ClusterSim: Modeling Thread Block Clusters in Hopper GPUs

Tim Lühnen*, Jyotirman Behera†, Devashree Tripathy† and Sohan Lal*
*Hamburg University of Technology, Germany
†Indian Institute of Technology Bhubaneswar, India
*{tim.luehnen, sohan.lal}@tuhh.de
†{a25cs09003, devashreetripathy}@iitbbs.ac.in

***Abstract*—Modern Graphics Processing Units (GPUs), such as NVIDIA's Hopper and Blackwell, leverage Thread Block Clusters (TBCs) to enhance performance and resource management. TBCs introduce a hierarchical organization, grouping thread blocks into clusters that enable efficient synchronization and distributed shared memory access. This innovation improves data locality and reduces latency in inter-thread block communication, unlocking new opportunities for executing complex parallel workloads. However, modeling the intricate interactions within TBCs, especially the balance between data locality and resource contention, is challenging. This is further complicated by limited access to cutting-edge hardware like Hopper GPUs, which restricts direct experimentation. As a result, robust simulation models are needed to accurately replicate TBC behavior. This paper presents a detailed simulation model that captures TBC performance characteristics. Our model enables researchers to explore TBC functionalities and evaluate performance implications without requiring physical Hopper GPUs. Validation against an NVIDIA H100 GPU shows a Mean Absolute Relative Error (MARE) of 4.7%, demonstrating the model's accuracy and utility for advancing research in GPU architectures and parallel computing.**

***Index Terms*—GPUs, thread block clusters, computer architecture modeling and simulation, Hopper architecture**

I. Introduction

The rapid advancement of GPUs has revolutionized High-performance computing (HPC), enabling unprecedented acceleration for data-parallel workloads in fields such as artificial intelligence, scientific simulations, and large-scale data analytics. As the demand for computational power continues to surge, GPU architectures have evolved to incorporate increasingly sophisticated features aimed at maximizing throughput, energy efficiency, and programmability. Among these innovations, the introduction of TBCs in NVIDIA's Hopper architecture represents a significant leap, allowing for enhanced inter-thread block communication and improved exploitation of data locality. Since the first general-purpose GPU (NVIDIA Tesla in 2006 [20]) to the newest (Hopper in 2023 [26]), NVIDIA has consistently improved GPU architectures to enhance performance, energy efficiency, and overall capabilities. While some advancements, such as increased core counts, higher memory bandwidth, and the introduction of caches (Fermi architecture [25]), offer transparent performance gains, others like tensor cores, tensor memory accelerator, TBCs, and grid synchronization necessitate substantial application modifications. This is a complex task, as applications are often developed and maintained by several programmers over time, and without these code changes, the performance and energy efficiency benefits of these architectural innovations cannot be realized. Enhancing performance per GPU can reduce the number of GPUs needed or enable larger model training within the same hardware constraints.

An important trend in GPU architecture evolution is the increasing number of SMs, growing several folds ($9\times$) from Tesla to Hopper architecture. Due to the large increase in the number of SMs and application complexity, a thread-block as the only unit to express locality in applications is not deemed enough. To express and exploit locality larger than a thread block, NVIDIA introduced a new optional level of hierarchy in the CUDA programming model, called TBC in the Hopper Architecture [28]. All thread blocks in a cluster are guaranteed to be concurrently scheduled on a group of SMs known as GPU processing cluster (GPC). These thread blocks can directly access each other's shared memory via an SM-to-SM network, forming a distributed shared memory (DSMEM) model. This enables efficient intra-cluster data sharing and reduces the latency compared to accessing global memory.

However, the potential benefits of TBC for various applications remain largely unexplored. While applications with large data that exceeds the shared memory capacity of a single SM are expected to be the prime beneficiaries, there are other use cases, including those involving frequent inter-thread block communication. Moreover, the extent of performance improvement across different applications is not known. Furthermore, while the SM-to-SM network enables clustering, network congestion can also degrade the performance. In addition, the TBC feature is only available on NVIDIA's Hopper and Blackwell GPUs. As a result, many researchers and developers are unable to leverage this advanced feature. As an alternative, cycle-accurate simulators can be used for evaluating the effectiveness of architectural changes. The most widely used simulator for NVIDIA GPUs is GPGPU-Sim [15]. However, GPGPU-Sim does not support kernels using TBC and lacks a simulation model for the SM-to-SM network. This limitation poses a challenge for researchers to study the

This research is partially supported by the following grants: NSF-MeitY Grant (e-file: 3149156), ANRF Grant (ANRF/ECRG/2024/006088/ENS), MeitY Grant (VARCOE/23/02), and IITBBS Project RP433. The authors would like to thank the anonymous reviewers for their invaluable comments and suggestions.

performance implications of these new architectural features.

In this paper, we address these challenges by extending GPGPU-Sim to support the execution of TBCs by implementing key parallel thread execution (PTX) instructions required for their functionality. We further adapt the thread-block scheduler to capture the unique constraints of TBCs, enabling accurate simulation of their behavior. To analyze inter-SM communication, we conduct microbenchmarking of the SM-to-SM network to investigate its characteristics, including a topology that closely resembles the Hopper architecture.

By addressing the limitations of the state-of-the-art simulator and providing insights into the impact of architectural changes, our research contributes to advancing GPU architecture research and enables developers to optimize applications with the TBC feature without relying on costly Hopper GPUs. While there are existing studies for shared L1 caches for GPUs, cooperative caching network, and co-scheduling thread-blocks in the same streaming multiprocessor (SM) [11], [13], [18], to the best of our knowledge, there is no prior work on the performance evaluation of TBC in GPUs. Moreover, the state-of-the-art techniques for shared L1 caches are transparent to applications, while TBC optimization requires that the data sharing is explicitly managed by programmers. In summary, we make the following main contributions in this paper:

- We developed a simulation model capable of executing CUDA kernels using TBCs, achieving a mean absolute relative error of 4.7% across a range of benchmarks.
- To support TBCs simulation, we implemented key PTX instructions, designed a custom thread block scheduler, and benchmarked the SM-to-SM network to understand its topology and performance characteristics, which we then accurately modeled within the simulator.
- We infer that the SM-to-SM network within a cluster resembles a crossbar-like topology, based on uniform inter-SM latencies, equitable bandwidth sharing under broadcast and pairwise access, and the ability to sustain multiple concurrent, non-interfering data paths, characteristics inconsistent with mesh or ring-based designs.
- We demonstrated that reducing DSMEM latency by 50% improves performance by 4.2%, while an ideal DSMEM achieves an average speedup of 14.7%, indicating potential for future improvements. Notably, some microbenchmarks achieve a $1.87\times$ speedup when bandwidth is doubled, and up to $10.5\times$ under ideal conditions.

The paper is organized as follows: Section II provides the background, Section III details the architecture, Section IV outlines the experimental setup, Section V presents the results, and Section VII concludes the paper.

II. Background

The CUDA programming model organizes threads into thread blocks. In a thread block, threads can exchange data using on-chip shared memory, and they can synchronize using synchronization barriers. For inter thread block data exchange in pre-Hopper GPUs, the producing thread block needs to write the data to the global memory from where the consuming thread block can read it. Figure 1a illustrates the organization of current NVIDIA GPUs. When running a CUDA Kernel the thread blocks are assigned to SMs. These SMs are grouped into texture processing clusters (TPCs). Multiple TPCs are further grouped into GPCs. NVIDIA GPUs with compute capability 9.0 introduced the concept of TBCs. Thread blocks within the same cluster are allocated to the same GPC. The SMs of a GPC are interconnected using an SM-to-SM network. Threads can read/write from/to the shared memory of thread blocks that belong to the same TBC using this direct network. This direct communication link allows bypassing the L2 cache. To synchronize threads within a cluster, the newly hardware accelerated synchronization barriers can be used.

