Benchmarking Thread Block Cluster

Tim Lühnen
Massively Parallel Systems Group
Hamburg University of Technology
Hamburg, Germany
tim.luehnen@tuhh.de

Tobias Marschner
Massively Parallel Systems Group
Hamburg University of Technology
Hamburg, Germany
tobias.marschner@tuhh.de

Sohan Lal
Massively Parallel Systems Group
Hamburg University of Technology
Hamburg, Germany
sohan.lal@tuhh.de

***Abstract*—Graphics processing units (GPUs) have become essential accelerators in the fields of artificial intelligence (AI), high performance computing (HPC), and data analytics, offering substantial performance improvements over traditional computing resources. In 2022, NVIDIA's release of the Hopper architecture marked a significant advancement in GPU design by adding a new hierarchical level to their CUDA programming model: the thread block cluster (TBC). This feature enables the grouping of thread blocks, facilitating direct communication and synchronization between them. To support this, a dedicated SM-to-SM network was integrated, connecting streaming multiprocessors (SMs) to facilitate efficient inter-block communication. This paper delves into the performance characteristics of this new feature, specifically examining the latencies developers can anticipate when utilizing the direct communication channel provided by TBCs. We present an analysis of the SM-to-SM network behavior, which is crucial for developing accurate analytical and cycle-accurate simulation models. Our study includes a comprehensive evaluation of the impact of TBCs on application performance, highlighting scenarios where this feature can lead to significant improvements. For instance, applications where a data-producing thread block writes data directly into the shared memory of the consuming thread block can be up to 2.3× faster than using global memory for data transfer. Additionally, applications constrained by shared memory can achieve up to a 2.1× speedup by employing TBCs. Our findings also reveal that utilizing large cluster dimensions can result in an execution time overhead exceeding 20%. By exploring the intricacies of the Hopper architecture and its new TBC feature, this paper equips developers with the knowledge needed to harness the full potential of modern GPUs and assists researchers in developing accurate analytical and cycle-accurate simulation models.**

***Index Terms*—CUDA, Benchmarking, Hopper GPU, Thread block cluster**

I. Introduction

GPUs have emerged as a highly popular platform for accelerating applications characterized by a high degree of data-level parallelism. As a result, HPC applications are increasingly benefiting from massively parallel GPUs, as evidenced by the fact that 400 supercomputers in the TOP500 list use GPU-based accelerators from NVIDIA and AMD [1]. While there is an ongoing race in the world to acquire the latest GPUs, fueled by the competition to win the AI race, utilizing them to their full potential is a much harder challenge as applications need to be optimized to fully tap into the performance benefits offered by the ever-evolving GPU architectures.

Since the release of the first general-purpose GPU architecture (Tesla in 2006 [2]) to the latest (Hopper in 2022 [3]), NVIDIA has made numerous changes and extensions to improve the performance, energy efficiency and overall capabilities of their compute devices. While some architectural improvements, such as the increase in the number of cores, higher memory bandwidth, or the introduction of caches (Fermi architecture [4]) are transparent to applications as performance gains are obtained without any code changes, other architectural enhancements, such as tensor cores, the tensor memory accelerator, thread block clustering or grid synchronization require significant changes to codes to realize their potential, which is a tedious and challenging task.

The Hopper architecture introduced a new level of hierarchy in the CUDA programming model called TBC, which also requires significant code changes. With the use of distributed shared memory (DSMEM), thread blocks within the same TBC can access the shared memory of other blocks, enabling more efficient data sharing and communication across blocks.

Unfortunately, the performance characteristics and accompanying features of TBCs remain largely unknwon. To assess whether TBCs offer advantages for a given application, it is crucial to thoroughly understand their performance behavior. Additionally, a dedicated SM-to-SM network was introduced in GPUs to facilitate communication within clusters, but current GPU simulators [5], [6] lack this network. Therefore, to create accurate simulation as well as analytical models, it is crucial to know the instruction latencies and the SM-to-SM network behavior in detail.

