# Simultaneous and Heterogenous Multithreading: Exploiting Simultaneous and Heterogeneous Parallelism in Accelerator-Rich Architectures

Kuan-Chieh Hsu and Hung-Wei Tseng, *University of California, Riverside, Riverside, CA, 92521, USA*

***The addition of domain-specific hardware accelerators and general-purpose processors that support vector and scalar models makes modern computers undoubtedly heterogeneous. However, existing programming models and runtime systems target using the most efficient category of processing units to delegate computation from each code region, undermining the potential parallelism that heterogeneous processing units can provide. Simultaneous and heterogenous multithreading (SHMT) is a programming and execution model that activates all possible heterogeneous processing units for computation from a code region to enable "real" heterogeneous parallelism. SHMT presents an abstraction and a runtime system to facilitate parallel execution. Despite the new type of parallelism, SHMT also needs to additionally address the heterogeneity in data precision that various processing units support to ensure the quality of the result. This article implements and evaluates SHMT on an embedded system platform with a GPU and an edge tensor processing unit.***

With the awareness of global environmental issues, the consensus of reducing carbon footprint, and the increasing cost of manufacturing chips in more advanced process technologies, computer architects must maximize the use of precious transistor resources while supporting the growing computation demand in applications. The recent trend in developing and integrating hardware accelerators, including GPUs and hardware accelerators for artificial intelligence (AI) and machine learning (ML) or digital signal processors, helps alleviate specific application demand across all forms of computers.

Recent research projects have made progress in widening the spectrums of applications and compute kernels on domain-specific accelerators with decent performance.[1,2,3,4,5] However, conventional programming models and frameworks, including domain-specific languages, still leverage the entrenched idea of delegating the execution of code regions to their most "optimal" processing resources. As a result, accelerators become idle resources due to the limitation of existing execution models. The resulting applications also miss the new parallel execution opportunity that accelerator-rich architectures create, where we can simultaneously use heterogeneous processors but leave other resources idle without contributing to the application's progress.

0272-1732 © 2024 IEEE
Digital Object Identifier 10.1109/MM.2024.3414941
Date of publication 8 July 2024; date of current version 14 August 2024.

## SIMULTANEOUS AND HETEROGENEOUS MULTITHREADING

Simultaneous and heterogenous multithreading (SHMT) is a new parallel programming and execution model that aims at enabling simultaneous use of heterogeneous computing resources. SHMT can simultaneously execute threads that are homogeneous at the algorithm level but heterogeneous from an architectural perspective. In addition to exploiting parallelism using homogeneous threads, that is, parallel execution streams on the same architecture, SHMT can exploit heterogeneous parallelism by breaking up the computation from the same algorithm block into parallel threads running on

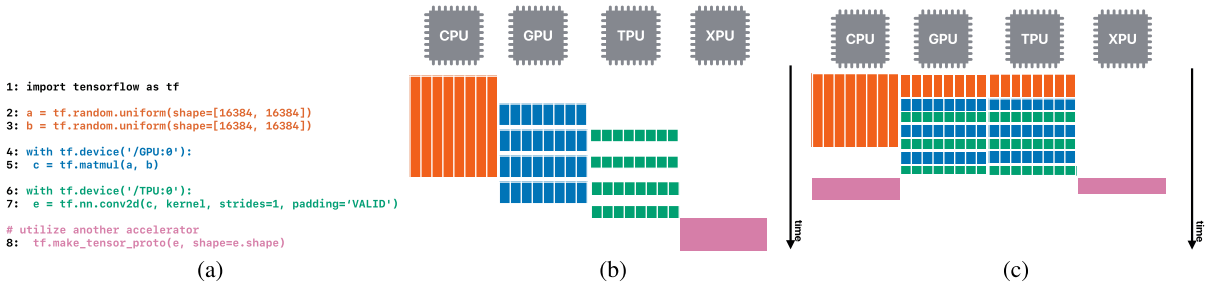


**FIGURE 1.** (a) A typical domain-specific language code example, and the execution model of (b) conventional heterogeneous computers with software pipelining and (c) SHMT. TPU: tensor processing unit; XPU: auxiliary processing unit.

multiple types of computing resources where each resource may have distinct architecture and hardware abstraction.

