



Simultaneous and Heterogenous Multithreading

Kuan-Chieh Hsu

University of California, Riverside
Riverside, California, USA
khsu037@ucr.edu

Hung-Wei Tseng

University of California, Riverside
Riverside, California, USA
htseng@ucr.edu

ABSTRACT

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 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.

ACM Reference Format:

Kuan-Chieh Hsu and Hung-Wei Tseng. 2023. Simultaneous and Heterogenous Multithreading. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*, October 28–November 01, 2023, Toronto, ON, Canada. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3613424.3614285>

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)[76, 77] or Ray Tracing Cores (RT Cores)[12] on NVIDIA GPUs, Tensor Processing Units (TPUs) on Google Cloud servers [46, 48, 49], Neural Engines on Apple’s iPhones [8], 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 [20, 22, 25, 37, 39, 40, 59, 67, 68], 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, 74, 88].

This paper presents SHMT, simultaneous and heterogeneous multithreading, to evaluate the performance and tackle the challenges of simultaneously using heterogeneous computing resources. 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 1 illustrates the advantage of SHMT against the conventional execution model. Figure 1 assumes a program containing five primary functions, A to E, and five computing resources, including CPUs, GPUs, and three accelerators. Figure 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 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 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



This work is licensed under a Creative Commons Attribution International 4.0 License.

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0329-4/23/10.

<https://doi.org/10.1145/3613424.3614285>

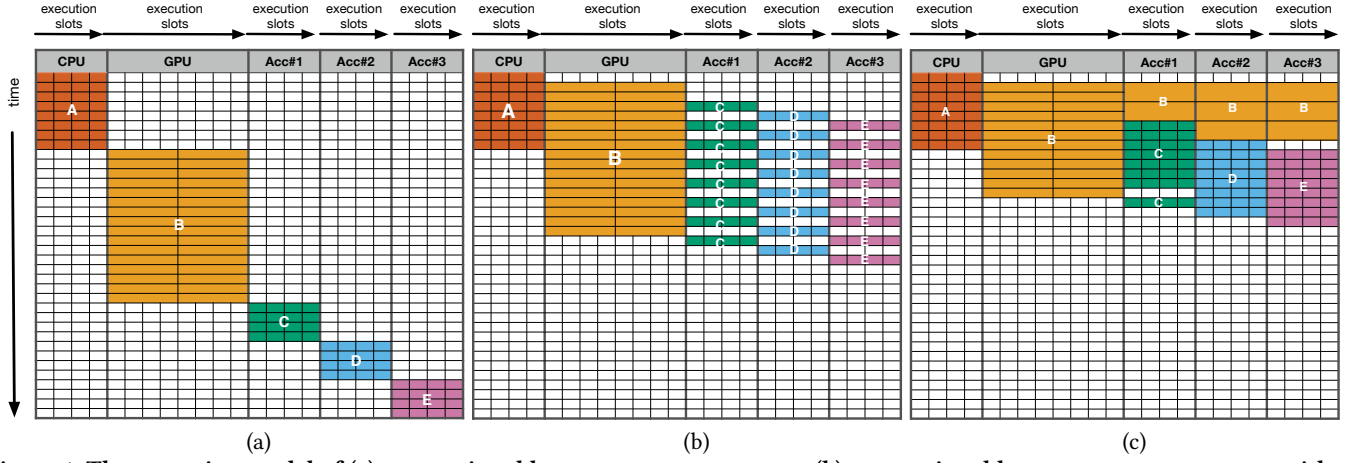


Figure 1: The execution model of (a) conventional heterogeneous computers (b) conventional heterogeneous computers with software pipelining, and (c) SHMT.

and divide equivalent operations and data on different computing resources. Second, unlike traditional programming systems that delegate each code region to a single type of hardware, SHMT must be able to coordinate the execution on heterogeneous hardware efficiently. Finally, and probably the most challenging, 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 $3.92\times$ speedup and $2.07\times$ on average. With our proposed quality assurance mechanisms, SHMT still achieves $1.95\times$ 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.

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.

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) [36] 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 [76, 77] 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 [6, 78]. 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 [12].

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.

2.2.1 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. Famous 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 [20], TCUDB [40], and RQTPU [37] 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 the use of matrix multiplication in TPUs to accelerate Fourier Transform [22, 68] and facilitate MRI image reconstruction [67]. GPTPU [39] reverse-engineered the Edge TPU compiler and built a tensor operator-based programming framework for Edge TPU to accelerate Rodinia benchmark applications [14].

2.2.2 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 2.2.1, this approach requires less programming language or ISA support in exposing the internal hardware features to programmers.

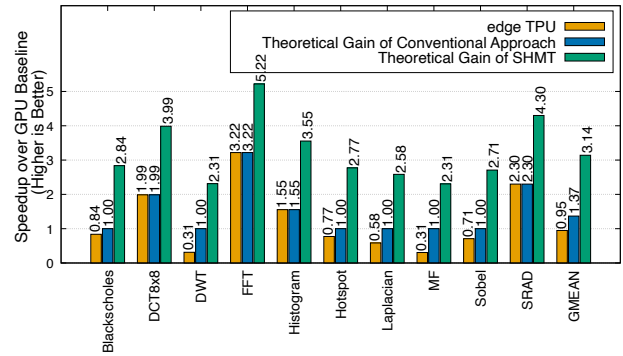


Figure 2: The potential speedup of SHMT (and Edge TPU) relative to GPU-only implementation

Neural Processing Units (NPU) [3, 25, 66, 71] follow this route to solve general-purpose problems using NN accelerators. NPUs leverage universal the approximation theorem [19] 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 [111] 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.

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 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 conventional approaches that delegate kernels to the best-performing accelerator is 1.37x.

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 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 3.14x.

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

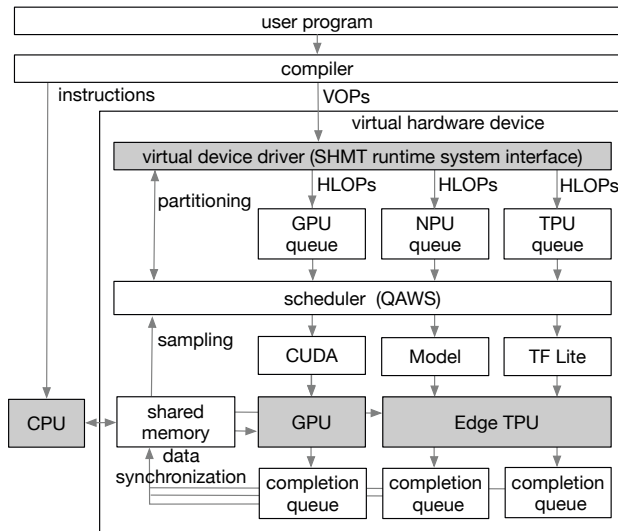


Figure 3: SHMT overview

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 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.