This paper presents a comprehensive microbenchmarking study to determine the latencies of instructions introduced for managing TBCs. We measure the overhead of using TBCs, investigate the most effective communication strategy for inter-block communication, and aim to gain deeper insights into the SM-to-SM network. To show practical use cases, we modified four applications to leverage TBCs. Our main contributions are summarized as follows:

- We conduct a detailed microbenchmarking analysis of TBC, providing key insights into its performance characteristics and practical implications. We specifically benchmark instruction latencies, offering valuable information for developers and for GPU architects to develop accurate analytical and cycle-accurate simulation models.
- We benchmark cluster launch overhead; study SM-to-SM network and various data transfers patterns, showing that most efficient pattern can provide a speedup of 2.3×.

- We show applications constrained by shared memory can achieve up to a 2.1× speedup by employing TBC compared to global memory.

The paper is organized as follows: Section 2 describes the background, Section 3 details the microbenchmarks and case studies used in this paper, Section 4 explains the experimental setup, Section 5 discusses the results, Section 6 reviews related work, and Section 7 presents the conclusions.

II. Background

Unlike multi-core CPUs, which consist of a few powerful cores, GPUs are built with thousands of simpler cores that execute tasks in parallel. In the Compute Unified Device Architecture (CUDA) programming model, threads are organized into thread blocks, and when a kernel is launched, each thread block is assigned to a SM. Within a thread block, data can be shared through shared memory, and synchronization is achieved via barriers. Figure 1 shows that the SMs of a GPU are grouped into texture processing clusters (TPCs) which are further organized into GPU processing clusters (GPCs). Earlier GPU generations had only a handful of SMs, but with the latest Hopper architecture, up to 144 SMs are possible [7].

In the Hopper architecture, NVIDIA also introduced a new level of hierarchy which allows thread blocks to be grouped into so-called TBCs. Blocks of a cluster are guaranteed to be co-scheduled on the same GPC. Similar to a single thread block, a TBC can enforce synchronization barriers for all of its thread blocks. Data can also be exchanged directly between participating thread blocks via so-called DSMEM. This allows developers to use a more granular level of locality than previously provided by thread blocks alone. Figure 1 shows that the SMs of a GPC are connected by an SM-to-SM network, which allows direct communication between them, bypassing the L2 cache and reducing latency. The dimensions of a cluster can either be specified by a function attribute at compile time or at runtime using the `cudaLaunchKernelEx` API call. New special registers are available that allow the determination of cluster dimensions and index. To synchronize and exchange data, the following two CUDA API [8] calls were added:

- static void sync(): Synchronizes all blocks in the cluster.
- static T* map_shared_rank(T *addr, int rank): Returns the address of a shared memory variable of another block.

These APIs make use of the following PTX [9] instructions:

- barrier.cluster.arrive: Marks that a warp has arrived at the barrier, without pausing it.
- barrier.cluster.wait: Pauses a warp until all warps have reached barrier.cluster.arrive.
- mapa: Returns the generic address of a shared memory address for a given thread block rank.

III. Benchmarks

In this section, we introduce the benchmarks and microbenchmarks used in our study, organized into two subsections. The first subsection describes the microbenchmarks designed to measure latencies, cluster launch overhead, and data transfer patterns. The second subsection focuses on how TBCs were integrated into real-world applications for performance evaluation.

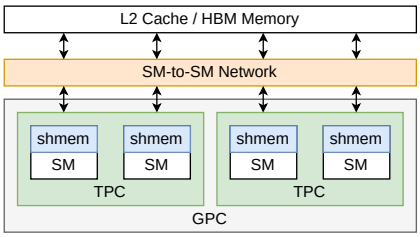


Fig. 1: SMs of a GPC are connected via an SM-to-SM network, enabling inter-thread block communication.

A. Microbenchmarks

1) Cluster launch: When multiple blocks are grouped into a cluster, the block scheduler faces a new constraint: all blocks within a TBC must be scheduled on the same GPC. If a GPC does not have sufficient capacity, the TBC cannot be assigned to it.

```
cudaEventRecord(start);
cudaLaunchKernelEx(&config, kernel);
cudaEventRecord(end);
cudaDeviceSynchronize();
cudaEventElapsedTime(&time, start, end);
```

Listing 1: Measuring the execution time of a kernel.

To measure the overhead associated with launching a TBC, we run identical kernels with the same grid and block dimensions, varying only the cluster dimensions. To ensure that blocks do not terminate immediately after being scheduled, each block sleeps for 100,000 cycles before the completion. To measure the execution time of kernels, we used CUDA events shown in Listing 1.