Cycle-accurate simulators play a crucial role in GPU architecture research by providing a platform for testing and validating new features and configurations without the need for physical hardware. GPGPU-Sim [15] is one of the most widely used open-source cycle-accurate simulator for NVIDIA GPUs. It models the detailed architecture and behavior of GPUs, allowing researchers to analyze the performance implications of various architectural changes. It works by parsing the PTX of CUDA applications and performing a functional simulation to emulate the execution of instructions. The simulation data is then fed into a performance model, which performs the cycle-accurate simulation. This approach helps in understanding how different aspects of the GPU architecture, such as memory hierarchy and interconnect networks, impact overall performance. However, GPGPU-Sim has key limitations: it lacks support for PTX instructions related to TBC operations and does not model an SM-to-SM interconnect. As a result, it cannot simulate benchmarks that rely on TBC features introduced in NVIDIA's Hopper architecture.

III. ClusterSim Modelling Framework

To model TBCs in GPGPU-Sim 4.0 [15], four key components need to be modeled, 1) PTX instructions related to TBCs, 2) the thread block scheduler, 3) the synchronization barriers, and 4) the SM-to-SM network. In the following, we detail the modeling approach for each of these components.

A. PTX Instructions Related to TBC

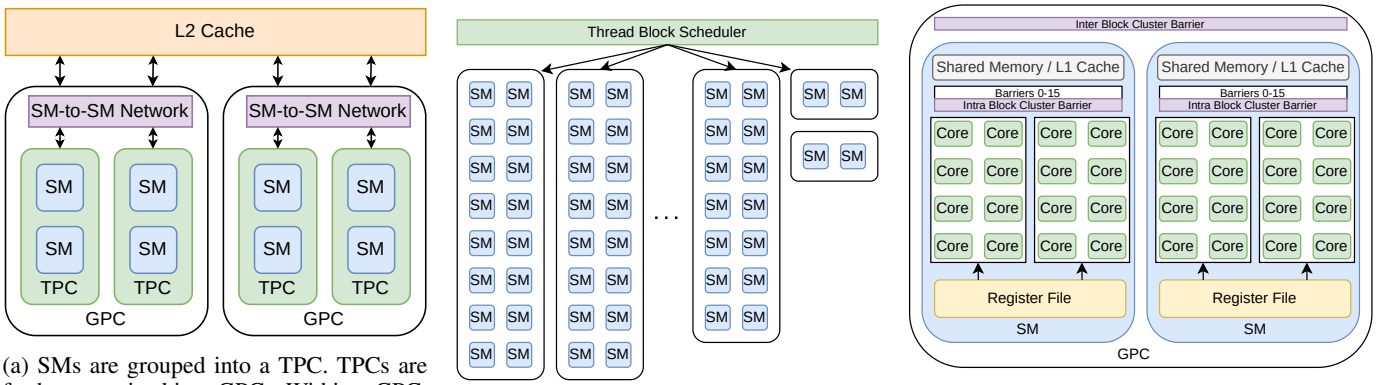
```
1  bar.sync 0;                 // Synchronizes threads
2  // within a block
3  ld.shared.u32 r1, [addr];   // Load from shared memory
```

Listing 1: PTX Code without TBCs

```
1  barrier.cluster.arrive;  // Arrive at cluster barrier
2  // (non-blocking)
3  barrier.cluster.wait;    // Wait for all blocks in
     cluster
4  mapa.u64 r3, [addr], r2; // Map shared memory addr for a
5  // block rank
6  ld.u32 r4, [r3];         // Load from mapped shared
     memory
```

Listing 2: PTX Code with TBCs

Without TBCs, synchronization and shared memory access are limited to threads within a single block, using `bar.sync` for synchronization and `ld.shared`



(a) SMs are grouped into a TPC. TPCs are further organized into GPCs. Within a GPC, SMs communicate directly via the SM-to-SM network.

(b) SMs are grouped into clusters of 2 to 16 SMs as per [21].

(c) New synchronization barriers added at the SM level for intra-block and one at the GPC level for inter-block synchronization.

Fig. 1: Overview of a modern GPU (e.g., Hopper) and key changes for TBC simulation.

for memory access as shown in Listing 1. With TBCs, new PTX instructions enable synchronization and data exchange across multiple blocks within a cluster as shown in Listing 2. The `barrier.cluster.arrive` and `barrier.cluster.wait` instructions provide cluster-level synchronization, while special registers like `%clusterid` and instructions such as `mapa` allow blocks to identify their cluster and access shared memory mapped across the cluster. This extension enables more flexible and scalable parallelism by supporting inter-block communication.

To enable the simulation of TBCs, we extend the PTX instruction set [2]. Our main contribution is the implementation of *new PTX instructions*, *directives*, and *special registers* introduced with compute capability 9.0 [29], which are essential for supporting TBCs. These extensions allow GPGPU-Sim to accurately simulate TBCs behavior while maintaining compatibility with recent architectural features. We outline the new PTX elements added to the simulator below.

Instructions: Following instructions enable direct data exchange and synchronization among thread blocks within the same cluster.

- `barrier.cluster.arrive`: Marks that a thread has arrived at the cluster barrier without stalling it.
- `barrier.cluster.wait`: Stalls the thread until all threads of the cluster have reached `barrier.cluster.arrive`.
- `mapa`: Returns the generic address of a shared memory address for a given thread block rank.

Directives: PTX directives provide metadata and control information regarding clusters that guide the assembler, compiler, or runtime execution.

- `.explicitcluster`: Specifies that a kernel needs to be launched with explicit cluster details.
- `.maxclusterrank`: Maximum number of blocks per cluster.
- `.reqnctapercluster`: Declares number of blocks per cluster.

Special registers: Special registers store hardware-specific information, such as cluster IDs, number of clusters, and execution states. These registers allow efficient cluster coordination and enable direct access to low-level GPU features. The following special registers were implemented:

- `%clusterid`: Unique ID of a cluster in the grid.
- `%nclusterid`: Number of clusters in the grid.
- `%cluster_ctaid`: Thread-block ID in the cluster.
- `%cluster_nctaid`: Number of thread-blocks in the cluster.
- `%is_explicit_cluster`: Gets set by the `.explicitcluster` directive.

