Adapting Simultaneous and Heterogenous Multithreading to a generalized implementation

Leopold Kernegger
Student
TGM
Vienna, Austria
lkernegger@proton.me

***Abstract*—This paper evaluates the performance and practicality of Simultaneous and Heterogeneous Multithreading (SHMT), a framework designed to enhance computational efficiency by dynamically distributing tasks across multiple heterogeneous co-processors. Benchmarks conducted on Coral Edge TPU and Jetson Nano reveal that SHMT introduces significant overhead in specific use cases like the Blackscholes algorithm, limiting its practical advantages.**

***Index Terms*—SHMT, queues, simultaneous computing, QAWS**

I. Introduction

In 2023, Kuanchieh Hsu and Hung-Wei Tseng released a paper called ***Simultaneous and Heterogeneous Multithreading***, introducing a novel approach to simultaneous computing. Instead of using a dedicated accelerator for a task, they proposed splitting a task into subtasks, which could then be processed on the individual co-processors of the processor.

The paper presents a series of promising results, showing an average performance uplift of 95% and an average reduction in energy consumption of 51%. To make these gains practical for real-world applications, we rewrote their experimental implementation into a library—both to verify their results and to provide a way for others to use this technology. [1]

II. What is SHMT

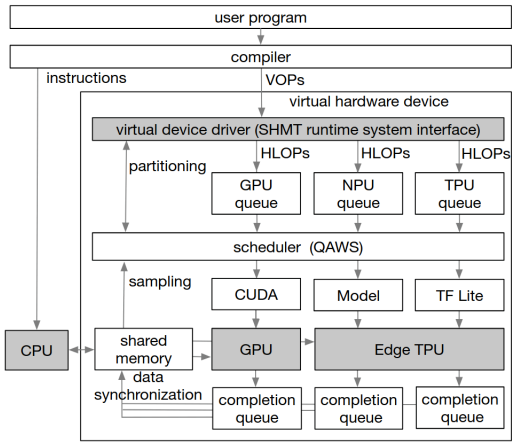

Fig. 1: Explanatory Diagram from P.4 of [1]

Simultaneous and Heterogeneous Multithreading, short SHMT, works through two base mechanisms, Quality Aware work stealing and Process queues.

A. Quality-Aware Work-Stealing (QAWS) Scheduling Policy

Quality-Aware Work-Stealing (QAWS) is an advanced scheduling policy designed to optimize workload distribution in heterogeneous computing environments while ensuring computational accuracy. Unlike traditional work-stealing, which dynamically redistributes computational tasks to balance workload and maximize resource utilization, QAWS introduces an additional quality-awareness mechanism to account for varying levels of precision among processing units. This is particularly relevant in environments where domain-specific accelerators, such as Tensor Processing Units (TPUs) and Edge TPUs, exhibit different numerical precision characteristics compared to general-purpose CPUs or GPUs. [2]

QAWS operates by employing sampling-based methods to determine the criticality of data partitions, allowing it to classify tasks based on precision requirements. Two key approaches are utilized: (1) device-dependent limits, which impose predefined thresholds on task assignments based on hardware precision capabilities, and (2) application-dependent criticality ranking, where data partitions are dynamically ranked according to their numerical importance. By integrating these strategies, QAWS ensures that high-precision computations are allocated to hardware capable of maintaining result fidelity, while less critical computations are offloaded to energy-efficient accelerators. Empirical evaluations demonstrate that QAWS achieves a 1.95× speedup over GPU-exclusive execution while reducing energy consumption by 51.0%, highlighting its effectiveness in balancing performance and computational accuracy in heterogeneous systems. [1]

B. Process Queue Management in SHMT

The SHMT framework employs a structured queuing mechanism to facilitate task execution across heterogeneous hardware components. As seen in Fig. 1, each computing resource within the system, including GPUs, NPUs, and TPUs, is associated with an incoming queue (e.g., GPU queue, NPU queue, TPU queue) that receives high-level operations (HLOPs) from the SHMT runtime scheduler. These HLOPs are derived from virtual operations (VOPs) generated by the compiler and are dynamically partitioned by the SHMT runtime system interface.

The scheduler assigns HLOPs to appropriate processing units based on hardware availability, workload distribution, and quality-awareness constraints. Task execution occurs within specialized execution environments, such as CUDA for GPUs, machine learning models for NPUs, and TensorFlow Lite for TPUs. Once execution is complete, results are transferred to the corresponding completion queues, where they undergo aggregation and synchronization before final integration into the computational pipeline.

Additionally, shared memory facilitates data synchronization between the CPU and accelerators, ensuring efficient communication. The structured queuing mechanism enables real-time scheduling adjustments, balancing throughput, energy efficiency, and computational accuracy across heterogeneous hardware resources. By dynamically adjusting task allocation in response to system workload changes, SHMT mitigates performance bottlenecks and enhances overall efficiency. This queuing model has been shown to optimize hardware utilization while maintaining result fidelity, reinforcing SHMT's applicability in high-performance heterogeneous computing environments. [1], [3]

III. Methodology

a) *Hardware:* To evaluate the performance of the SHMT framework, we conducted all tests on the Coral Edge Dual TPU on an M.2 interface, though only one TPU was active as SHMT does not yet support dual-tpu execution. Additionally, we used an NVIDIA Jetson Nano 4GB Developer Kit without performance throttling (maximum clock speed configuration).

b) *Code Porting and Execution Environment:* To ensure a fair comparison, we ported the original SHMT implementation to execute exclusively on the Jetson Nano's GPU and CPU directly. By calling identical algorithms as the original SHMT implementation, we eliminated variations arising from implementation differences.

c) *Execution:* All executions of SHMT—including scheduling, queuing, and hardware dispatch—used the original SHMT code called via a shared object (.so) library. The OS environment for all tests was NVIDIA's official Jetson Nano image, based on Ubuntu 18.04. All tests were run on identical pre-generated random data to minimize variance.