Additionally, we want to understand how many clusters can be launched in parallel and the resulting SM utilization on the GPU. To achieve this, we launch multiple clusters with different cluster sizes and read the `%smid`-register. This special register "returns the processor (SM) identifier on which a particular thread is executing" [9, Section 10.8], allowing us to determine on which SM each thread block gets scheduled.

2) API/PTX instruction latency: To measure the latency of the newly introduced API/PTX instructions, we use the built-in clock function shown in Listing 2.

```
auto start = clock();
asm volatile("barrier.cluster.arrive;");
auto time = clock() - start;
```

Listing 2: Measurement of the execution time inside the kernel.

3) Data transfer: One of the main use cases of the DSMEM is transferring data between two blocks in a producer consumer fashion. In this microbenchmark, we want to transfer data from block 1 to block 0 using different patterns and find the

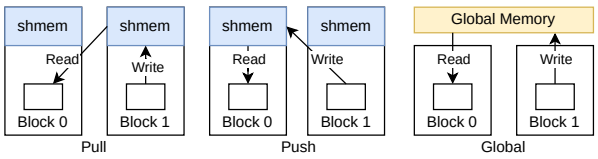


Fig. 2: Exchanging data between blocks.

most efficient pattern. Figure 2 shows the three patterns that we study. In the *pull* pattern, block 1 writes the data to its own shared memory, and block 0 reads the data from there. In the *push*, pattern, block 1 writes the data directly into block 0's shared memory. In the *global* pattern, the data is shared over the global memory. To make sure that the data is completely written, the blocks are synchronized using cluster synchronization after the write step.

We also ran the push-configuration with different underlying datatypes (`uint8_t` up to `uint64_t`) to check if stores using a wider bit-width affect the measured throughput.

4) 15-to-1 Copy: In a cluster of 16 thread blocks, one acts as the receiver while the remaining 15 blocks sequentially write a 2MiB block of dummy data to the receiver (push pattern). The benchmark can be run in *sequential mode* where all thread blocks write one after the other or in *simultaneous mode* where all thread blocks write data concurrently. The block designated as the "Receiver" can be configured. For measurements, we use the special register `%globaltimer`, which provides a nanosecond- resolution timer that updates every 32ns and, crucially, is consistent across all SMs. This consistency allows us to accurately compare timestamps and events occurring across multiple SMs, which is crucial for analyzing TBCs. The goal of the benchmark is to evaluate the bandwidth of each thread block relative to the others and assess how the DSMEM behaves under light network load (sequential mode) and heavy network load (simultaneous mode).

B. Application Case Studies

To demonstrate the use cases of TBC, we modified the following applications to leverage this feature:

1) Histogram (HG): In a histogram, we count the frequency of values in an array, with these values referred to as bins. In a parallel histogram, each block reads a portion of the input array and performs the histogram calculation in its local shared memory. At the end, the bin values are aggregated in the global memory. If the input contains more bins than can fit in the shared memory of a single SM, DSMEM can be used to distribute the bins across multiple blocks within the same cluster. We compare the cluster version against a version, where all the bins are stored in the global memory.

2) Batched Fast Walsh Transform (FWT): The FWT is a generalized class of the FFT. In our study, we computed the FWT on a sequence of real numbers. In the batched FWT, a single block calculates one transform. If the sequence is too long to fit into the shared memory of a single block, the cluster version stores it in the DSMEM. For these cases, we compare the performance of the cluster version against a version where the sequences reside in the global memory.

3) Batched Fast Fourier Transform (FFT): FFT exhibits a memory access pattern similar to that of FWT, with the main difference being the use of complex numbers instead of real numbers. As with FWT, each block computes an entire FFT. If the input sequence exceeds the shared memory capacity of a single block, the data is stored in the DSMEM.

4) Automated Teller Machine (ATM): In the ATM simulation, a simplified financial scenario is modeled where a large number of wire transfers are processed concurrently on a GPU. Each transaction requires a thread to acquire two locks using atomic instructions: one for the source account and another for the destination account. If a thread fails to capture one of the locks, it waits up to a thousand cycles before attempting the transaction again.