Figure 1 illustrates the concept of SHMT and compares SHMT with conventional heterogeneous programming models. Figure 1(a) provides an exemplary domain-specific language code where the programmer and the runtime system will delegate lines 2 and 3 to the CPU, lines 4 and 5 to GPUs, lines 6 and 7 to TPUs, and line 8 to some other accelerator. Figure 1(b) presents the runtime execution flow in state-of-the-art programming models. State-of-the-art models can exploit parallelism within the same type of processors, improve the execution by allowing the code segments to progress with partial results, and pipeline the execution of different code segments 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 idle hardware resources. SHMT, as Figure 1(c) depicts, allows line 2–8 to use all available computing resources. As a result, SHMT can significantly improve hardware utilization and lead to better end-to-end latency and energy consumption.

## Advantages of SHMT

SHMT is a competitive concept in modern heterogeneous, accelerator-rich computers for the several reasons, which we discuss in the next sections.

### *Performance Gain at Zero Hardware Cost*

The ubiquity of hardware accelerators and the broader application spectrums of hardware accelerators allow SHMT to improve the performance of applications at zero hardware cost. Figure 2 illustrates the performance advantage on an embedded platform. Figure 2 compares the performance of running 10 application's core algorithms on their state-of-the-art edge tensor processing unit (TPU) and GPU implementations on Jetson Nano. By simply relying on an edge TPU, the applications

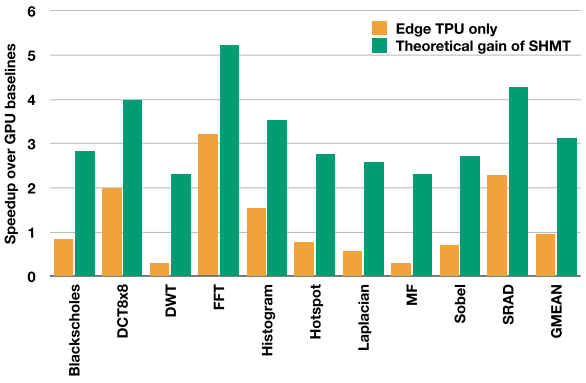


**FIGURE 2.** The potential speedup of SHMT (and Edge TPU) relative to a GPU-only implementation. FFT: fast Fourier transform; DWT: discrete wavelet transform; DCT8×8: discrete cosine transform 8×8; SRAD: speckle reducing anisotropic diffusion; MF: mean filter.

are 5% slower than those of GPUs. The average theoretical speedup from conventional approaches that delegate kernels to the best-performing accelerator is $1.37\times$. Using the same performance number, we derived the theoretical performance gain of SHMT: $3.14\times$.

### *Environmentally Friendly Computer Architecture*

Data centers currently account for 3% of global electricity and are projected to reach 4% in 2030. The nonstoppable operations of data centers can consume a significant amount of energy in idle hardware resources if we cannot make progress. As SHMT reclaims idle hardware accelerators and allows applications to make progress using them, it has substantial implications in energy savings (e.g., 51% in this article) that help to reduce the carbon footprint in electricity. The continuous upgrade of hardware, including processors and accelerators, generates a significant carbon footprint when producing new components. As SHMT uses

multiple heterogeneous accelerators simultaneously, applications can continuously use an existing accelerator, slowing down the demand for hardware upgrades or the upscale of data centers.

### *Flexibility and Supply-Chain-Issue Free*

As SHMT aims at executing parallel threads on various hardware components, it makes the program inherently flexible when scheduling computing resources. With broader choices of hardware resources for performing critical tasks, the system can reduce the queuing delay after user requests. On the other hand, SHMT also makes computer systems less fragile to supply-chain issues. SHMT reduces applications' and services' reliance on specific hardware components by allowing threads to easily use other accelerators if one becomes a scarce resource, enabling more flexibility in data center architectures and service deployments.

## Challenges of SHMT

Despite its benefits, several roadblocks create challenges for implementing and supporting the SHMT model.

### *Programming*

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.

### *Performance and Scheduling*

As the microarchitecture and execution model of each hardware accelerator differs, the relative performance ratio and data exchange overhead among hardware accelerators change as data sizes or system dynamics change. Therefore, even if the programmer can partition computation to simultaneous threads working on different data partitions, the resulting program can be suboptimal for the underlying hardware or cannot guarantee speedup.

### *Quality*

Unlike homogeneous hardware components that accept data in the exact representation and deliver the same accuracy, application-specific hardware accelerators aim to provide just enough result quality for the target workloads. Most of the hardware accelerators, especially those targeting AI/ML workloads, do not support the precision modes for exact, general-purpose computing. As a result, SHMT must deal with heterogeneous hardware components, accept data, and deliver results in different formats and accuracies. Carelessly using heterogeneous hardware components simultaneously can lead to unwanted execution results.