3.1 Overview

Figure 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 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 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 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 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.

3.2 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.

3.2.1 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 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

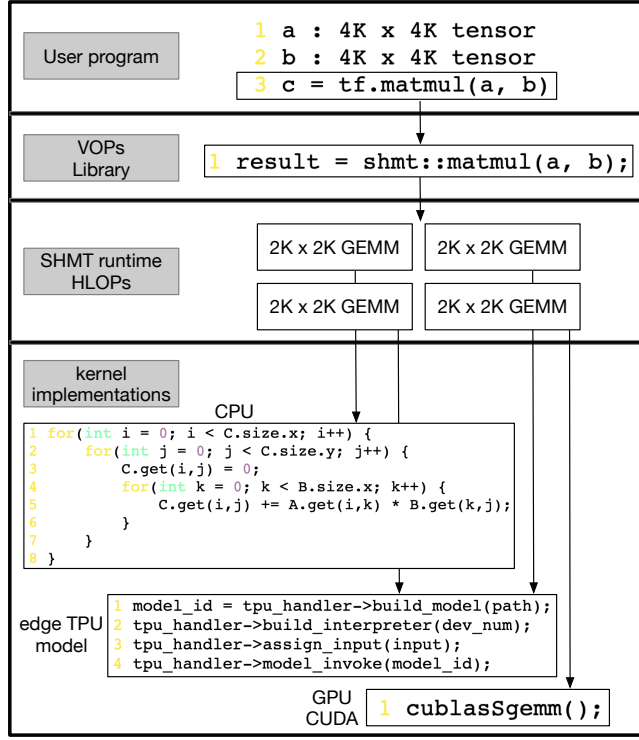


Figure 4: The SHMT's programming model

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

implemented supported computation in hardware accelerators. In our current list, these VOPs can either use an element-wise vector processing model or a tile-wise matrix processing model to partition and parallelize the computation without violating the correctness.

3.2.2 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 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.

3.3.1 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.

3.3.2 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.

3.4 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.

3.5 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

Algorithm 1 Device Limitation

Input: P , $limits$

```

1:  $N \leftarrow |P|$ 
2:  $M \leftarrow |limits|$ 
3:  $|Q| \leftarrow N$ 
4: for  $i \leftarrow 0$  to  $N$  do
5:    $s \leftarrow sampling\_module(P_i)$ 
6:    $Q_i \leftarrow M - 1$  ▷ your default choice
7:   for  $j \leftarrow 0$  to  $M$  do
8:     if  $s < limits_j[0]$  then
9:        $Q_i \leftarrow limits_j[1]$ 
10:    break
11: return  $Q$ 

```

Algorithm 2 Top-K Criticality

Input: P , K , W

```

1:  $N \leftarrow |P|$ 
2:  $|Q| \leftarrow N$ 
3:  $|window[]| \leftarrow W$ 
4: for  $i \leftarrow 0$  to  $N$  do
5:    $window[i \cdot W] \leftarrow sampling\_module(P_i)$ 
6:   if  $(i \cdot W == W - 1)$  or  $(i == N - 1)$  then
7:     sort(window)
8:     for  $j \leftarrow 0$  to  $W$  do
9:        $Q_i \leftarrow (j < K) ? 0 : 1$ 
10: return  $Q$ 

```

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 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 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:

Algorithm 3 The striding sampling**Input:** D, N, s

```

1:  $|S| \leftarrow N$ 
2: for  $i \leftarrow 0$  to  $N$  do
3:    $S_i \leftarrow D[i * s]$ 
4: return  $S$ 

```

Algorithm 4 The uniform random sampling**Input:** D, N

```

1:  $|S| \leftarrow N$ 
2: for  $i \leftarrow 0$  to  $N$  do
3:    $S_i \leftarrow D[random()]$ 
4: return  $S$ 

```

Algorithm 5 The reduction sampling**Input:** D, s

```

1:  $S \leftarrow []$ 
2:  $\mathbf{dims} \leftarrow \text{dimension}(D)$ 
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:      $S.append(D[i0, i1, ...])$ 
7: return  $S$ 

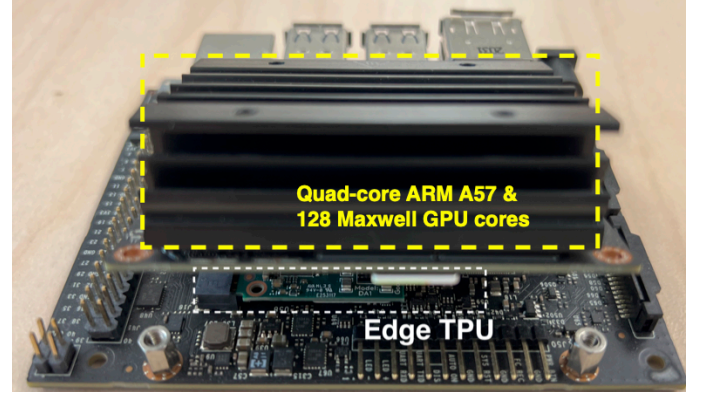
```

(1) device-dependent limitation and (2) application-dependent top-K criticality, respectively. In Algorithm 1, $\mathbf{P}$ is an array of input partitions, and $\mathbf{limits}$ is an array of paired numbers - the limitation number of a device and the index of the corresponding device queue. $\mathbf{limits}$ is sorted by the first index in descending order. In Algorithm 2, the two additional inputs other than $\mathbf{P}$, K , and W , are the threshold value of top-K and window size W , respectively. Any given K has to be smaller than the W . And the result array $\mathbf{Q}$ 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 [58]. 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 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

**Figure 5:** The SHMT prototype platform

and 5 is a step size, and the N for Algorithms 3 and 4 is the desired number of samplings.

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.

4.1 The system assembly

This paper built an exemplary SHMT prototype using NVIDIA's Jetson Nano and Google's Edge TPU. Figure 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 [31], 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) [47, 77], 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.