We compare two implementations of ATM: one that synchronizes all account accesses over the global memory, and the other that performs synchronization using thread block clusters's DSMEM. Additionally, we run the benchmark with different numbers of bank accounts (512 vs. 8192) to assess how the different memory architectures behave under conditions of high and low lock contention, respectively.

IV. Experimental Setup

We conducted experiments on an NVIDIA H100 PCIe, which features 114 SMs, a 50MB L2 cache, and configurable shared memory of up to 228KB. The benchmarks were executed in a SLURM environment using CUDA 12.3. While the GPU model remained constant across all runs, the specific silicon varied depending on the node assigned by the SLURM scheduler. For the case studies, we evaluated the performance across various cluster dimensions. The input sizes, as shown in Table I, were chosen to ensure that each cluster dimension utilized the minimum number of blocks per cluster necessary to accommodate the input data. For the batched benchmarks, we used 10,000 batches to fully utilize all SMs of the GPU.

Cluster dimension	**2**	**4**	**8**	**16**
HG (Bins)	24K	48K	96K	192K
Batched FFT (Length)	8K	16K	32K	64K
Batched FWT (Length)	16K	32K	64K	128K

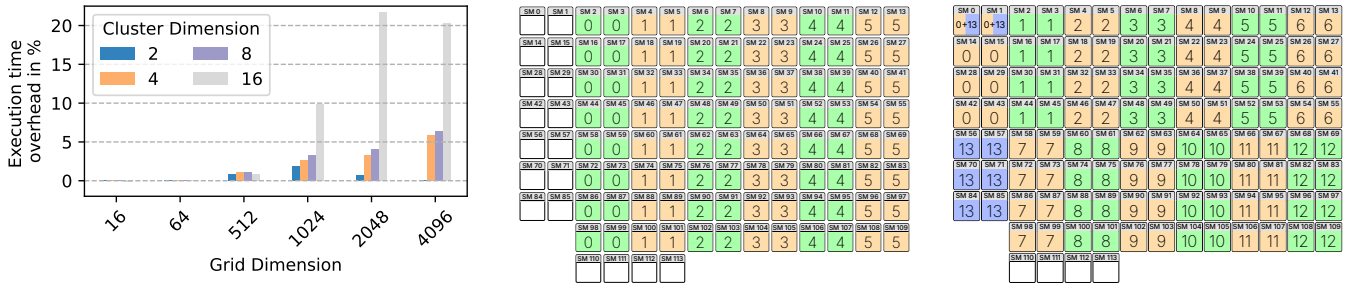
TABLE I: Input parameters used to run the benchmarks.

V. Results

We first present microbenchmarking results, followed by the results of real-world applications.

A. Microbenchmarks

1) Cluster Launch: Figure 3a shows the execution time overhead when launching a kernel using TBCs. For small grids, no overhead is noticeable, as the block scheduler can assign the TBCs to the GPCs without any additional delay. However, for larger grid dimensions where it is no longer possible to schedule all clusters simultaneously, a delay becomes noticeable. For the cluster dimension of two, the overhead



(a) Overhead of launching cluster kernels (b) Cluster dimension 16 (c) Cluster dimension 8

Fig. 3: Overhead of launching cluster kernels and assignment of thread blocks to physical SMs with varying cluster sizes. Figure *b* and *c* show the physical SMs indicated by `%smid` and assigned cluster IDs.

is negligible and within the range of measurement deviation. For cluster dimensions of four and eight, a delay of up to 5% is measurable. For clusters with 16 blocks, the overhead can reach up to 22%.

Figure 3b and 3c show the assignment of TBCs to the H100's physical SMs when using cluster sizes 8 and 16. Each box in the diagram represents one physical SM, with the `%smid` indicated at the top and the assigned cluster ID shown in the center. With a cluster size of 16, 18 SMs remain unused, whereas with a cluster size of 8, only 4 SMs remain unused. Additionally, SMs 0 and 1 are performing "double-duty" and are split across two different clusters. A cluster size of 4 (not shown in the figure) paints a similar picture to Figure 3c, with SMs 110-113 remaining unused, while a cluster size of 2 does achieve 100% SM-utilization. Therefore, it's crucial to carefully consider the choice of a cluster size. Additionally, we made two general observations:

- Multiple thread blocks that belong to the same cluster are *never* assigned to the same physical SM.
- The rank of a thread block within its cluster *directly corresponds* to its assignment on a physical SM. For example, in Figure 3b, this means that in cluster ID 0, the thread block with rank 2 was consistently assigned to physical SM 16.