## SHMT System Architecture

As an initial work exploring the SHMT model, we propose a framework that contains three main components to support and address the challenges of SHMT. Figure 3 presents an overview of the proposed framework.

### *Virtual Operations and HLOPs*

SHMT promotes a set of virtual operations (VOPs) and high-level operations (HLOPs) in addition to existing abstractions between hardware and software. A VOP is a mathematical or algorithmic operation that accepts tensors as inputs and produces a tensor as a result. The proposed framework abstracts the underlying hardware resources as a powerful and resourceful hardware accelerator that offers a set of VOPs for a CPU program to "offload" computation to this virtual device (e.g., an SHMT virtual device driver in the operating system). The compiler or the programmer can use VOPs to describe the desired computation for SHMT. VOPs provide a machine-independent abstraction interface for the software and make no assumptions about the input/output data size. As they adopt domain-specific languages (e.g., TensorFlow or PyTorch) using standard libraries and accelerated libraries (e.g., cuBLAS and cuDNN) in modern programming languages, we expect the front-end authoring languages of user programs to remain the same. Most of the changes should occur only at the library level.

An HLOP defines a subset of computations from a VOP instance, and the SHMT framework contains a mapping between a VOP and its HLOPs. An HLOP also represents a task from a VOP instance that a computing resource in SHMT will later execute. Therefore, an HLOP will have predefined data sizes/granularities and types. An SHMT framework may contain multiple implementations of the same HLOP. Each implementation of an HLOP is a machine-dependent code that maps to a set of hardware operations and functions on the target hardware resource. For example, as an edge TPU implements convolution 2-D in hardware, the edge TPU's HLOP implementation 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 2-D implementation on a GPU will internally become a series of vector operations within the HLOP implementation. For AI/ML accelerators or neural processing units (NPUs), the implementation

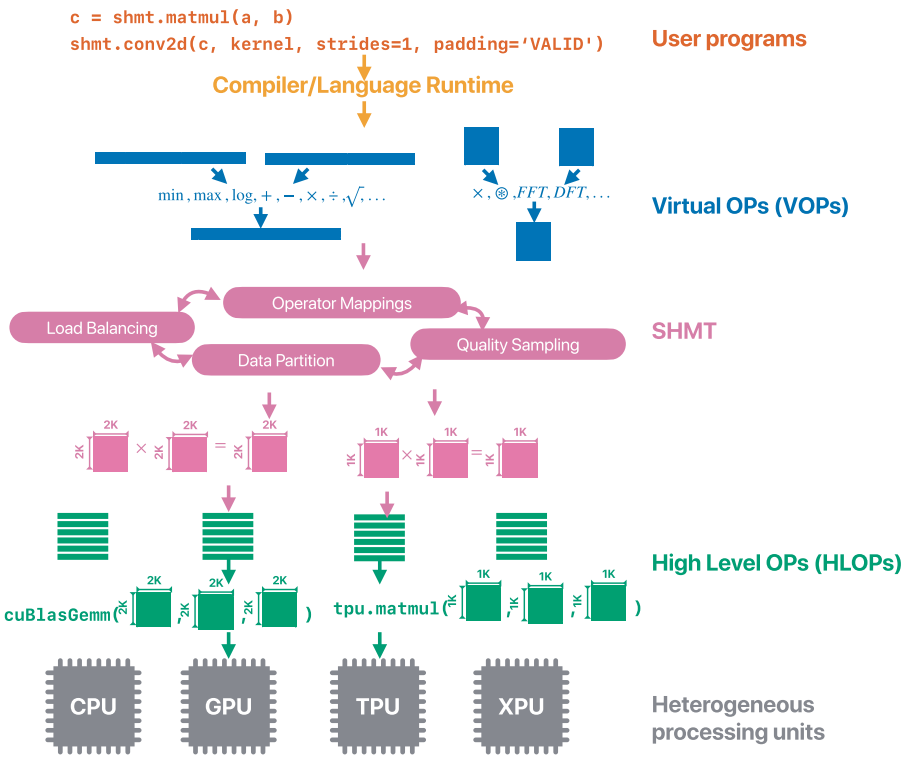


**FIGURE 3.** SHMT overview.

makes an inference through a pretrained model that approximates the result of convolution 2-D.