4.2 NPU implementations

Edge TPU can either serve as a matrix function accelerator as Section 2.2.1 describes or implement an NPU as Section 2.2.2 describes. In the case of using Edge TPU as matrix accelerators, we leverage existing library to implement corresponding HLOPs [39].

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* [30].
- (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.

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 4 presents.

5.1 Benchmark applications

Table 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.

Benchmark	Category	Baseline Implementation
Blackscholes	Finance	CUDA Examples [74]
DCT8x8	Image Processing	CUDA Examples [74]
DWT	Signal Processing	Rodinia 3.1 [14]
FFT	Signal Processing	CUDA Examples [74]
Histogram	Statistical	Opencv 4.5.5 [11]
Hotspot	Physics Simulation	Rodinia 3.1 [14]
Laplacian	Image Processing	Opencv 4.5.5 [11]
Mean Filter(MF)	Image Processing	Opencv 4.5.5 [11]
Sobel	Image Processing	Opencv 4.5.5 [11]
SRAD	Medical Imaging	CUDA Examples [74]

Table 2: Table of benchmarks

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 6 illustrates the speedup of SHMT with various scheduling policies. In Figure 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 Limitation** or (2) **Top-K** methods, and *Y* stands for the sampling method, either (1) **Stridding**, (2) **Uniform random sampling** or (3) **Reduction**.

We also include two policies that do not consider the quality of results, even distribution, and 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 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. 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 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 [58], Figure 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.

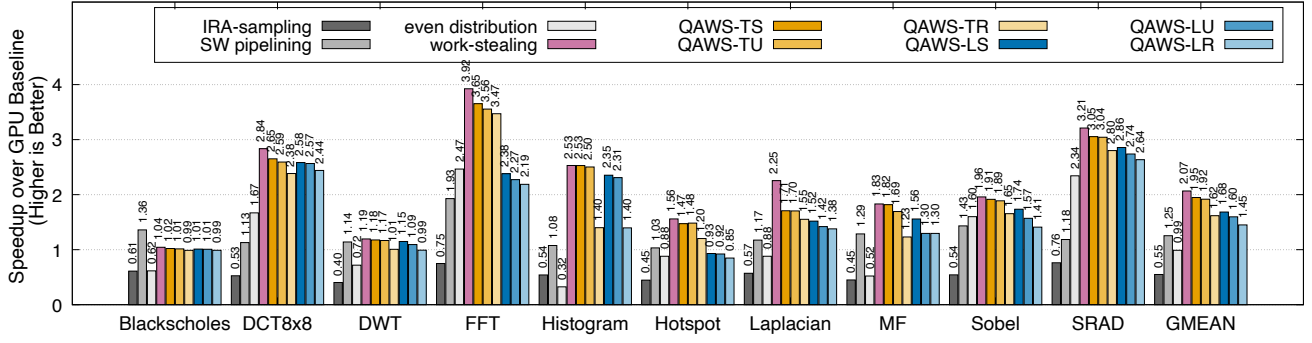


Figure 6: The application speedup relative to the baseline GPU implementations

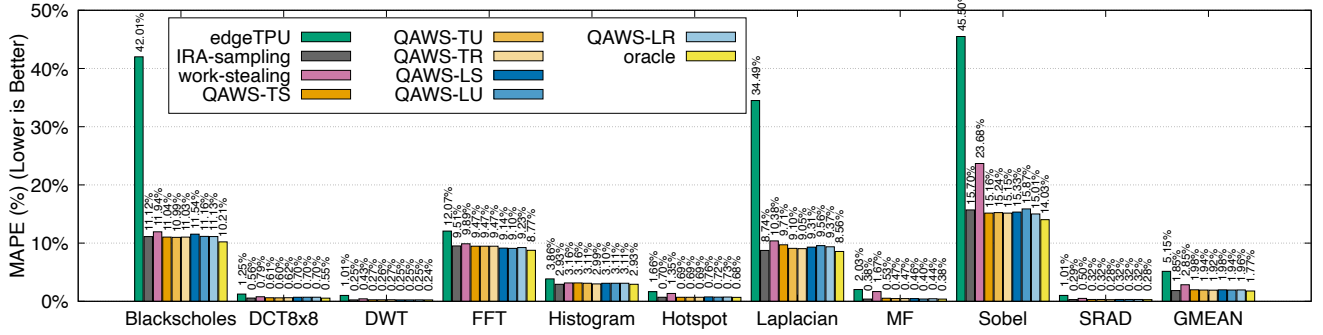


Figure 7: The Mean Absolute Percentage Errors (MAPEs) for SHMT applications

Figure 6 also includes the performance of optimized GPU implementations with software pipelining as another reference design. Software pipeline can only achieve 1.25 $\times$ speedup. For compute-intensive workloads, software pipelining cannot compete with SHMT. Software pipelining is effective for Blackscholes 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.

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

Figure 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 [53].

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 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.

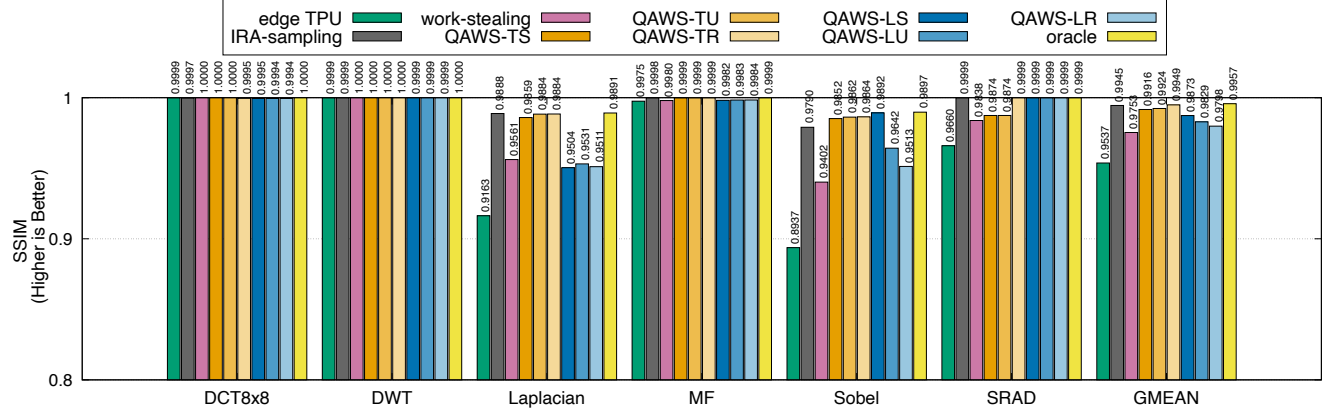


Figure 8: The Structural Similarity Index Measures (SSIMs) for image-related SHMT applications