2) API/PTX instructions: Figure 4 shows the amount of cycles it takes to execute the PTX instructions/API calls. Figure 4a to 4d use a block dimension of 1 thread to avoid measuring intra-block synchronization. Figure 4e to 4h use the maximum block dimension of 1024. For 1 thread per block, the median latency for the arrive instruction is 1050 cycles for the cluster dimension 1 to 8. For 16 blocks per cluster, it is 150 cycles higher. The wait instruction takes 45 cycles to execute for a single block; two blocks do not significantly increase the execution time. With more than 4 blocks per cluster, the median execution time increases up to 87 cycles.

The CUDA API call `cluster.sync()` gets translated to an arrive and a wait instruction by the compiler. The execution time is the sum of both instructions with a median of up to 1300 cycles. In contrast, using `__syncthreads()`, which synchronizes the threads within a single block, takes 14 cycles for similar block dimensions.

Prior to the Hopper architecture, one way to synchronize multiple thread blocks in a grid was by using `grid.sync()`. For similar grid and block dimensions, the `grid.sync()` takes approximately 2200 cycles. However, `grid.sync()` can handle synchronization for a significantly larger number of blocks compared to the maximum cluster dimension of 16.

For a block size of 1024 threads (Figure 4e to 4g), the behavior is mostly similar, where increasing the number of blocks per cluster also increases latency. However, there is a noticeable increase in cycles required for intra-block synchronization in general.

The latency of `cluster.sync()`(1300 cycles) is 40.9% less than that of `grid.sync()` (2200 cycles) and 92.8× more than that of intra-block synchronization using `__syncthreads()` (14 cycles). In other words, the latency of `cluster.sync()` is much more closer to that of `grid.sync()` rather than to `__syncthreads()`.

3) Data Transfer: Figure 5 shows the speedup of transferring n bytes from one block to another using *push* and *pull* mechanism compared to transferring data using the global memory (*global* mechanism). The fastest way to do transfer is when the data producing block directly writes the data into the target block (*push* mechanism). It is up to 230% faster for transferring 8192 bytes. If the producer writes the data into its own shared memory and the consumer fetches the data from there, the latency for smaller sizes (up to 128 bytes) is higher than doing the transfer over the global memory. For larger sizes, the global memory has the highest latency.

Table II shows the impact of the underlying datatype on data transfer. We observed that each store-instruction over the distributed shared memory incurs the exact same cost regardless of its associated datatype. Doubling the bit-width of the datatype halved the runtime of the data transfer. Therefore, the fastest way to transfer data is by using 64-bit data types.

Datatype	`uint8_t`	`uint16_t`	`uint32_t`	`uint64_t`
Runtime	154.11μs	77.06μs	38.59μs	19.36μs

TABLE II: Runtime of transferring data between two blocks using different datatypes. The block dimension is set to 32 threads and the total amount transferred is 256KiB.