The abstractions of VOPs and HLOPs help SHMT address both the programming and scheduling challenges. As user programs and authoring languages interact with the machine-independent VOPs, VOPs hide the hardware details and facilitate code development. As HLOPs carry the actual execution of a VOP instance and the various implementations that an HLOP has, this abstraction also enables flexibility in scheduling.

### *Runtime Systems*

SHMT's runtime system resides in the virtual device driver. The runtime system 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 initialization of the SHMT system, each hardware resource's driver is responsible for providing SHMT with its list of available HLOPs and their implementations.

For each VOP that SHMT receives, the runtime system figures out the available hardware resources to perform the VOP, gathers 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 that monitors 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 the SHMT runtime system can later dequeue and use for data aggregation and synchronization purposes.

In modern heterogeneous systems, hardware accelerators are typically separated intellectual property cores or chips that communicate with the CPU cores through the system interconnect. Like the idea of processor caches, most of the 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 that take the starting address of the source data structure and use the element size and 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 of the hardware accelerators optimize their computation models and architectures for targeted application domains, thus supporting limited data precisions. Suppose that the scheduling policy determines the appropriate use of a target hardware resource despite the potential loss of accuracy. 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.

The runtime system design enables optimizations that address performance and scheduling concerns. At the HLOP level, each HLOP individual implementation provides performance-optimized code on the hardware. At the scheduler level, the runtime system dynamically assigns tasks to ensure the best workload distribution. The runtime system can also implement load-balancing mechanisms, like work-stealing policies, to maximize hardware efficiency.

### *Quality-Aware Scheduling*

SHMT also addresses the quality issue at the scheduling policy level with a low-overhead proposal that considers results and performance. The proposed quality-aware work-stealing (QAWS) scheduling policy has low execution overhead but helps to maintain quality and balance the workload. For each input data partition, SHMT samples the data to determine criticality and assigns computation to a device accordingly. We leverage the experience from previous works that consider critical regions as data partitions with the widest value distributions.[6] For data partitions where SHMT considers as critical to the quality of the execution result, the runtime system will never assign the related HLOPs to a hardware resource that does not support the data's desired precision.

Based on our experiments, we found that the scheduling policy can be easy but still effective. The current version of SHMT need only consider the following two criteria:

1) *Predefined device-dependent limits*: SHMT determines task scheduling on a device considering device-dependent hardware limits. Each computing device has a set of acceptable hardware limits, including the supporting data precision, data types, and accuracies. SHMT assigns only those data inputs lower than the criticality limits to that computing resource. Regarding work stealing for loading balancing, SHMT only allows a device to steal HLOPs from another device with the same or a lower hardware limit to ensure quality.
2) *Application-dependent top-K% criticality*: SHMT ranks criticality within a window of data partitions and schedules top-*K*% partitions to the most accurate device, second-*L*% to the second-most precise 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, an indication of the percentage of data inputs that are generally critical to the results of 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.

*WE IMPLEMENTED THE VIRTUAL SHMT HARDWARE DEVICE AS A DYNAMICALLY LOADABLE KERNEL MODULE.*

## RESULTS

### The System Prototype

We evaluated the proposed SHMT framework and ideas on a custom-built, exemplary SHMT prototype system. The system uses Nvidia's Jetson Nano, and we added Google's Edge TPU as an AI/ML accelerator. The Jetson Nano module contains a quad-core ARM A57 processor and 128 Maxwell GPU cores. 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 of the use cases for the following reasons. First, the processing power and the available types of processors and accelerators for this system platform resemble the hardware components of modern smartphones or mobile devices, allowing this platform to assess the real performance of using SHMT in these scenarios. Second, the ratio of computing power between

the Maxwell GPUs and the Edge TPUs (472 giga floating-point operations versus 4 tera operations) resembles those on data center servers [67 teraflops (Tflops) FP32 of A100, and 275 Tflops of TPUv4],[7,8] allowing this platform to assess the relative performance of SHMT on cloud servers.

## Implementations of HLOPs

As an initial proof of concept, the current SHMT supports two types of accelerators, GPUs and Edge TPUs, and implements HLOPs optimized for their architectures. For the GPU HLOPs, the implementation invokes the state-of-the-art library that Nvidia and the open source community provide.