Blackscholes was selected because it was demonstrative and already implemented in the original SHMT paper, facilitating direct result verification.

IV. Implementation

This section provides a detailed overview of the practical implementation employed to benchmark the SHMT framework, including an illustrative pseudocode representation of key operations. The benchmark implementation is divided into two distinct parts: the library interface, responsible for data initialization and kernel invocation, and the kernel implementation, which manages internal execution and resource handling.

The library interface initializes test data by generating random floating-point values to mimic realistic workloads. After preparation, this data is provided to the computational kernel—in this case, the Blackscholes algorithm—alongside parameters specifying the problem and block sizes.

```
FUNCTION main():
   INITIALIZE random seed
   SET problem_size TO 256
   SET block_size TO 128
   CALCULATE total_elements AS 3 ×
problem_size²
   ALLOCATE input_array[total_elements]

   FOR EACH index FROM 0 TO total_elements -
1:
      SET input_array[index] TO
random_float(0, 100)

   EXECUTE output_array ←
blackscholes_2d(input_array, block_size,
total_elements)

   OUTPUT first 10 elements FROM
output_array

   DEALLOCATE input_array
   DEALLOCATE output_array
END FUNCTION
```

The kernel implementation receives the input data and calculates appropriate internal parameters, including verifying that the provided data size matches expected dimensions for processing. Subsequently, memory buffers for input and output data are allocated. Utilizing SHMT's runtime environment, a virtual operation scheduling system (VOPS) dispatches the computational workload to the selected hardware components based on the specified internal mode. Performance metrics, including execution overhead, are captured during this process.

```
FUNCTION blackscholes_2d(input,
block_size, total_elements):
   SET execution_mode TO "shmt"
   DETERMINE internal_mode FROM
execution_mode

   CALCULATE channel_elements AS
total_elements ÷ 3
   COMPUTE problem_size AS
SQRT(channel_elements)
   VERIFY problem_size² EQUALS
channel_elements ELSE RAISE ERROR

   DEFINE params WITH ("blackscholes_2d",
problem_size, block_size,
performance_mode)

   ALLOCATE input_array OF SIZE 3 ×
problem_size² × sizeof(float)
   ALLOCATE output_array OF SIZE 3 ×
problem_size² × sizeof(float)
   COPY input INTO input_array

   INITIALIZE performance_metrics
(time_breakdown)
   INITIALIZE kernel_dispatcher (vops)

   EXECUTE device_sequence ←
```

```
kernel_dispatcher.run_kernel(internal_mode,
params, input_array, output_array,
performance_metrics)

   ALLOCATE result_array OF SIZE identical
TO output_array
   COPY output_array INTO result_array

   DEALLOCATE input_array AND output_array
   RELEASE performance_metrics

   RETURN result_array
END FUNCTION
```

The structured approach presented here ensures clarity in the benchmarking procedure, supporting reproducibility and precise identification of overheads and potential optimizations within the SHMT runtime environment.

V. Results

TABLE I: Test results for Blackscholes algorithm.

Data Size	SHMT Exec Time (µs)	CPU Exec Time (µs)	GPU Exec Time (µs)	SHMT / CPU Ratio
256	169783	1241	96076	0.0073
512	435026	1664	101451	0.0038
1024	1544394	2343	98103	0.0015
2048	5733472	3237	207575	0.00056
4096	22662321	5095	101622	0.00022
8192	96189518	8077	102246	0.000084
10240	302399601	10333	192788	0.000034
12288	636004832	12035	195629	0.000019

The results obtained are notably below expectations. The most plausible explanation for this outcome is outlined as follows:

Performance differences are largely due to substantial overhead from measurement instrumentation and QAWS scheduling in the SHMT library. CUDA scalability was limited due to the Jetson Nano's GPU memory bandwidth constraints and our practical CUDA implementation skills. A complete rewrite of substantial SHMT code sections would be required to meaningfully reduce this overhead and enhance performance.

VI. How is this impactful to robotics?

Robots represent a rapidly advancing technology, increasingly transitioning from specialized industrial automation to versatile general-purpose machines. As robots become more generalized, their computational demands escalate dramatically. For instance, traditional industrial robots typically require minimal sensor input, relying primarily on predetermined, reliable motion paths. Conversely, emerging robotic applications—such as domestic robots like Tesla's Optimus—must process data from diverse sensors, including auditory, tactile, visual, and LiDAR inputs, alongside multiple gyroscopes and accelerometers. This diversity in sensor data inherently requires specialized computational units, such as NPUs or TPUs for AI-driven vision processing. Given the critical requirement for real-time operation, efficient integration and simultaneous utilization of these heterogeneous processors become essential. Frameworks like SHMT, or future refinements thereof, directly address this challenge, enabling complex computations to be dynamically distributed across various specialized processors to achieve the necessary real-time performance and reliability in advanced robotics systems.

References

[1] K.-C. Hsu and H.-W. Tseng, "Simultaneous and Heterogenous Multithreading," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, in MICRO '23. Toronto, ON, Canada: Association for Computing Machinery, 2023, pp. 137–152. doi: 10.1145/3613424.3614285.

[2] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999, doi: 10.1145/324133.324234.

[3] V. Maksimovic, M. Simic, M. Stojkov, and M. Zaric, "Task queue implementation for edge computing platform." [Online]. Available: https://arxiv.org/abs/2410.19344