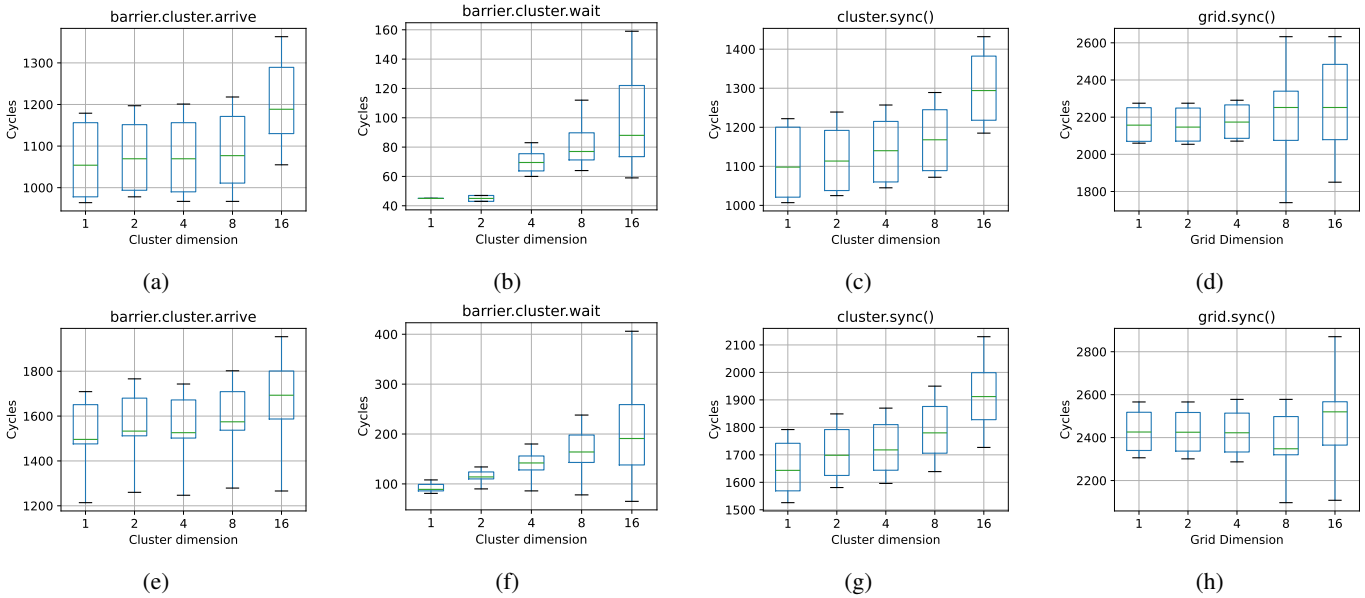


Fig. 4: Cycles required to execute PTX instructions/API calls with varying cluster dimensions. Figure *a* to *d* use a block dimension of 1 thread, *e* to *h* use 1024 threads.

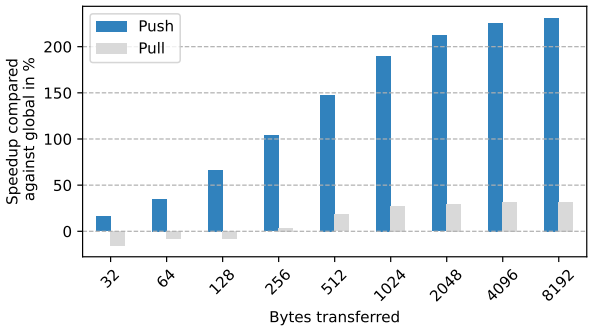


Fig. 5: Speedup of transferring data between two blocks. The block dimension is set to 32 threads.

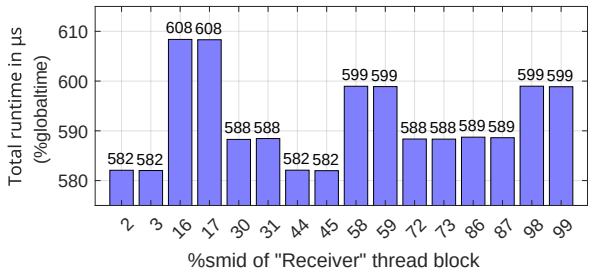


Fig. 6: Total runtime of the 15-to-1 copy microbenchmark depending on receiver's thread block rank (and physical SMID) for the simultaneous mode.

4) 15-to-1 Copy: When running the microbenchmark in *sequential mode*, where all thread blocks write their payload one after the other, all thread blocks exhibit the same bandwidth relative to the others. Regardless of the designated receiver thread block, each writer thread block needs around 308μs to complete sending its 2MiB payload. Running the benchmark in the *simultaneous mode*, however, provokes network congestion on the distributed shared memory, with some thread blocks finishing in 308μs while others need almost twice as long as 608μs. This affects the total runtime of the microbenchmark as shown in Figure 6. Depending on which of the 16 thread blocks is assigned the role of the "Receiver" in the cluster, the total runtime of the benchmark fluctuates between 582μs and 608μs, resulting in an increase or decrease of roughly 4%. Therefore, it is advisable for developers to experiment with different role assignments for thread blocks within a cluster, as this choice can significantly impact performance.