For Edge TPUs, the device can either serve as a matrix function accelerator or a general-purpose neural accelerator (i.e., an NPU)[5] for algorithms that do not map well to tensor algebra. Each HLOP of the Edge TPU using the NPU approach is a pretrained model that approximates the result of the HLOP. Based on the microarchitecture of Edge TPUs and our experiences, we suggest that these models should 1) use multilayer perceptrons with convolution and dense operators, and sigmoid or a rectified linear unit as activation functions, and 2) be the first found and the simplest topology whenever the learning curve of a full-precision model training significantly improves throughout topology searching.

We use the following steps to construct an NPU model on an 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 posttraining quantization for the trained model into an Edge-TPU-compatible model using TensorFlow Lite and *edgetpu_compiler*.[9]
4) Test the Edge-TPU-compatible model with a 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 a quantization-aware training mode to retrain the model with weights in 8-bit precisions.

## Performance Gain

Figure 4 shows the performance of SHMT's real-system implementation on Jetson Nano with an Edge TPU. Comparing the end-to-end latency of SHMT with optimized baseline GPU implementations, SHMT with the best-performing quality-assurance policy achieves a 1.95× speedup, while work-stealing without quality assurance can achieve 2.07× speedup. Figure 4 also includes the performance of optimized GPU implementations with software pipelining as another reference design. Software pipeline can achieve only a 1.25× speedup. For compute-intensive workloads, software pipelining cannot compete with SHMT. Software pipelining is effective for Blackscholes and mean filter 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 algorithms exist.

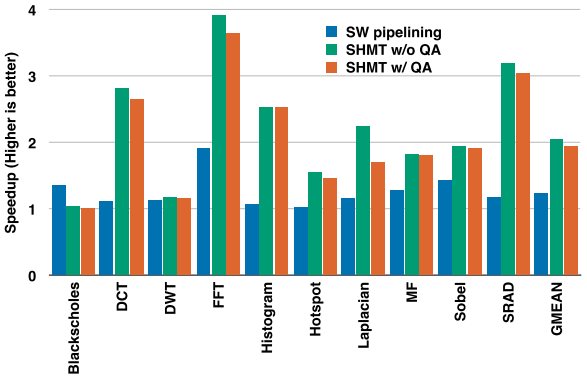


**FIGURE 4.** The application speedup relative to the baseline GPU implementations. SW: software; w/: with; w/o: without; QA: quality assurance; GMEAN: geometric mean.

## Performance of Quality-Assurance Mechanisms

The design of SHMT's quality-assurance mechanism should balance the quality of results and sampling overhead. Our experiments found that SHMT can deliver reasonable results with relatively low overhead for the examined datasets and applications. Figure 5(a) and (b) shows the speedup and mean absolute percentage errors (MAPEs) under various sampling rates (the portion as samples from the raw datasets). A sampling rate of $2^{-14}$ means that we select 256 samples from a 2048 × 2048-sized input. We changed the sampling rate from $2^{-21}$ to $2^{-14}$. 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 SHMT policies without sacrificing performance gain considerably.