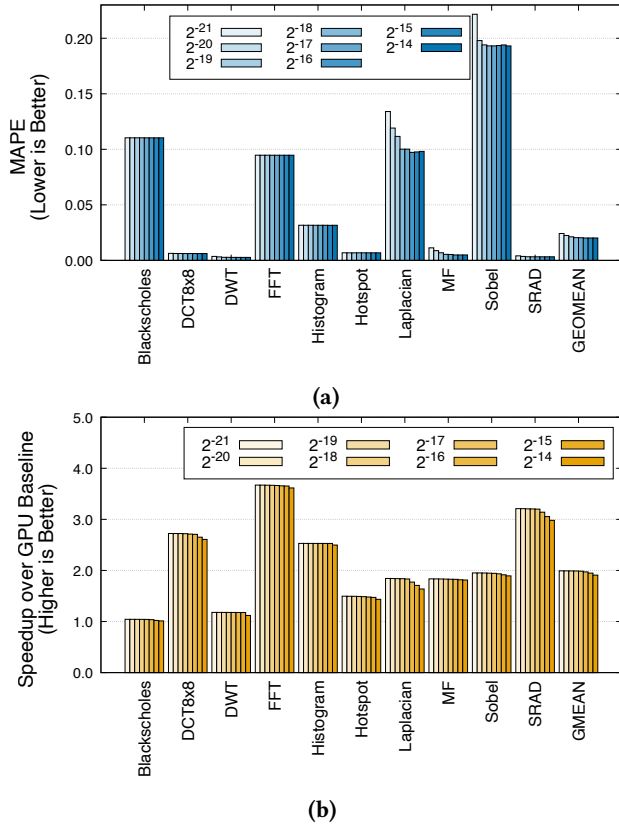


Figure 9: (a) Quality v.s. QAWS sampling rates, (b) Speedup v.s. QAWS sampling rates

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.

5.4 QAWS sampling rate

The number of samples during each sampling phase is another parameter that helps optimize the sampling overhead. Figure 9(a) and Figure 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.

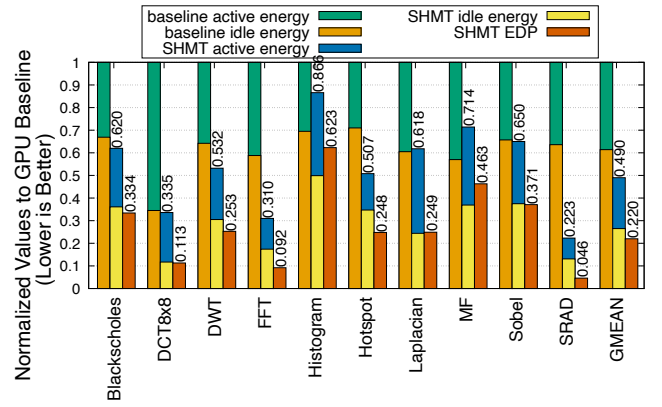


Figure 10: Energy consumption and energy-delay products (EDP)

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.

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 10 reports the breakdown of energy consumption of both GPU baseline and SHMT with *QAWS-TS*. The same figure also shows the relative 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,

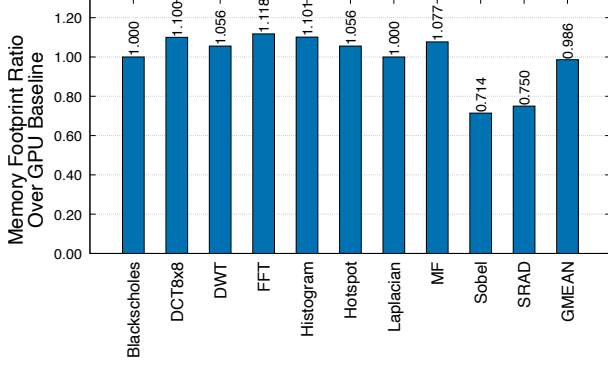


Figure 11: SHMT's Memory Footprint (Normalized to GPU Baseline)

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: Communication Overhead

on average, the 51.0% energy reduction of SHMT with QAWS-TS comes from the 1.95 $\times$ speedup that reduces the period consuming the power with 5.23 Watt at peak.

5.6 Memory and communication overhead

Figure 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 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 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 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.

5.7 Discussion on SHMT's limitation

Figure 12 shows the speedups of SHMT under QAWS-TS variation when problem sizes of benchmarks vary. Within the tested problem

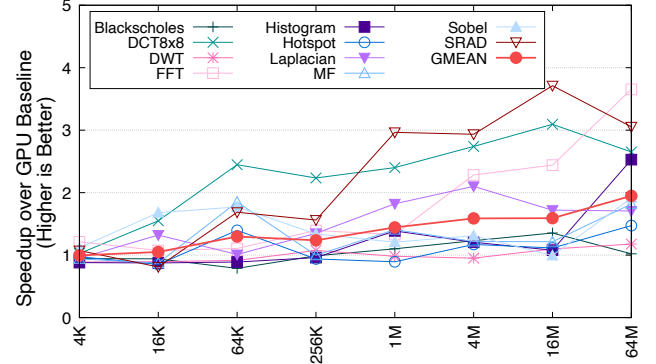


Figure 12: Speedup v.s. problem sizes

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 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 [70, 82, 106]) that makes SHMT easy to exploit.