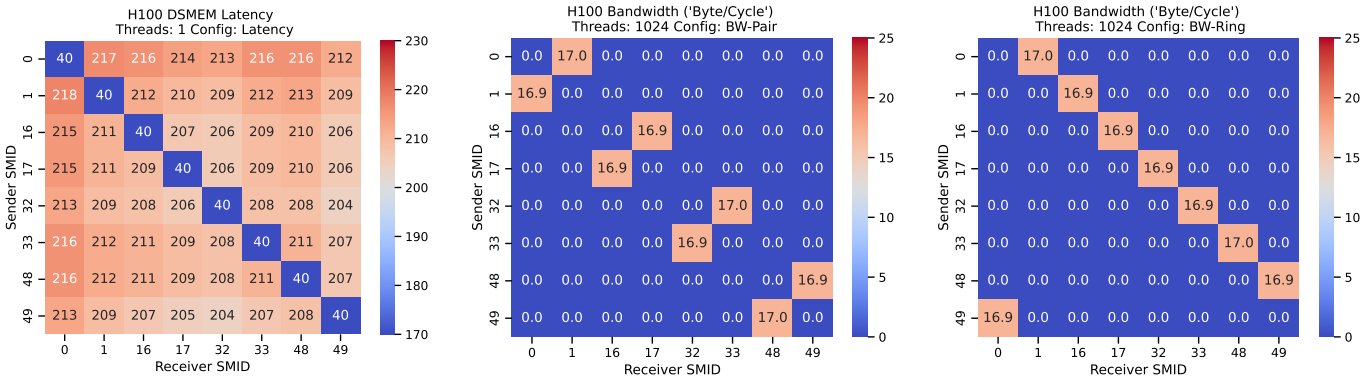
B. Thread Block Scheduler

When launching a TBC, the thread block scheduler must adhere to a new constraint: all thread blocks within the same TBC must be assigned to the same GPC. Consistent with prior work [21], thread blocks are assigned to the GPCs in a round-robin manner. dimensions that exceed their capacity. However, with large cluster dimensions, an additional constraint emerges—some GPCs lack the capacity to execute kernels whose cluster dimensions exceed their architectural limits.

The study also revealed that thread blocks within the same TBC are never scheduled to the same SM, ensuring spatial separation. According to their findings, there are six GPCs capable of running a TBC with 16 thread blocks. Figure 1b visualizes that the scheduler can only assign kernels with 16 blocks to a few GPCs, one GPC can run up to 14 thread blocks, and two others can accommodate only two blocks each. We grouped SMs into GPCs based on the findings of Luehnen et al. [21], employing a configuration parameter that specifies how many SMs are assigned to each GPC.

C. Synchronization Barriers

Thread blocks within the same TBC can synchronize using `barrier.cluster.arrive` and `barrier.cluster.wait` PTX instructions. The



(a) Latency (b) Pair-wise bandwidth (c) Ring-wise bandwidth

Fig. 2: Measured SM-to-SM (a) latency, and bandwidth under (b) pairwise and (c) ring-wise access patterns on H100.

`barrier.cluster.arrive` instruction marks a warp as having reached the synchronization barrier without stalling its execution, whereas the `barrier.cluster.wait` instruction stalls the warp until all warps in the TBC have executed the corresponding `barrier.cluster.arrive`.

As illustrated in Figure 1c, our implementation employs a two-level synchronization mechanism comprising a per-SM level barrier and a centralized GPC-level barrier. The SM-level barrier handles synchronization among warps within individual thread blocks. Once all threads within a block reach the SM barrier, a signal is sent to the GPC-level barrier to indicate that the block has reached the synchronization point. Execution resumes only after all thread blocks in the TBC have reached their respective barriers, at which point both the SM and GPC barriers are reset.

We measure the latencies of the synchronization instructions using microbenchmarks. We assigned a latency of 610 cycles to the `barrier.cluster.arrive` instruction to match the observed hardware behavior. For `barrier.cluster.wait`, we model a static base latency of 60 cycles, representing the minimum time required to execute the instruction. The higher latencies observed for larger cluster dimensions are attributed to synchronization delays as blocks wait for others to reach the barrier as the cluster size grows.

D. SM-to-SM Network

To generate a detailed fingerprint profile of the SM-to-SM network, we developed microbenchmarks that measure both *latency* and *bandwidth* between SMs within a TBC. *Pointer chasing* microbenchmark was employed to obtain accurate, fine-grained measurements.

1) Latency Microbenchmarking: To measure inter-SM latency, we developed a pointer-chasing microbenchmark similar to previous studies [22] that enforces serialized memory accesses between SMs. Each memory access is dependent on the result of the previous one, eliminating instruction-level parallelism and exposing the true communication latency. This approach allows us to isolate raw latency between pairs of SMs within a cluster.

In our setup, each SM takes turns acting as the reader, sequentially reading data from all other SMs in the cluster. This one-to-all probing reveals latency variation due to physical topology and routing. To minimize intra-block effects, only one thread per block was used during each latency test.

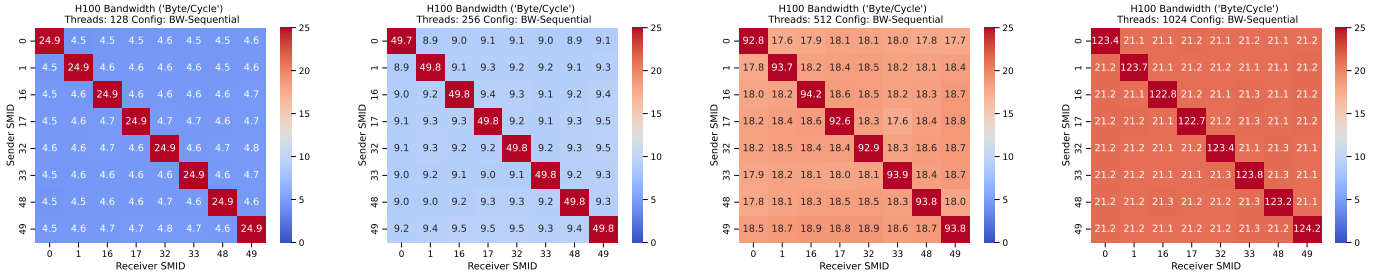
2) Bandwidth Microbenchmarking: To measure SM-to-SM bandwidth, we designed a parallel memory copy kernel in which threads in one SM read from shared memory regions located in other SMs. By maximizing concurrency and minimizing data dependencies, the microbenchmark aims to estimate the peak throughput of the interconnect. Our results reveal noticeable bandwidth asymmetries across different SM pairs, suggesting the presence of network contention or non-uniform load balancing behavior within the fabric.

The bandwidth microbenchmark follows a similar setup to the latency test, with the key distinction being the use of multiple threads per block to initiate concurrent memory transfers. Bandwidth is measured per SM across a variety of access patterns inspired by [22]. The objective is to saturate the interconnect, thereby exposing its performance characteristics and enabling inference of the underlying network topology.

- **Sequential access:** All SMs read from one another in sequence.
- **Pair-wise access:** SMs are grouped into pairs, with each SM in a pair reading from each other simultaneously.
- **Ring-wise access:** Each SM reads from the next SM in a circular fashion.
- **Broadcast access:** All thread blocks concurrently read from the same target SM.