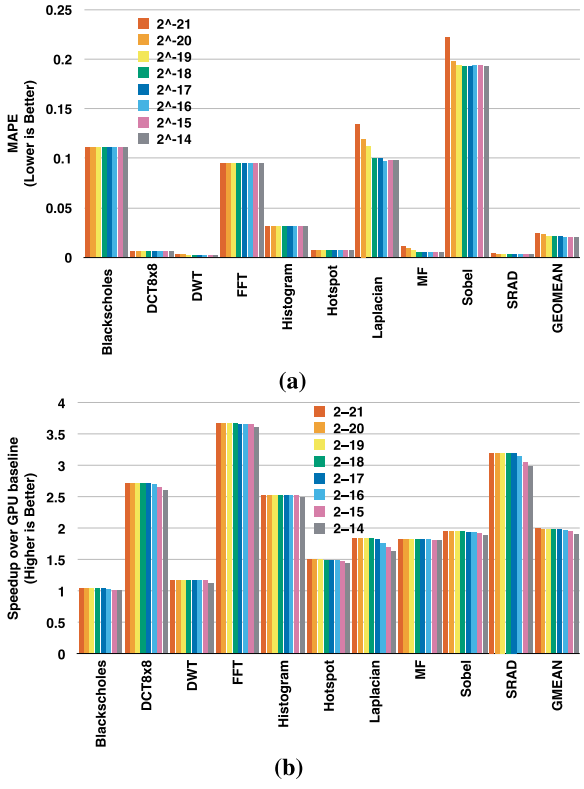


**FIGURE 5.** (a) Quality versus sampling rates and (b) speedup versus sampling rates.

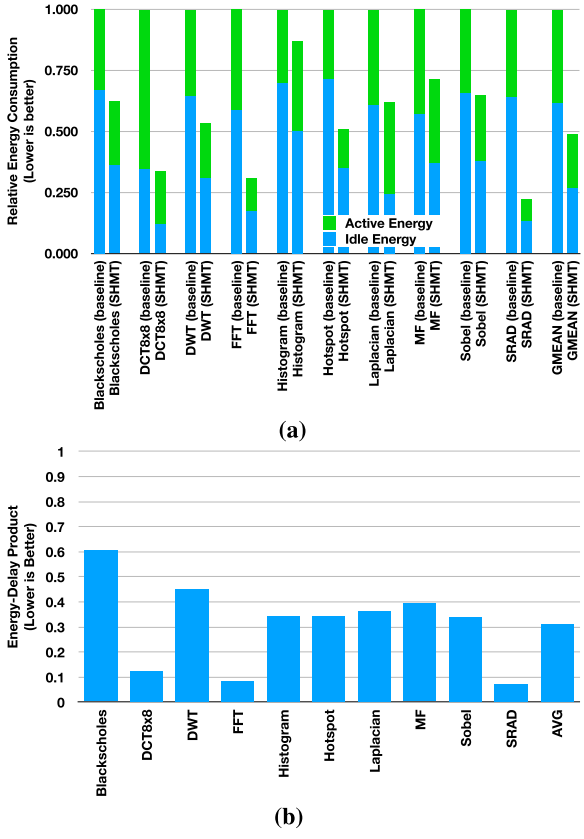


**FIGURE 6.** (a) Energy consumption and (b) energy-delay product. AVG: average.

## Energy Consumption

By reducing the total execution time and offloading computation to a lower-power-consuming Edge TPU that potentially idles, SHMT has a strong promise for energy savings. Figure 6(a) reports the breakdown of energy consumption of both the GPU baseline and SHMT. Figure 6(b) shows the relative energy-delay products (EDPs) of SHMT compared to the GPU baseline. SHMT reduces energy consumption and EDPs by 51% and 69% on average, respectively.

## Scalability

Figure 7 shows the speedups of SHMT when problem sizes of benchmarks vary. Within the tested problem-size interval, from 4 K to 64 M, the speedup increases as the problem size increases. We did not go beyond 64 M as the working set size of GPU kernels in some applications will surpass the physical memory limitation and crash, not the limitation of SHMT. SHMT is more effective for larger problem sizes as a larger problem size provides more parallelism among HLOPs for various devices. The main reason behind SHMT's scalability is that SHMT does not lead to significant memory and communication overhead when algorithms leverage embarrassingly matrix-tile, data-level parallelism. Therefore, the adoption of SHMT helps the system to enjoy more parallel processing resources to

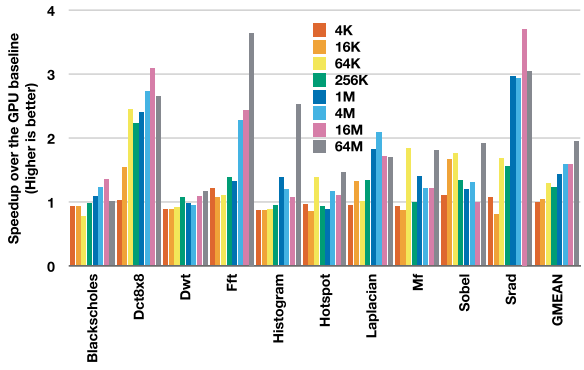


**FIGURE 7.** Speedup versus problem sizes.

tackle larger problem sizes without significantly further burdening the system.

## DISCUSSIONS

Through our experience with developing and evaluating SHMT, we found several aspects that may need more investigation and may potentially inspire more research topics.

### Hardware/Software Interface in Accelerator-Rich Architectures

In addition to its flexibility and programmability, SHMT's two-layer VOPs and HLOPs abstractions also bring back the "free ride" of software programmers on computer architecture advancement. SHMT programs can naturally take advantage of a new hardware component if the HLOP implementation on the new hardware is present. In fact, recent research projects have revealed the potential of automating the process of matching tensor operators to the functions of hardware accelerators.[10] However, if SHMT's experience suggests a future of compilation and low-level programming on an abstraction like VOPs, which kinds of VOPs would offer the best efficiency, flexibility, and programmability for future applications?