6 RELATED WORK

Existing runtime for parallel programming on heterogeneous systems. Popular domain-specific languages, including TensorFlow [1] and Pytorch [80], 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 [65], or model-level optimizations like TVM [15] and AutoTVM [16] 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 [88] 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 [13] 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 [5, 18, 24, 50, 51, 64, 72, 73, 79, 81, 86, 104, 105, 110]. Some works extend the same method to computer clusters such as distributed deep learning training/inferencing [26, 32, 38, 42, 45, 57], federated learning [35, 54, 93, 100], decentralized ad-hoc computing [23], inter-datacenter scheduling [83], and scalable computing on supercomputer environments [84, 94]. Although 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 [10, 17, 72, 85, 90, 107] but still falling into the same type. HDA [56] 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 [102] 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 [9, 55, 91, 98]. 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 [44, 108, 109], (2) pipelining [29, 101], (3) operation fusing [2, 75], (4) neural architecture searching (NAS) [61–63, 92, 95], and (5) model quantization/compression [4, 7, 28, 33, 43, 87, 89, 96, 97]. Essentially, these techniques re-consider 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 [4, 33, 96], (2) providing numerical composition solutions to increase resulting precision such as iterative refinement [34] and extended precision [27], or (3) performing mixed-precision computation [21, 52, 69, 99, 103] or providing multi-resolution data [41] to adjust overall required quality according to needs. Existing approximated techniques include loop perforation [60] and numerical approximation. Another example is IRA [58] 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. Conventional 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.

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 $1.95\times$ speedup and 51.0% energy reduction by enabling simultaneous and heterogeneous execution of architectures compared to GPU baseline.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments. This work was sponsored by the two National Science Foundation (NSF) awards, CNS-2007124 and CNS-2231877. This work was also supported by Intel Corporation and new faculty start-up funds from University of California, Riverside.

A ARTIFACT APPENDIX

A.1 Abstract

This document describes the artifact of "Simultaneous and Heterogenous Multithreading" and the process of reproducing the experimental results in this paper. To run the benchmarks that this paper evaluates, the evaluator must have a computer equipped with (1) NVIDIA's GPU, (2) Google's Edge TPU, and (3) capable of running a Linux distribution supporting the software stacks for the GPU and Edge TPU. To build a prototype virtual hardware device that supports SHMT in hardware, the evaluator must also have a prototype similar to Jetson Nano platform where the platform contains (1) a Cortex-A57 ARM processor, (2) a 128-core Maxwell NVIDIA GPU that is capable of running the GPU implementation of HLOP kernels, and (3) a Edge TPU that is capable of running the NPU implementation of HLOP kernels.

A.2 Artifact check-list (meta-information)

- **Program:** Blackscholes [74], Discrete Cosine Transform (DCT8x8) [74], Discrete Wavelet Transform (DWT) [14], Fast Fourier Transform (FFT) [74], Histogram [11], Hotspot [14], Laplacian [11], Mean Filter(MF) [11], Sobel [11], Speckle Reducing Anisotropic Diffusion (SRAD) [74]
- **Compilation:** cmake 3.10, gcc 7.5.0
- **Data set:** Synthetic datasets from each program's dataset generator.
- **Run-time environment:** Ubuntu 18.04 with NVIDIA's customized 4.9.253-tegra kernel, CUDA 10.2, nvidia-docker 20.10.7
- **Hardware:** A Jetson Nano (4 GB ram) platform equipped with Edge TPU
- **Execution:** To reduce the disturbance from another workload, we recommend running experiments with a sole user.
- **Metrics:** End-to-end latency (second), Mean Absolute Percentage Error (MAPE)
- **Output:** Each benchmark program will display its execution result through the console or log files.
- **How much time is needed to complete experiments (approximately)?:** 1 - 2 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** We will be using an MIT license for our code.
- **Data licenses (if publicly available)?:** The datasets are publicly available through their original licensing terms.
- **Archived (provide DOI)?:** <https://zenodo.org/record/8210452>

A.3 Description

A.3.1 How to access. We archive the source code and workloads at <https://zenodo.org/record/8210452>. For the latest version, the user can access our GitHub page: <https://github.com/escalab/SHMT>

A.3.2 Hardware dependencies. To build the SHMT prototype system, the user will need the hardware components and the construction guide as Section 4.1 mentions. In summary, the experimental Jetson Nano-based platform contains the following hardware components.

- **Processor:** Cortex-A57 ARM processor
- **DRAM:** 4 GB 64-bit LPDDR4
- **GPU:** 128-core Maxwell NVIDIA GPU
- **Edge TPU:** M.2 Accelerator A+E key

A.3.3 Software dependencies. The SHMT artifact relies on the following software components.

- CUDA 10.2
- nvidia-docker 20.10.7

Since the SHMT artifact leverages nvidia-docker to avoid manually installing many software dependencies, the following software dependencies are required if nvidia-docker is not used during compilation.

- cmake 3.10 or newer
- gcc 7.5.0
- Opencv 4.5.5
- CUDA 10.2
- OpenMP

A.3.4 Models. Since this work implements the Edge TPU kernels using NPU [25] as Section 4.2 mentions, the user can refer to NPU [25] for how to generate neural-network-based kernel models. To reduce the workflow time of preparing the Edge TPU kernels, we prepare pre-trained kernel models under `models/` directory for this particular experiment.

The user can refer to `src/Python/generate_kernel_model.py` for more details about the pre-train workflow.

A.4 Installation

Before installing any SHMT software/library, the user should install the software components as Section A.3.3 mentions. Then, the user can install the SHMT artifact through the following steps.

```
git clone https://github.com/escal/SHMT
sh scripts/docker_setup_partition.sh
sh scripts/docker_launch_partition.sh
```

And within the docker container, do the following steps.

```
mkdir build
cd build
cmake ..
make -j4
```

This step will generate the example executable that utilizes SHMT library.

A.5 Experiment workflow

To run the example executable named `gpgtpu`, the user can leverage the existing shell scripts under `scripts/` called `AE_run.sh` to begin the process.

```
sh ../scripts/AE_run.sh
```

A.6 Evaluation and expected results

A.6.1 Evaluate Results. The evaluator can redirect the outputs to a log file and carefully examine the results.

A.6.2 Expected Results. Compared to the GPU baseline, SHMT with QAWS-TS policy can offer 1.95× speedup with MAPE equals to 1.98% on average. Please refer to Section 5 for the expected results.

A.7 Notes

To build the dependent shared library `libgtpu_utils.so` from source code that provides generic APIs interacting with Edge TPU,

which the artifact already provides under the `lib/aarch64` directory, the user can additionally download the submodules when cloning the artifact. Please follow the installation instructions under `edgetpu/` for more details.