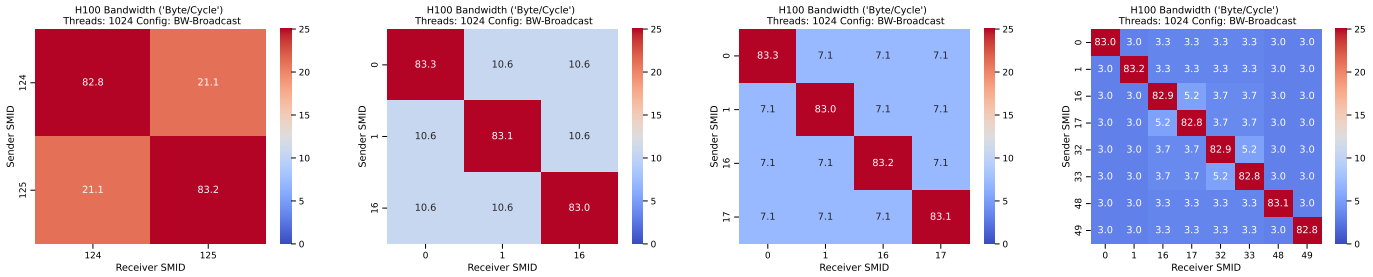
3) Measured Latency and Bandwidth:

a) Latency: The measured inter-SM latencies are organized in a matrix, where the *sender* SM ID is represented along the y-axis and the *receiver* (reader) SM ID along the x-axis. Figure 2a presents a heatmap of this matrix, highlighting spatial patterns in communication latency. The latencies range from approximately 204 to 218 cycles, exhibiting modest variation across SM pairs. Similar patterns were observed for



(a) 128 threads (b) 256 threads (c) 512 threads (d) 1024 threads

Fig. 3: Measured SM-to-SM bandwidth on H100 as a function of thread count per block in a sequential access pattern.



(a) 2 thread blocks (b) 3 thread blocks (c) 4 thread blocks (d) 8 thread blocks

Fig. 4: Measured SM-to-SM bandwidth on H100 under a broadcast access pattern, as the number of thread blocks reading from a single source block increases.

other cluster sizes. This uniformity suggests consistent routing and low contention under the pointer-chasing workload.

b) Bandwidth: Bandwidth was measured under increasing thread counts to analyze saturation behavior. Different access patterns reveal varying performance characteristics:

Sequential Access: As expected, bandwidth scales roughly linearly with thread count, for instance, doubling the number of threads results in a proportional increase in bandwidth as shown in Figure 3. However, beyond 512 threads, gains diminish. When increasing threads from 512 to 1024, the measured bandwidth only increases from 19.5 to 21.1 bytes/cycle, suggesting that the network is nearing its saturation point. Additional tests conducted with 768 threads (not shown due to space constraints) further support this trend, suggesting that saturation begins to occur even before reaching 1024 threads.

Pairwise Access: In this pattern, each SM communicates only with its designated partner. Figure 2b shows that despite the reduced contention, bandwidth remains lower than the maximum observed in the sequential test, reaching only 16.1 to 17 bytes/cycle, approximately 5 bytes/cycle less than when no other operations occur in parallel (sequential access case). This is likely due the concurrent traffic.

Ring-wise Access: Similar to the pairwise pattern, each SM reads from its immediate neighbor in a circular fashion. As shown in Figure 2c, this ring pattern results in slightly lower bandwidth than the pairwise case, likely due to a more uniform distribution of traffic across the network, which increases arbitration overhead and contention.

Broadcast Access: As more thread blocks concurrently read from a single SM, the bandwidth per reader decreases proportionally as shown in Figure 4. Doubling the number of readers roughly halves the per-thread-block bandwidth, indicating that total available bandwidth is shared equally among all requesters.

4) Topology Inference: Based on these observations, we conclude the following:

- **Latency** between SMs is fairly uniform, indicating consistent hop distances and efficient routing.
- **Bandwidth** scales with thread count up to a saturation point of $\sim$ 21.2 bytes/cycle. Once saturated, bandwidth is fairly divided among active SMs.
- **Parallel connections** (i.e., n disjoint sender-receiver pairs) can operate concurrently, each achieving the same bandwidth.

While ring-wise and pairwise access patterns exhibit minor bandwidth reductions, latency remains largely uniform, and multiple concurrent pairs achieve high bandwidth. This behavior is inconsistent with a pure *ring* or *2D mesh*, which would show greater bandwidth degradation with increasing concurrency or overlapping communication paths [4], [8], [16].

These findings suggest that the underlying SM-to-SM interconnect supports multiple simultaneous, non-interfering data paths, strongly indicative of a *crossbar-like* topology with round-robin arbitration or another form of equitable scheduling, ensuring uniform access to shared network bandwidth across the cluster. Also, as NVIDIA deploys other crossbars within an SM, it is more likely that a crossbar is also used for

the SM-to-SM network.

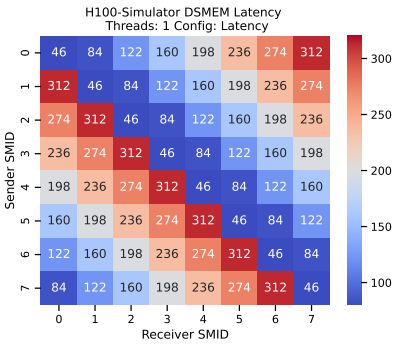


Fig. 5: Latency pattern of a ringbus as the SM-to-SM network.

We also implemented ringbus as an SM-to-SM network in the simulator. Figure 5 illustrates the DSMEM access latency for a ringbus. As the distance between the sender and receiver increases, the latency also grows, behavior that does not align with our observations on the H100 GPU.

While an ideal crossbar would support multiple concurrent SM pairs without bandwidth degradation, our results show a slight drop in the pair-wise and ring-wise compared to sequential access. This may be attributed to shared hardware resources between SMs within the same TPC, such as interconnect ports, or scheduler logic, which introduce contention even in the absence of topological bottlenecks. A fully connected network (where every SM has a direct link to every other SM) would produce very similar results, uniform low latency and consistent bandwidth across SM pairs. However, such a topology is rare in practice due to its high area and wiring cost [7], especially as the number of SMs scales.

To further rule out a fully connected network, we also evaluate an *all-to-all* communication pattern. In a thread block cluster containing N blocks, each block divides its threads into N equal groups. The first $\frac{1}{N}$ of the threads read from the first thread block, the second $\frac{1}{N}$ from the second block, and so on. This creates simultaneous, uniform communication between all thread blocks in the cluster. The bandwidth behavior resembles a broadcast (*many-to-one*) pattern. For example, in a cluster of 8 thread blocks (1024 threads each), the all-to-all pattern splits into 8 groups of 128 threads per block. The observed bandwidth matches that of a 128-thread broadcast, suggesting a crossbar-like SM-to-SM interconnect rather than a fully connected network.

In a fully connected network, bandwidth would remain stable during all-to-all communication because of dedicated links and the absence of contention. The observed bandwidth degradation and flattening therefore strongly suggest a shared crossbar interconnect with arbitration rather than a fully connected topology. To model contention between local shared memory requests and DSMEM requests, we implemented a parameterized arbitration mechanism that models the blocking of local shared memory requests while an incoming DSMEM request is being processed. This helped us to model a throughput drop under broadcast scenarios, closely matching the behavior observed in hardware.

IV. Experimental Setup

This section describes how we evaluated our model.

A. Simulator

We modify GPGPU-Sim 4.0 [15] to enable TBC modeling. We use a modified version of AccelSim Tuner [15] as well as custom microbenchmarks to generate a configuration file for H100 GPU. We conduct experiments using CUDA 12.8 on an NVIDIA H100 SXM, with key parameters listed in Table I.

Parameter	Value	Parameter	Value
#SMs	132	L1 $ size/SM	256kB
SM freq (MHz)	1620	L2 $ size	50MB
Max #Threads per SM	2048	#Memory controllers	32
Max CTA size	1024	Memory type	HBM
Shared memory/SM	228kB	DSMEM Network	Crossbar
Arrive Latency	610	Wait Latency	60

TABLE I: Main GPGPU-Sim configuration parameters for H100 (Hopper architecture) [1].