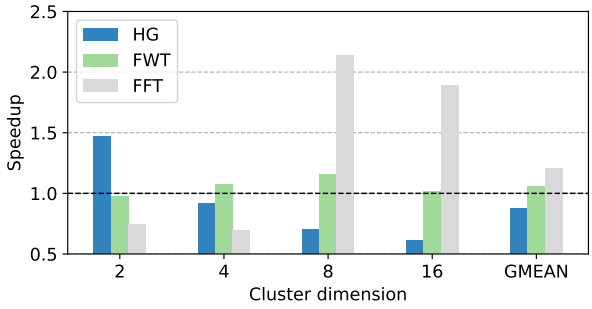
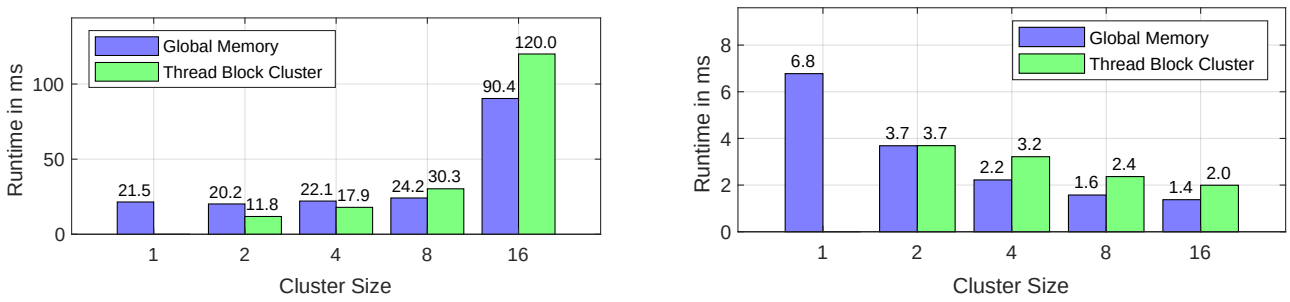


Fig. 7: Speedup of cluster applications using DSMEM compared to using global memory for different cluster dimensions.

B. Application Case Studies

Figure 7 shows the speedup achieved by using DSMEM instead of global memory for HG, FWT and FFT.



(a) 512 bank accounts

(b) 8192 bank accounts

Fig. 8: ATM results comparing global memory and distributed shared memory with 1024 threads/block and 100k transactions.

1) Histogram: For the histogram, the highest speedup is observed with 2 blocks per cluster, showing an improvement of up to 50%. However, if the data set is too large for two blocks, the histogram performs better when processed directly in the global memory rather than in the DSMEM .

2) Batched FWT: For the batched FWT, there is a 20% average improvement, but with a cluster dimension of 16, the performance matches that of the global memory version.

3) Batched FFT: For the batched FFT, the cluster version initially performs worse with smaller inputs but shows improved performance with larger sizes. At an FFT size of 32K, it is 2.1× faster than the global memory version. This improvement is because the cluster version stores the entire FFT in shared memory, rendering the cache less relevant.

4) ATM: Figure 8 shows the results of the ATM microbenchmark for 512 bank accounts where lock contention is high (left) and 8192 bank accounts where lock contention is low (right). In scenarios with low lock contention, clustering does not provide any measurable benefit over the global memory in our testing. As shown in Figure 8b, the global memory solution either matches or outperforms the clustered implementation. If lock contention is very high, clustering can provide improvements. Figure 8a shows that when deploying two thread blocks in a cluster, the overall runtime improves and outperforms the global memory version by roughly 1.7×.

VI. Related Work

There have been various microbenchmarking studies to evaluate system [10]–[12] and application [13] performance. Some of these studies have focused on GPU architectures [14]–[17], including the work of Abdelkhalik et al. [18], which focused on the Ampere architecture. In their study, the authors measured the clock cycles per instruction for the PTX ISA and the corresponding SASS instructions. Recently, Luo et al. [19] performed a study which mircobenchmarked the NVIDIA Hopper architecture. While it focused on several novel features of the Hopper architectures such as FP8 support, DPX instructions, tensor cores, the tensor memory accelerator and distributed shared memory, the main performance characteristics of TBC, such as cluster synchronization latency, cluster launch overhead, and optimal communication patterns remain unclear. In contrast, our work focuses on a detailed analysis of TBC, revealing its main performance characteristics required for the optimal use of this new feature, as well as for accurate analytical and cycle-accurate simulator models. Shan et al. [20] integrated TBC into stencil kernels and found that minimizing data exchange between shared memory of different blocks resulted in negligible performance benefits.