A.8 Methodology

Submission, reviewing and badging methodology:

- <https://www.acm.org/publications/policies/artifact-review-and-badging-current>
- <http://cTuning.org/ae/submission-20201122.html>
- <http://cTuning.org/ae/reviewing-20201122.html>

REFERENCES

- [1] Martin 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, Dandelion 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. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. <https://www.tensorflow.org/>. Software available from tensorflow.org.
- [2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. MICRO '16. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*.
- [3] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. ISCA '14. General-purpose code acceleration with limited-precision analog computation. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture*. <https://doi.org/10.1109/ISCA.2014.6853213>
- [4] Sam Amiri, Mohammad Hosseinabady, Simon McIntosh-Smith, and Jose Nunez-Yanez. DATE '18. Multi-precision convolutional neural networks on heterogeneous hardware. In *2018 Design, Automation Test in Europe Conference Exhibition*.
- [5] Sam Amiri, Mohammad Hosseinabady, Andres Rodriguez, Rafael Asenjo, Angeles Navarro, and Jose Nunez-Yanez. FPL '18. Workload Partitioning Strategy for Improved Parallelism on FPGA-CPU Heterogeneous Chips. In *2018 28th International Conference on Field Programmable Logic and Applications*.
- [6] Analog Devices, Inc. 2023. Analog Devices' Processors and DSP. <https://www.analog.com/en/product-category/processors-dsp.html>.
- [7] Renzo Andri, Beatrice Bussolino, Antonio Cipolletta, Lukas Cavigelli, and Zhe Wang. MICRO '22. Going Further With Winograd Convolutions: Tap-Wise Quantization for Efficient Inference on 4x4 Tiles. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*.
- [8] Apple Inc. 2020. Apple M1. <https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/>.
- [9] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. ASPLOS '22. 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*.
- [10] Saambhavi Baskaran, Mahmut Taylan Kandemir, and Jack Sampson. MICRO '22. An architecture interface and offload model for low-overhead, near-data, distributed accelerators. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*.
- [11] G. Bradski. 2000. The OpenCV Library. *Dr. Dobbs' Journal of Software Tools* (2000).
- [12] John Burgess. 2020. RTX on—The NVIDIA Turing GPU. *IEEE Micro* 40, 2 (2020), 36–44. <https://doi.org/10.1109/MM.2020.2971677>
- [13] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc.
- [14] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. IISWC '09. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization*.
- [15] 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. OSDI '18. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*.
- [16] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. NIPS '18. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*.
- [17] Xinyu Chen, Yao Chen, Feng Cheng, Hongshi Tan, Bingsheng He, and Weng-Fai Wong. MICRO '22. ReGraph: Scaling Graph Processing on HBM-enabled FPGAs with Heterogeneous Pipelines. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*.
- [18] Yujeong Choi and Minsoo Rhu. HPCA '20. PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units. In *2020 IEEE International Symposium on High Performance Computer Architecture*.
- [19] George Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* (1989).
- [20] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. ICS '19. Accelerating Reduction and Scan Using Tensor Core Units. In *Proceedings of the ACM International Conference on Supercomputing*.
- [21] F. Fernandes dos Santos, C. Lunardi, D. Oliveira, F. Libano, and P. Rech. HPCA '19. Reliability Evaluation of Mixed-Precision Architectures. In *2019 IEEE International Symposium on High Performance Computer Architecture*.
- [22] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wen-mei Hwu. PACT '21. Accelerating Fourier and Number Theoretic Transforms using Tensor Cores and Warp Shuffles. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques*. <https://doi.org/10.1109/PACT52795.2021.00032>
- [23] Janick Edinger, Martin Breitbach, Niklas Gabrisch, Dominik Schäfer, Christian Becker, and Amr Rizk. IPDPS '21. Decentralized Low-Latency Task Scheduling for Ad-Hoc Computing. In *2021 IEEE International Parallel and Distributed Processing Symposium*.
- [24] Venmugil Elango. IPDPS '21. Pase: Parallelization Strategies for Efficient DNN Training. In *2021 IEEE International Parallel and Distributed Processing Symposium*.
- [25] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. MICRO '12. Neural Acceleration for General-Purpose Approximate Programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*.
- [26] Yuping Fan, Zhiling Lan, Paul Rich, William Allcock, and Michael E. Papka. IPDPS '22. Hybrid Workload Scheduling on HPC Systems. In *2022 IEEE International Parallel and Distributed Processing Symposium*.
- [27] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. PPOPP '21. 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*.
- [28] Boyuan Feng, Yuke Wang, Tong Geng, Ang Li, and Yufei Ding. SC '21. 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*.
- [29] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo. MobiCom '16. 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*.
- [30] Google LLC. 2020. edgetpu compiler. <https://coral.ai/docs/edgetpu/compiler>.
- [31] Google LLC. 2022. Google Pixel 6a. https://store.google.com/product/pixel_6a?hl=en-US.
- [32] 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. MICRO '22. ROG: A High Performance and Robust Distributed Training System for Robotic IoT. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*.
- [33] Cong Guo, Chen Zhang, Jingwen Leng, Zihan Liu, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. MICRO '22. ANT: Exploiting Adaptive Numerical Data Type for Low-bit Deep Neural Network Quantization. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*.
- [34] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. SC '18. 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*.
- [35] Chaoyang He, Murali Annamaram, and Salman Avestimehr. NIPS '20. Group Knowledge Transfer: Federated Learning of Large CNNs at the Edge. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*.
- [36] Nhut-Minh Ho and Weng-Fai Wong. HPEC '17. Exploiting half precision arithmetic in Nvidia GPUs. In *2017 IEEE High Performance Extreme Computing Conference*. <https://doi.org/10.1109/HPEC.2017.8091072>
- [37] Pedro Holanda and Hannes Mühleisen. DaMoN '19. Relational Queries with a Tensor Processing Unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*.
- [38] Xueyu Hou, Yongjie Guan, Tao Han, and Ning Zhang. IPDPS '22. DistrEdge: Speeding up Convolutional Neural Network Inference on Distributed Edge Devices. In *2022 IEEE International Parallel and Distributed Processing Symposium*.