Arrive Latency and Wait Latency are the latencies of the `barrier.cluster.arrive` and `barrier.cluster.wait` instructions, respectively. The SM-to-SM network is modeled as a crossbar as per our findings in Section III-D. The simulator supports TBCs with cluster dimensions ranging from 2 to 16 SMs, as observed in prior work [21]. Additionally, the size of each GPC can be configured independently to support varying architectures.

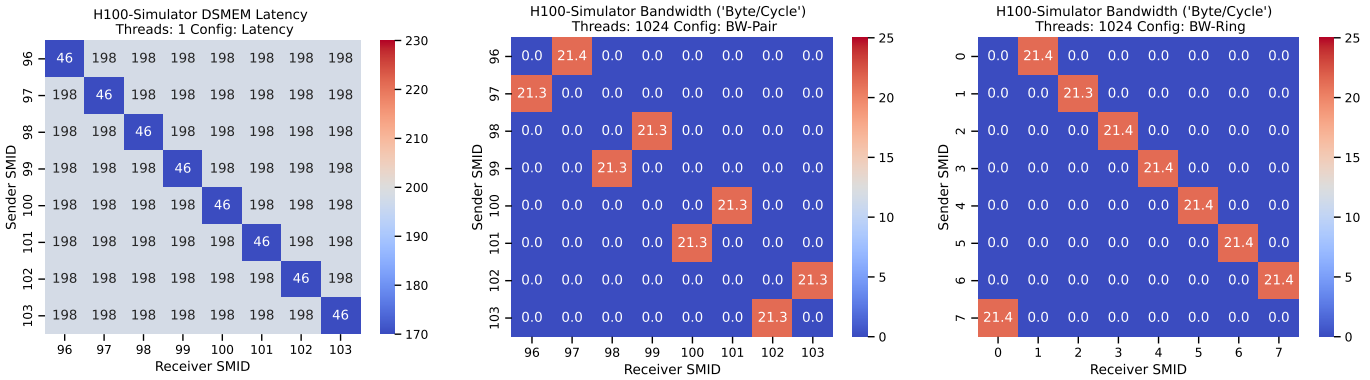
B. Benchmarks

As there is no existing benchmark suite, we optimized a diverse set of benchmarks to demonstrate varied usage scenarios. The four microbenchmarks serve as necessary building blocks for characterizing low-level behavior and validating simulator accuracy. To evaluate the accuracy of TBC modeling, we employed a total of thirteen workloads from [21], [27], Polybench [30] and ISPASS [3] benchmark suites. These workloads span a range of computational patterns and are summarized in Table II. *MM* has a 2D cluster with 2x2 thread blocks, while the other benchmarks use a 1D cluster. The benchmarks are categorized into two groups: microbenchmarks and applications from benchmark suites.

1) Microbenchmarks: We used a total of four microbenchmarks, each targeting a specific aspect of TBC functionality: *PUSH* and *PULL* are producer-consumer style microbenchmarks based on [21]. In the *PUSH* variant, the producing thread block writes data directly into the shared memory of the consuming thread block. Conversely, in the *PULL* variant, the consuming thread block actively loads data from the shared memory of the producer. The *LAT* and *BW* microbenchmarks are used to measure the latency and bandwidth as described in Section III-D.

Name	Abbreviation	GridDim	ClusterDim	BlockDim	Input Size
Benchmarks					
Histogram [27]	HG	132	2	512	10M array, 12800 bins
Fast Fourier Transform [21]	FFT	4	4	1024	16K sequence
Batched Fast Fourier Transform [21]	BFFT	80	4	1024	16K sequence, 20 batches
Fast Walsh Transform [21]	FWT	2	2	512	16K sequence
Batched Fast Walsh Transform [21]	BFWT	40	2	512	16K sequence, 20 batches
Breadth First Search [3]	BFS	8	8	1024	7115 nodes, 103689 edges
Matrix Multiply [27]	MM	16x16	2x2	16x16	256x256 Matrix
1D Convolution [30]	1DCONV	4	2	32	filter size 3, 4096 array
3MM [30]	3MM	16	4	1024	128 problem size
Microbenchmarks					
Push [21]	PUSH	2	2	32	8192 Bytes
Pull [21]	PULL	2	2	32	8192 Bytes
Latency	LAT	8	8	1	-
Bandwidth	BW	8	8	1024	-

TABLE II: Benchmarks. The *ClusterDim* column indicates the number of SMs in the cluster.



(a) Latency (b) Pair-wise bandwidth (c) Ring-wise bandwidth

Fig. 6: Simulated SM-to-SM (a) latency, and bandwidth under (b) pairwise and (c) ring-wise access patterns.

2) Applications from Benchmark Suites: We optimized five applications (listed in Table II) to utilize TBCs. We then compared the number of execution cycles reported by our modified simulator against those measured on actual hardware. For hardware measurements, we used NVIDIA Nsight Compute (NCU), leveraging its profiling capabilities to gather performance data.

V. Experimental Evaluation

This section presents our evaluation results. We begin by assessing how well the simulator captures SM-to-SM network behavior using targeted microbenchmarks. To assess overall simulation accuracy, we compare simulator-reported cycle counts with measurements from the NVIDIA NCU profiler. We first report results without performance modeling to establish a baseline, and then demonstrate the improvements achieved by incrementally enabling synchronization latency modeling and our proposed DSMEM crossbar model.

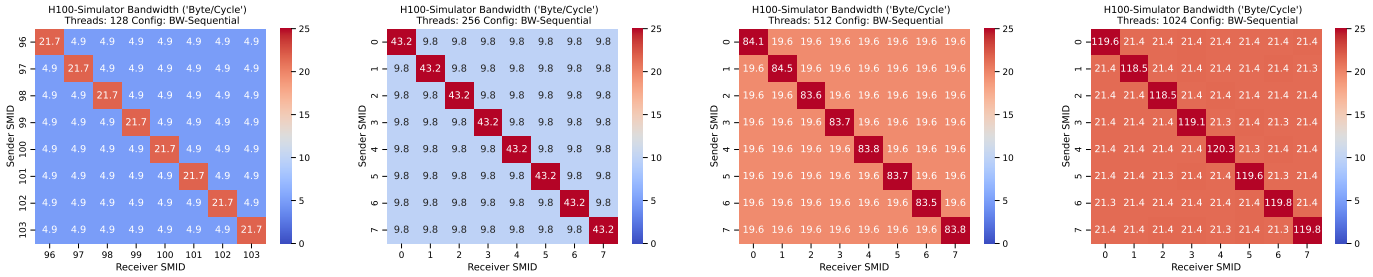
A. SM-to-SM Network Latency

To evaluate how accurately our SM-to-SM network model replicates real hardware behavior, we ran the latency and bandwidth microbenchmarks described in Section III-D on the modified simulator. While the SMIDs differ from those on actual hardware due to variations in block scheduling, this difference has no measurable impact on performance.

Figure 6a shows the results of the latency microbenchmark on the simulator. Our model reports a fixed latency of 198 cycles for all DSMEM connections. In contrast, real hardware exhibits latency variation, ranging from 187 to 218 cycles an aspect not captured by our model. This is due to the fixed latency for the DSMEM connections in our model. Additionally, the latency for local shared memory access in our simulator is 46 cycles, approximately 6 cycles higher than the value observed on real hardware. This discrepancy arises from the execution of the same microbenchmark as on the actual device, which introduces a few cycles of overhead due to additional instructions, such as loop control.

