra-Low Precision 4-bit Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems*, NIPS '20.

[161] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer. How to evaluate deep neural network processors: Tops/w (alone) considered harmful. *IEEE Solid-State Circuits Magazine*, 12(3):28–41, 2020.

[162] Tuan Ta, Khalid Al-Hawaj, Nick Cebry, Yanghui Ou, Eric Hall, Courtney Golden, and Christopher Batten. big.VLITTLE: On-Demand Data-Parallel Acceleration for Mobile Systems on Chip. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*, MICRO '22.

[163] Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. Stitch: Fusible Heterogeneous Accelerators Enmeshed with Many-Core Architecture for Wearables. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, ISCA '18.

[164] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, CVPR '19.

[165] Chunlin Tian, Li Li, Zhan Shi, Jun Wang, and ChengZhong Xu. HARMONY: Heterogeneity-Aware Hierarchical Management for Federated Learning System. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*, MICRO '22.

[166] Han D. Tran, Milinda Fernando, Kumar Saurabh, Baskar Ganapathysubramanian, Robert M. Kirby, and Hari Sundar. A scalable adaptive-matrix spmv for heterogeneous architectures. In *2022 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '22.

[167] Jack Turner, Elliot J. Crowley, and Michael F. P. O'Boyle. Neural Architecture Search as Program Transformation Exploration. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21.

[168] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE, 2008.

[169] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, CVPR '19.

[170] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training Deep Neural Networks with 8-bit Floating Point Numbers. In *Advances in Neural Information Processing Systems*, NIPS '18.

[171] Shuo Wang, Yun Liang, and Wei Zhang. Poly: Efficient Heterogeneous System and Application Management for Interactive Applications. In *2019 IEEE International Symposium on High Performance Computer Architecture*, HPCA '19.

[172] Ting Wang, Qian Zhang, and Qiang Xu. ApproxQA: A unified quality assurance framework for approximate computing. In *Design, Automation and Test in Europe Conference and Exhibition*, DATE '17.

[173] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. Dual-side sparse tensor core. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1083–1095, 2021.

[174] Joel Wolfrath, Nikhil Sreekumar, Dhruv Kumar, Yuanli Wang, and Abhishek Chandra. HACCS: Heterogeneity-Aware Clustered Client Selection for Accelerated Federated Learning. In *2022 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '22.

[175] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proc. VLDB Endow.*, 7(11):963–974, July 2014.

[176] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. Sparseloop: An analytical, energy-focused design space exploration methodology for sparse tensor accelerators. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 232–234, 2021.

[177] Yecheng Xiang and Hyoseung Kim. Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference. In *2019 IEEE Real-Time Systems Symposium*, RTSS '19.

[178] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. HASCO: Towards Agile HArdware and Software CO-design for Tensor Computation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*, ISCA '21.

[179] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 570–583, 2021.

[180] Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Sasa Misailovic, and Saurabh Bagchi. VideoChef: Efficient Approximation for Streaming Video Processing Pipelines. In *2018 USENIX Annual Technical Conference*, USENIX ATC '18.

[181] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A gpgpu compiler for memory optimization and parallelism management. *ACM Sigplan Notices*, 45(6):86–97, 2010.

[182] Zichao Yang, Heng Wu, Yuanjia Xu, Yuewen Wu, Hua Zhong, and Wenbo Zhang. Hydra: Deadline-aware and Efficiency-oriented Scheduling for Deep Learning Jobs on Heterogeneous GPUs. *IEEE Transactions on Computers*, 2023.

[183] 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, April 2017.

[184] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. Neural acceleration for GPU throughput processors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 482–493, New York, NY, USA, 2015. ACM.

[185] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke. Scalpel: Customizing DNN pruning to the underlying hardware parallelism. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 548–560, 2017.

[186] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. *ACM SIGPLAN Notices*, 46(3):369–380, 2011.

[187] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. Gamma: leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 687–701, New York, NY, USA, 2021. Association for Computing Machinery.

[188] Minjia Zhang, Zehua Hu, and Mingqin Li. DUET: A Compiler-Runtime Subgraph Scheduling Approach for Tensor Programs on a Coupled CPU-GPU Architecture. In *2021 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '21.

[189] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-X: an accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[190] Yunan Zhang, Po-An Tsai, and Hung-Wei Tseng. SIMD2: A Generalized Matrix Instruction Set for Accelerating Tensor Computation beyond GEMM. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22.

[191] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. SpArch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–274. IEEE, 2020.

[192] Zhang Xianyi and Martin Kroeker. OpenBLAS: An optimized BLAS library, 2021.

[193] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. AMOS: Enabling Automatic Mapping for Tensor Computations On Spatial Accelerators with Hardware Abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22.

[194] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 859–873, New York, NY, USA, 2020. Association for Computing Machinery.

[195] Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. Adaptive Parallel Execution of Deep Neural Networks on Heterogeneous Edge Devices. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19.

[196] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 359–371, New York, NY, USA, 2019. Association for Computing Machinery.

[197] Wentao Zhu, Can Zhao, Wenqi Li, Holger R. Roth, Ziyue Xu, and Daguang Xu. LAMP: Large Deep Nets with Automated Model Parallelism for Image Segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, MICCAI '20.

[198] 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.