- [39] Kuan-Chieh Hsu and Hung-Wei Tseng. SC '21. Accelerating Applications Using Edge Tensor Processing Units. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.
- [40] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. SIGMOD '22. TCUDDB: Accelerating Database with Tensor Processors. In *Proceedings of the 2022 International Conference on Management of Data*.
- [41] Yu-Ching Hu, Murtuza Taher Lokhandwala, Te I., and Hung-Wei Tseng. MICRO '19. Dynamic Multi-Resolution Data Storage. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.
- [42] Zhiming Hu, Ahmad Bisher Tarakji, Vishal Raheja, Caleb Phillips, Teng Wang, and Iqbal Mohamed. EMDL '19. DeepHome: Distributed Inference with Heterogeneous Devices in the Edge. In *The 3rd International Workshop on Deep Learning for Mobile Systems and Applications*.
- [43] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. CVPR '18. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.
- [44] Zhihao Jia, Matei Zaharia, and Alex Aiken. MLSys '19. Beyond Data and Model Parallelism for Deep Neural Networks.. In *Proceedings of Machine Learning and Systems*.
- [45] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. OSDI '20. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation*.
- [46] 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. ISCA '21. Ten Lessons From Three Generations Shaped Google's TPUs: Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*.
- [47] 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. ISCA '23. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *2023 ACM/IEEE 50th Annual International Symposium on Computer Architecture*.
- [48] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-specific Supercomputer for Training Deep Neural Networks. In *Communications of the ACM*.
- [49] 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 Snellman, 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. ISCA '17. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. <https://doi.org/10.1145/3079856.3080246>
- [50] Liu Ke, Udit Gupta, Mark Hempstead, Carole-Jean Wu, Hsien-Hsin S. Lee, and Xuan Zhang. HPCA '22. Hercules: Heterogeneity-Aware Inference Serving for At-Scale Personalized Recommendation. In *2022 IEEE International Symposium on High-Performance Computer Architecture*.
- [51] Hamidreza Khaleghzadeh, Ravi Reddy Manumachu, and Alexey Lastovetsky. 2020. A Hierarchical Data-Partitioning Algorithm for Performance Optimization of Data-Parallel Applications on Heterogeneous Multi-Accelerator NUMA Nodes. *IEEE Access* (2020).
- [52] Daya S Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke. ISCA '15. Rumba: An online quality management system for approximate computing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture*.
- [53] Sungil Kim and Heeyoung Kim. 2016. A new metric of absolute percentage error for intermittent demand forecasts. *International Journal of Forecasting* (2016).
- [54] Young Geun Kim and Carole-Jean Wu. MICRO '21. AutoFL: Enabling Heterogeneity-Aware Energy Efficient Federated Learning. In *54th Annual IEEE/ACM International Symposium on Microarchitecture*.
- [55] Anish Krishnakumar, Samet E. Arda, A. Alper Goksoy, Sumit K. Mandal, Umith Y. Ogras, Anderson L. Sartor, and Radu Marculescu. 2020. Runtime Task Scheduling Using Imitation Learning for Heterogeneous Many-Core Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020).
- [56] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. HPCA '21. Heterogeneous Dataflow Accelerators for Multi-DNN Workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture*.
- [57] Matthias Langer, Zhen He, Wenny Rahayu, and Yanbo Xue. 2020. Distributed Training of Deep Learning Models: A Taxonomic Perspective. In *IEEE Transactions on Parallel and Distributed Systems*.
- [58] Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. PLDI '16. Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- [59] Binrui Li, Shenggan Cheng, and James Lin. CLUSTER '21. tcFFT: A Fast Half-Precision FFT Library for NVIDIA Tensor Cores. In *2021 IEEE International Conference on Cluster Computing*. <https://doi.org/10.1109/Cluster48925.2021.00035>
- [60] Shikai Li, Sunghyun Park, and Scott Mahlke. ICS '18. Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation. In *Proceedings of the 2018 International Conference on Supercomputing*.
- [61] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. NIPS '21. MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning. In *Annual Conference on Neural Information Processing Systems*.
- [62] Ji Lin, Wei-Ming Chen, John Cohn, Chuang Gan, and Song Han. NIPS '20. MCUNet: Tiny Deep Learning on IoT Devices. In *Annual Conference on Neural Information Processing Systems*.
- [63] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. NIPS '21. On-Device Training Under 256KB Memory. In *Annual Conference on Neural Information Processing Systems*.
- [64] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. ASPLOS '22. VELTAI: Towards High-Performance Multi-Tenant Deep Learning Services via Adaptive Compilation and Scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [65] Google LLC. 2022. XLA: Domain-specific compiler for linear algebra to optimize tensorflow computations. <https://www.tensorflow.org/xla>.
- [66] Atieh Lotfi, Abbas Rahimi, Hadi Esmaeilzadeh, and Rajesh K Gupta. 2015. SqueezeCL: Squeezing OpenCL kernels for approximate computing on contemporary GPUs. In *Workshop on Approximate Computing*.
- [67] Tianjian Lu, Thibault Marin, Yue Zhuo, Yi-Fan Chen, and Chao Ma. HPEC '20. Accelerating MRI Reconstruction on TPUs. In *2020 IEEE High Performance Extreme Computing Conference*. <https://doi.org/10.1109/HPEC43674.2020.9286192>
- [68] Tianjian Lu, Thibault Marin, Yue Zhuo, Yi-Fan Chen, and Chao Ma. ISBI '21. Nonuniform Fast Fourier Transform on Tpus. In *2021 IEEE 18th International Symposium on Biomedical Imaging*.
- [69] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. ICLR '18. Mixed Precision Training. In *International Conference on Learning Representations*.
- [70] Mehryar Mohri. 2002. Semiring Frameworks and Algorithms for Shortest-Distance Problems. *Journal of Automata, Languages and Combinatorics* (2002).
- [71] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. HPCA '15. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *IEEE 21st International Symposium on High Performance Computer Architecture*. <https://doi.org/10.1109/HPCA.2015.7056066>
- [72] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. OSDI '20. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation*.
- [73] Daniel Nichols, Aniruddha Marathe, Kathleen Shoga, Todd Gamblin, and Abhinav Bhatle. IPDPS '22. Resource Utilization Aware Job Scheduling to Mitigate Performance Variability. In *2022 IEEE International Parallel and Distributed Processing Symposium*.
- [74] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue* (2008).
- [75] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. PLDI '21. 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*.
- [76] NVIDIA Corporation. 2019. NVIDIA T4 TENSOR CORE GPU. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf>.
- [77] NVIDIA Corporation. 2020. NVIDIA A100 Tensor Core GPU Architecture. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf>.
- [78] NXP Semiconductors N.V. 2023. NXP MSC8154E Quad-Core DSP with Security. <https://www.nxp.com/products/processors-and-microcontrollers/additional>