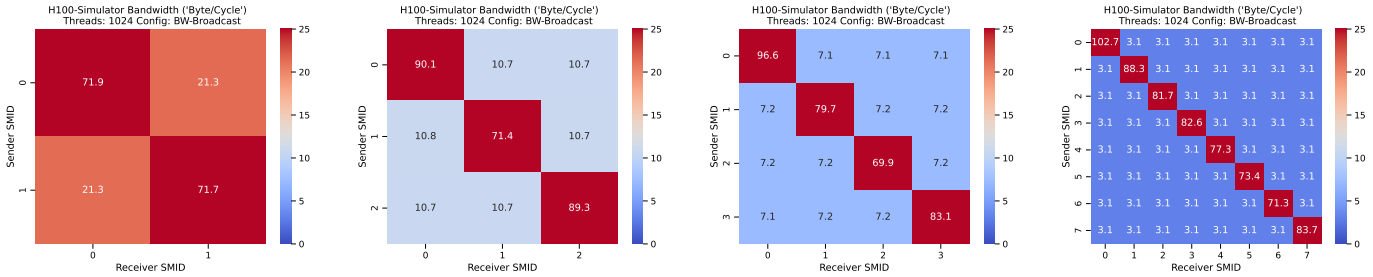
B. SM-to-SM Network Bandwidth

a) Sequential Access: Figures 7a to 7d present bandwidth measurements across increasing thread counts (sequential access pattern). With 128 threads per block, our model achieves 4.9 bytes/cycle, slightly above the 4.5 to 4.8 bytes/-cycle measured on real hardware. As expected, doubling the thread count doubles the bandwidth up to a point. Beyond



(a) 128 threads (b) 256 threads (c) 512 threads (d) 1024 threads

Fig. 7: Simulated SM-to-SM bandwidth as a function of thread count per block in a sequential access pattern.



(a) 2 thread blocks (b) 3 thread blocks (c) 4 thread blocks (d) 8 thread blocks

Fig. 8: Simulated SM-to-SM bandwidth under a broadcast access pattern, as the number of thread blocks reading from a single source block increases.

512 threads per block, bandwidth saturates at 21.4 bytes/cycle closely matching the hardware's observed peak of 21.1 to 21.3 bytes/cycle.

b) Broadcast Access: Figure 8 show results of the broadcasting bandwidth tests on the simulator. We see that a single reader achieves maximum bandwidth. Doubling the number of readers halves the bandwidth, from 21.4 to 10.7 bytes/cycle. With seven parallel readers, the bandwidth drops to 3.1 bytes/cycle per thread block. This aligns well with hardware results, where most measurements fall between 3.0 and 3.7 bytes/cycle, with a few outliers reaching 5.2 bytes/cycle. For the local shared memory, we also see the drop in performance when DSMEM access happens at the same time. In the sequential access we saw a bandwidth of 118.5 to 120.3 bytes/cycle, for the broadcast access the local shared memory bandwidth drops to 71.3 to 102.7 bytes/cycle.

c) Pair-wise and Ring-wise Access: For pair-wise and ring-wise communication patterns in Figure 6b and Figure 6c) respectively, our simulator sustains the peak bandwidth ($\sim$ 21.1 bytes/cycle) observed for sequential access pattern on the real hardware (Figure 3d). While real hardware exhibits a slight performance drop, approximately 5 bytes/cycle in the pair-wise case and up to 8 bytes/cycle for ring-wise communication, our model captures the overall throughput trends effectively. The difference in the simulation is due to the optimal arbitration assumed in our model, which simplifies network contention behavior while still preserving the general performance characteristics.

C. Correlation

We calculated correlation between the measured and simulated network microbenchmark results. The average Pearson correlation coefficient is very high ($\sim$ 0.99), indicating excellent alignment across diverse latency and bandwidth benchmarks. The MARE is low on average at approximately 9.5%. These findings confirm the simulator's high fidelity and reliability in modeling network microbenchmarks under varying configurations.

D. ClusterSim Accuracy

To evaluate the accuracy of ClusterSim, we executed a set of TBC benchmarks and compared the simulator-reported cycle counts with those measured using the NCU profiler. Figure 9 shows the relative error across three configurations: (1) without DSMEM latency (DSMEM latency equals the latency of local shared memory), (2) with a fixed static latency, and (3) with our proposed crossbar-based DSMEM model.

The results show that omitting DSMEM performance modeling leads to the highest relative error, with the simulator consistently underestimating cycle counts compared to real hardware. For instance, benchmarks such as *HG, BFS, MM*, and *3MM* exhibit errors up to 21.2%. The error is especially pronounced in microbenchmarks heavily dependent on DSMEM communication. The *PULL* microbenchmark shows an error of 48.2%. The *LAT* and *BW* microbenchmarks have the highest error of 67.8% and -90.6%, respectively. The MAREs is 22.8%.

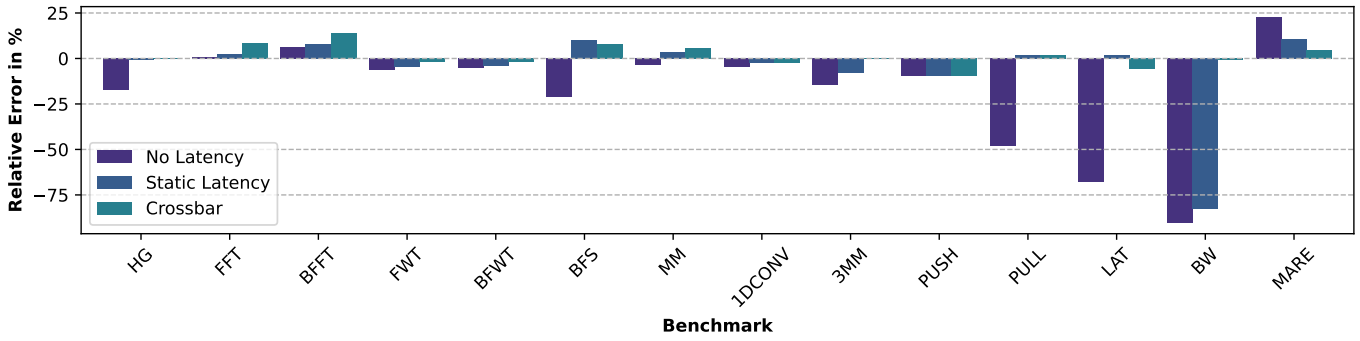


Fig. 9: Relative error in cycle counts between the simulator and NCU profiler across three configurations: (1) without DSMEM performance modeling, (2) with a static latency model, and (3) with the proposed crossbar-based model. The comparison highlights the accuracy improvements achieved through progressively more detailed DSMEM modeling.