VII. Conclusion

This paper presents a comprehensive analysis of the TBC feature introduced in NVIDIA's Hopper architecture, providing key insights into its performance characteristics and practical implications. We specifically benchmarked instruction latencies, offering valuable information for developers to anticipate TBC behavior and for GPU architects to develop accurate analytical and cycle-accurate simulation models. Our study demonstrates the potential performance improvements and degradations developers can expect when using different data transfer patterns. The most efficient method of transferring data between two blocks is the push pattern, where data is written directly from the producer to the consumer's shared memory, achieving a 2.3× speedup. In contrast, the pull pattern, where the consumer reads data from the producer's shared memory, performs significantly worse. Additionally, applications constrained by shared memory can achieve up to a 2.1× speedup by employing TBCs. However, our findings also reveal that utilizing large cluster dimensions can result in an execution time overhead exceeding 20%, and large cluster sizes can leave some SMs idle, leading to suboptimal resource utilization. Therefore, it is crucial to carefully consider the choice of cluster size and evaluate the potential benefits of launching an auxiliary kernel with a smaller cluster size or no clustering at all to fully utilize all available processors. Moreover, while the use of atomics in TBCs can provide performance improvements (1.7×) in cases of high lock contention, their use should be carefully considered, as performance might degrade in other scenarios. The insights provided in this paper equip developers with the knowledge needed to harness the full potential of modern GPUs and assist researchers in developing accurate analytical and cycle-accurate simulation models, thereby advancing GPU architecture research.

REFERENCES

[1] M. et al., "TOP500," 2023. https://www.top500.org/.

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, 2008.

[3] NVIDIA, "Hopper Architecture White Paper," 2022. https://www.hpctech.co.jp/catalog/gtc22-whitepaper-hopper_v1.01.pdf.

[4] NVIDIA, "Fermi Architecture White Paper," 2009. https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[5] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 473–486, IEEE, 2020.

[6] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE international symposium on performance analysis of systems and software*, pp. 163–174, IEEE, 2009.

[7] "Nvidia hopper architecture in-depth," Mar. 2022.

[8] NVIDIA, "Cuda c++ programming guide," 2024.

[9] NVIDIA, "Parallel thread execution isa version 8.5," 2024.

[10] B. Bershad, R. P. Draves, and A. Forin, "Using microbenchmarks to evaluate system performance," in *[1992] Proceedings Third Workshop on Workstation Operating Systems*, pp. 148–153, IEEE, 1992.

[11] R. Hoque and P. Shamis, "Distributed task-based runtime systems-current state and micro-benchmark performance," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 934–941, IEEE, 2018.

[12] S. Lal, A. Alpay, P. Salzmann, B. Cosenza, N. Stawinoga, P. Thoman, T. Fahringer, and V. Heuveline, "SYCL-Bench: A Versatile Single-Source Benchmark Suite for Heterogeneous Computing," in *International European Conference on Parallel and Distributed Computing, (Euro-Par)*, Springer International Publishing, 2020.

[13] M. Grambow, D. Kovalev, C. Laaber, P. Leitner, and D. Bermbach, "Using microbenchmark suites to detect application performance changes," *IEEE Transactions on Cloud Computing*, vol. 11, no. 3, pp. 2575–2590, 2022.

[14] M.-M. Papadopoulou, M. Sadooghi-Alvandi, and H. Wong, "Microbenchmarking the gt200 gpu," *Computer Group, ECE, University of Toronto, Tech. Rep*, 2009.

[15] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 235–246, IEEE, 2010.

[16] R. Taylor and X. Li, "A micro-benchmark suite for amd gpus," in *2010 39th International Conference on Parallel Processing Workshops*, pp. 387–396, IEEE, 2010.

[17] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2016.

[18] H. Abdelkhalik, Y. Arafa, N. Santhi, and A.-H. A. Badawy, "Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8, IEEE, 2022.

[19] W. Luo, R. Fan, Z. Li, D. Du, Q. Wang, and X. Chu, "Benchmarking and Dissecting the Nvidia Hopper GPU Architecture," Feb. 2024. arXiv:2402.13499 [cs].

[20] B. Shan and M. Araya-Polo, "Evaluation of programming models and performance for stencil computation on current gpu architectures," *arXiv preprint arXiv:2404.04441*, 2024.