- mpu-mcus-architectures/digital-signal-processors/high-performance-quad-core-dsp-with-security:MSC8154E.
- [79] Alberto Parravicini, Arnaud Delamare, Marco Arnaboldi, and Marco D. Santambrogio. IPDPS '21. DAG-based Scheduling with Resource Sharing for Multi-task Applications in a Polyglot GPU Runtime. In *2021 IEEE International Parallel and Distributed Processing Symposium*.
 - [80] 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. NIPS '19. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*.
 - [81] Kiran Ranganath, Joshua D. Suetterlein, Joseph B. Manzano, Shuaiwen Leon Song, and Daniel Wong. SC '21. 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*.
 - [82] Stanislav G. Sedukhin and Marcin Paprzycki. 2012. Generalizing Matrix Multiplication for Efficient Computations on Modern Computers. In *Parallel Processing and Applied Mathematics*.
 - [83] Jiuchen Shi, Jiawen Wang, Kaihua Fu, Quan Chen, Deze Zeng, and Minyi Guo. IPDPS '22. QoS-awareness of Microservices with Excessive Loads via Inter-Datacenter Scheduling. In *2022 IEEE International Parallel and Distributed Processing Symposium*.
 - [84] Siddharth Singh and Abhinav Bhatle. IPDPS '22. AxoNN: An asynchronous, message-driven parallel framework for extreme-scale deep learning. In *2022 IEEE International Parallel and Distributed Processing Symposium*.
 - [85] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. HPCA, '20. AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture*.
 - [86] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. HPCA '19. HyPar: Towards Hybrid Parallelism for Deep Learning Accelerator Array. In *2019 IEEE International Symposium on High Performance Computer Architecture*.
 - [87] Pierre Stock, Angela Fan, Benjamin Graham, Edouard Grave, Rémi Gribonval, Hervé Jegou, and Armand Joulin. ICLR '21. Training with Quantization Noise for Extreme Model Compression. In *International Conference on Learning Representations*.
 - [88] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering* (2010).
 - [89] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi (Viji) Srinivasan, and Kailash Gopalakrishnan. NIPS '20. Ultra-Low Precision 4-bit Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems*.
 - [90] Tuan Ta, Khalid Al-Hawaj, Nick Cebry, Yanghui Ou, Eric Hall, Courtney Golden, and Christopher Batten. MICRO '22. big.VLITTLE: On-Demand Data-Parallel Acceleration for Mobile Systems on Chip. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*.
 - [91] Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. ISCA '18. Stitch: Fusible Heterogeneous Accelerators Enmeshed with Many-Core Architecture for Wearables. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*.
 - [92] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. CVPR '19. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.
 - [93] Chunlin Tian, Li Li, Zhan Shi, Jun Wang, and ChengZhong Xu. MICRO '22. HARMONY: Heterogeneity-Aware Hierarchical Management for Federated Learning System. In *2022 55th IEEE/ACM International Symposium on Microarchitecture*.
 - [94] Han D. Tran, Milinda Fernando, Kumar Saurabh, Baskar Ganapathysubramanian, Robert M. Kirby, and Hari Sundar. IPDPS '22. A scalable adaptive-matrix SPMV for heterogeneous architectures. In *2022 IEEE International Parallel and Distributed Processing Symposium*.
 - [95] Jack Turner, Elliot J. Crowley, and Michael F. P. O'Boyle. ASPLOS '21. Neural Architecture Search as Program Transformation Exploration. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
 - [96] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. CVPR '19. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.
 - [97] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. NIPS '18. Training Deep Neural Networks with 8-bit Floating Point Numbers. In *Advances in Neural Information Processing Systems*.
 - [98] Shuo Wang, Yun Liang, and Wei Zhang. HPCA '19. Poly: Efficient Heterogeneous System and Application Management for Interactive Applications. In *2019 IEEE International Symposium on High Performance Computer Architecture*.
 - [99] Ting Wang, Qian Zhang, and Qiang Xu. DATE '17. ApproxQA: A unified quality assurance framework for approximate computing. In *Design, Automation and Test in Europe Conference and Exhibition*.
 - [100] Joel Wolfraht, Nikhil Sreekumar, Dhruv Kumar, Yuanli Wang, and Abhishek Chandra. IPDPS '22. HACCS: Heterogeneity-Aware Clustered Client Selection for Accelerated Federated Learning. In *2022 IEEE International Parallel and Distributed Processing Symposium*.
 - [101] Yecheng Xiang and Hyoseung Kim. RTSS '19. Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference. In *2019 IEEE Real-Time Systems Symposium*.
 - [102] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. ISCA '21. HASCO: Towards Agile Hardware and Software CO-design for Tensor Computation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*.
 - [103] Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Sasa Misailovic, and Saurabh Bagchi. USENIX ATC '18. VideoChef: Efficient Approximation for Streaming Video Processing Pipelines. In *2018 USENIX Annual Technical Conference*.
 - [104] Zichao Yang, Heng Wu, Yuanjia Xu, Yuewen Wu, Hua Zhong, and Wenbo Zhang. 2023. Hydra: Deadline-aware and Efficiency-oriented Scheduling for Deep Learning Jobs on Heterogeneous GPUs. *IEEE Trans. Comput.* (2023).
 - [105] Minjia Zhang, Zehua Hu, and Mingqin Li. IPDPS '21. DUET: A Compiler-Run-time Subgraph Scheduling Approach for Tensor Programs on a Coupled CPU-GPU Architecture. In *2021 IEEE International Parallel and Distributed Processing Symposium*.
 - [106] Yunan Zhang, Po-An Tsai, and Hung-Wei Tseng. ISCA '22. SIMD2: A Generalized Matrix Instruction Set for Accelerating Tensor Computation beyond GEMM. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*.
 - [107] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. ISCA '22. AMOS: Enabling Automatic Mapping for Tensor Computations On Spatial Accelerators with Hardware Abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*.
 - [108] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. ASPLOS '20. 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*.
 - [109] Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. SEC '19. Adaptive Parallel Execution of Deep Neural Networks on Heterogeneous Edge Devices. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*.
 - [110] Wentao Zhu, Can Zhao, Wenqi Li, Holger R. Roth, Ziyue Xu, and Daguang Xu. MICCAI '20. LAMP: Large Deep Nets with Automated Model Parallelism for Image Segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*.
 - [111] Yuhao Zhu. PPOPP '22. RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.