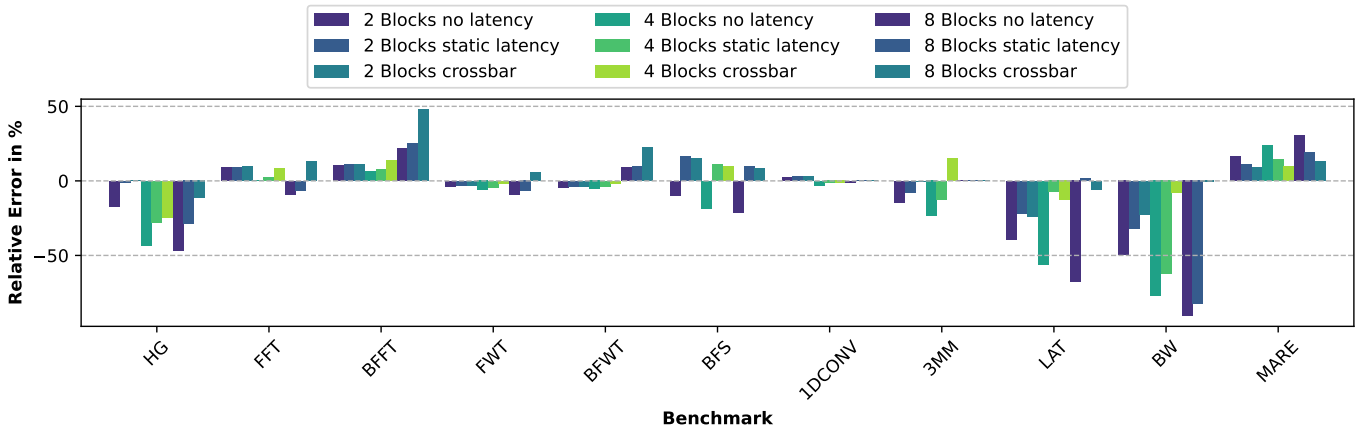


Fig. 10: Relative error as the number of thread blocks per cluster increases.

Incorporating a static latency model for DSMEM significantly reduces error, reducing the MARE across several benchmarks to below 10%. However, this approach falls short in bandwidth-sensitive cases. For instance, the *BW* microbenchmark still exhibits a large error of -82.6%, due to the static latency's inability to model peak bandwidth, yielding a MAREs of 10.8%.

Our crossbar-based DSMEM model achieves the highest accuracy, particularly in bandwidth-sensitive benchmarks such as *3MM* and *FWT*. It provides a substantial improvement in the *BW* microbenchmark, reducing the relative error to less than -1%. Overall, this model achieves a MARE of just 4.7%, demonstrating its effectiveness in capturing both latency and bandwidth characteristics of the SM-to-SM interconnect.

E. Sensitivity to Cluster Size

Figure 10 shows the relative error across different cluster sizes and DSMEM modeling approaches. Overall, the crossbar model consistently improves simulation accuracy compared to the no latency and static latency approaches. As cluster size increases, error tends to grow for all models due to amplified network contention, bandwidth sharing, and arbitration effects. However, the crossbar model maintains the highest accuracy on average.

For cluster size 2, the average MARE is about 10.9% for static latency and 9.3% for the crossbar model, compared to 16.1% for no latency. At cluster size 4, the crossbar model achieves 9.8% MARE, outperforming no latency (24.0%) and static latency (14.2%). For cluster size 8, crossbar's MARE is 12.9%, while no latency and static latency models show 30.6% and 19.0%, respectively, further confirming the crossbar model's superior accuracy.

For *BFFT*, the crossbar model's high error likely stems from factors unrelated to DSMEM modeling. Since even the no-latency and static latency models already exhibit high error, this suggests that other architectural or microarchitectural effects dominate the observed discrepancy, and the crossbar model cannot compensate for these. Our analysis revealed that this error is closely tied to the use of a large thread block size (TB size) of 1024 threads. When smaller block sizes are used, the model's accuracy improves significantly: for a cluster size of 8, the error drops from 45% to 13.1% and 5.0% for TB sizes of 512 and 256, respectively. This suggests that the current simulator may not fully capture the intricate prioritization and

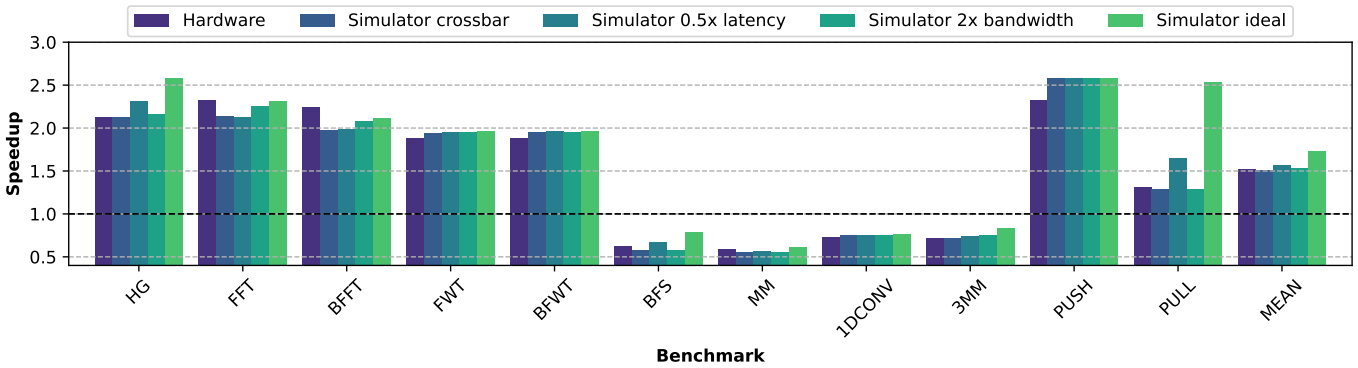


Fig. 11: Speedup of the benchmarks with TBC compared to benchmarks without TBC. The non-TBC versions were executed on real hardware. For the TBC versions, speedup is shown relative to execution on real hardware, on our simulator with the standard crossbar, on a simulator with a crossbar of 0.5× latency, with 2× bandwidth, and on an ideal network modeled to provide the same performance as local shared memory.

resource allocation mechanisms of the Hopper architecture. As the number of active warps increases, even minor inaccuracies in the simulator's scheduling logic can become magnified, resulting in noticeable deviations from actual hardware performance under high-contention scenarios.

In the *LAT* benchmark, crossbar and static latency models perform similarly, as it is not bandwidth-sensitive. Conversely, for the *BW* benchmark, the static latency model underestimates cycles because it ignores bandwidth constraints, while the crossbar model captures these effects, aligning better with hardware measurements. At smaller cluster sizes, *LAT* and *BW* exhibit larger errors since the overall benchmark runtime is shorter, making secondary effects more influential on the total runtime. Figure 10 does not include results for certain benchmarks that impose fixed cluster-dimension requirements: *PUSH* and *PULL* are restricted to two thread blocks, while *MM* requires a square-shaped cluster.

F. Performance Impact of Network Parameters

Figure 11 shows the speedup of TBC-optimized benchmarks relative to their non-TBC counterparts, using the non-TBC version on real hardware as the baseline. We evaluate speedup across several scenarios: (1) TBC-enabled benchmarks on real hardware, (2) on the simulator with our crossbar model, (3) on the simulator with a crossbar configured to 0.5× latency, (4) on the simulator with a crossbar configured to 2× bandwidth, and (5) in an ideal configuration where DSMEM behaves like local shared memory.