### Quality-Assurance and Scheduling Policies

The current progress of SHMT is still at the concept-proving stage. However, putting SHMT into real practice would require more investigation of the runtime system's policies, especially the aspect of ensuring the execution result. The current implementation follows the best-effort philosophy but cannot guarantee the correctness. The current scheduling policy is performance oriented, but future work can cover carbon footprint considerations or clean energy costs.

### SHMT in Other Architectures

The current SHMT proposal assumes discrete accelerators and a runtime system. However, as modern processors integrate accelerators onto the same chip, the runtime system should also become a part of the system on chip to ensure the lowest overhead of the model. In addition, emerging processors' support in matrix instruction set architectures also opens up the potential of SHMT at the level of processor instructions.

## CONCLUSION

SHMT presents a new model that is fundamentally different from all existing parallel computing models as SHMT creates heterogeneous threads that perform homogeneous operations at the algorithm level, but the implementation of each thread is native to the target heterogeneous hardware. To the best of our knowledge, this is the only parallel computing model that exploits heterogeneous hardware in this way. SHMT can yield orders-of-magnitude improvements in performance and energy without adding costs to existing components. SHMT also has implications in more environmentally friendly execution models and the potential to inspire more research projects.

## ACKNOWLEDGMENTS

This work was sponsored by two U.S. National Science Foundation awards: CNS-2007124 and CNS-2231877. This work was also supported by Intel Corporation and new faculty start-up funds from the University of California, Riverside.

## REFERENCES

1. P. Holanda and H. Mühleisen, "Relational queries with a tensor processing unit," in *Proc. 15th Int. Workshop Data Manage. New Hardware, (DaMoN)*, 2019, pp. 1–3.
2. A. Dakkak, C. Li, J. Xiong, I. Gelado, and W-m Hwu, "Accelerating reduction and scan using tensor core units," in *Proc. ACM Int. Conf. Supercomput. (ICS)*, 2019, pp. 46–57.
3. Y.-C. Hu, Y. Li, and H.-W. Tseng, "TCUDB: Accelerating database with tensor processors," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2022, pp. 1360–1374, doi: 10.1145/3514221.3517869.
4. K.-C. Hsu and H.-W. Tseng, "Accelerating applications using edge tensor processing units," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2021, pp. 1–14.
5. H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit.*, Vancouver, BC, Canada, 2012, pp. 449–460, doi: 10.1109/MICRO.2012.48.
6. M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsiveness: Using canary inputs to dynamically steer approximation," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implementation (PLDI)*, 2016, pp. 161–176.
7. "NVIDIA A100 tensor core GPU architecture." NVIDIA. Accessed: 2020. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

8. N. P. Jouppi et al., "TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proc. ACM/IEEE 50th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2023, 1–14, doi: 10.1145/3579371.3589350.
9. Google LLC. "Edge TPU compiler." coral.ai. Accessed: 2020. [Online]. Available: https://coral.ai/docs/edgetpu/compiler
10. P. A. Martínez, J. Woodruff, J. Armengol-Estapé, G. Bernabé, J. M. García, and M. F. P. O'Boyle, "Matching linear algebra and tensor code to specialized hardware accelerators," in *Proc. 32nd ACM SIGPLAN Int. Conf. Compiler Construction (CC)*, 2023, pp. 85–97, doi: 10.1145/3578360.3580262.

**KUAN-CHIEH HSU** received his Ph.D. degree in computer science and engineering from the University of California, Riverside, Riverside, CA, 92521, USA. His research interests include heterogeneous computing and high-performance computing, with a primary focus on enhancing the versatility of current artificial intelligence accelerators for broader application domains cost-effectively. He is a Student Member of IEEE. Contact him at khsu037@ucr.edu.

**HUNG-WEI TSENG** is an associate professor in the Department of Electrical and Computer Engineering, University of California, Riverside, Riverside, CA, 92521, USA. His research interests include designing architecture, programming language frameworks, and system infrastructures that allow applications and programmers to use modern heterogeneous hardware components more efficiently, with a focus on using artificial intelligence (AI)/machine language (ML) accelerators to improve the performance of non-AI/ML workloads. Tseng received his Ph.D. degree from the University of California, San Diego. Contact him at htseng@ucr.edu.