TBC-optimized benchmarks achieved notable speedups, up to 2.3× for several benchmarks (e.g., *FFT*, *FWT*, and *HG*) where DSMEM effectively leverages data locality and inter-block communication. However, some benchmarks like *BFS* and *MM* saw slowdowns (0.62× and 0.59×, respectively), reflecting the nuanced performance impact of DSMEM. On average, we observed a 1.52× speedup. Our simulator accurately captured both improvements and slowdowns, enabling systematic exploration of TBC behaviors, their limitations, and optimization strategies for applications.

To explore how changes in the SM-to-SM network configuration affect performance, we used the simulator to model alternative crossbar settings as shown in Figure 11. The results indicate that benchmarks such as *HG* and *BFS* gain speedups with reduced latency, while *FFT* and *FWT* benefit slightly from increased bandwidth. For latency-sensitive microbenchmarks, halving the DSMEM latency leads to a speedup of up to 1.6×. On average, halving the latency results in a 4.2% speedup compared to the crossbar simulator model, whereas doubling the bandwidth yields only a 1.7% improvement. With an ideal DSMEM configuration, the average performance increases by 14.7%. Since *LAT* and *BW* do not have non-TBC variants, they are omitted from the figure. We measured their performance relative to the TBC version running on the simulator. Adjusting the network parameters for these benchmarks has the most significant effect: halving latency results in a 1.5× speedup for LAT, while doubling bandwidth yields a 1.87× speedup for BW. Under ideal conditions, *BW* can even achieve up to a 10.5× speedup. These findings demonstrate the performance potential for benchmarks sensitive to DSMEM bandwidth and latency, highlighting opportunities for optimization in future GPU architectures.

VI. Related Work

Modeling Efforts for Hopper Architecture: The recent release of the NVIDIA Hopper architecture has spurred initial research to understand its performance characteristics. Elster et al. [12] provided an overview of the Hopper and Grace computer chips' new features. Luo et al. [22], [23] conducted benchmarking studies to measure inter-SM data transfer latency and bandwidth. Shan et al. [33] explored the use of TBCs for stencil computation but observed limited performance gains. These early studies highlight the nascent stage of Hopper architecture research and the need for more in-depth modeling.

General GPU Architecture Modeling and Simulation: Prior research has extensively explored modeling and

simulation for various GPU architectures. Studies have focused on performance simulation [5], [6], [10], [15], [31], [32], power consumption modeling [17], and simulation speed optimization [24], [34]. Notably, research on inter-SM communication includes cooperative caching network [11], which reduces L2 bandwidth demand, and shared L1 cache simulation model [14], which minimizes data replication but increases latency. This work extends the state of the art by integrating a TBC performance model into GPGPU-Sim, enabling broader experimentation with this feature.

Thread Block Clusters and Locality Optimization: TBCs introduce a new level of hierarchy in CUDA programming, allowing threads within a cluster to cooperatively access distributed shared memory, improving locality and reducing global memory latency [9], [26]. While previous studies have addressed inter-SM communication latency and locality optimization [11], [13], [18], the performance evaluation of TBCs is a novel area of research. Dublish et al. [11] and Ibrahim et al. [13] explored cooperative and shared L1 cache techniques, respectively, while Li et al. [19] focused on locality-aware block scheduling. Unlike these transparent L1 cache optimizations, TBCs require explicit application-level management. This explicit management introduces new challenges and opportunities for optimizing kernel execution and resource utilization.

VII. Conclusion

We present *ClusterSim*, a simulation model enabling detailed study and experimentation of NVIDIA Hopper's TBC feature. To support TBC, we integrated essential PTX instructions, implemented a custom thread-block scheduler, and designed an SM-to-SM network model that accurately captures cluster-level communication. Extensive microbenchmarking reveals that the SM-to-SM network exhibits a crossbar-like topology, characterized by uniform latencies, fair bandwidth sharing during broadcast and pairwise accesses, and multiple concurrent, non-interfering data paths, properties that rule out mesh or ring-based designs. Our model achieves a high simulation fidelity, with a mean absolute relative error of 4.7% compared to real H100 hardware, consistently outperforming simpler latency models across cluster sizes. Furthermore, we demonstrated the utility of our simulation model by evaluating improvements to the SM-to-SM network, such as reduced latency and increased bandwidth, revealing significant potential performance gains. Our work lowers the barrier for TBC research by providing the community with a valuable simulation model to explore and optimize future GPU architectures.

References

[1] "NVIDIA H100 Tensor Core GPU Architecture Overview," https://resources.nvidia.com/en-us-tensor-core. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core

[2] "PTX ISA 8.5," https://docs.nvidia.com/cuda/parallel-thread-execution/. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/

[3] https://github.com/gpgpu-sim/ispass2009-benchmarks, 2009, accessed: 2018-04-11.

[4] V. S. Adve and M. K. Vernon, "Performance analysis of mesh interconnection networks with deterministic routing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 3, pp. 225–246, 1994.

[5] A. Bakhoda, G. L. Yuan, W. W. L. 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*, Apr. 2009, pp. 163–174. [Online]. Available: https://ieeexplore.ieee.org/document/4919648

[6] Y. Bao, Y. Sun, Z. Feric, M. T. Shen, M. Weston, J. L. Abellán, T. Baruah, J. Kim, A. Joshi, and D. Kaeli, "NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 333–345. [Online]. Available: https://dl.acm.org/doi/10.1145/3559009.3569666

[7] Bhuyan and Agrawal, "Design and performance of generalized interconnection networks," *IEEE Transactions on computers*, vol. 100, no. 12, pp. 1081–1090, 1983.

[8] S. Bourduas and Z. Zilic, "Modeling and evaluation of ring-based interconnects for network-on-chip," *Journal of Systems Architecture*, vol. 57, no. 1, pp. 39–60, 2011.

[9] J. Choquette, "NVIDIA Hopper H100 GPU: Scaling Performance," *IEEE Micro*, 2023.

[10] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. E, "ATTILA: a cycle-level execution-driven simulator for modern GPU architectures," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2006, pp. 231–241. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1620807

[11] S. Dublish, V. Nagarajan, and N. Topham, "Cooperative Caching for GPUs," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 4, pp. 39:1–39:25, Dec. 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/3001589

[12] A. C. Elster and T. A. Haugdahl, "Nvidia Hopper GPU and Grace CPU Highlights," *Computing in Science & Engineering*, vol. 24, no. 2, pp. 95–100, Mar. 2022, conference Name: Computing in Science & Engineering.

[13] M. A. Ibrahim, O. Kayiran, Y. Eckert, G. H. Loh, and A. Jog, "Analyzing and Leveraging Shared L1 Caches in GPUs," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT, 2020.

[14] ——, "Analyzing and Leveraging Shared L1 Caches in GPUs," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, Sep. 2020, pp. 161–173. [Online]. Available: https://dl.acm.org/doi/10.1145/3410463.3414623

[15] 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)*. IEEE, 2020, pp. 473–486.

[16] J. Lee, S. Li, H. Kim, and S. Yalamanchili, "Design space exploration of on-chip ring interconnection for a cpu–gpu heterogeneous architecture," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1525–1538, 2013.

[17] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: enabling energy optimizations in GPGPUs," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 487–498, Jun. 2013. [Online]. Available: https://dl.acm.org/doi/10.1145/2508148.2485964

[18] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-Aware CTA Clustering for Modern GPUs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2017.

[19] ——, "Locality-Aware CTA Clustering for Modern GPUs," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 297–311. [Online]. Available: https://doi.org/10.1145/3037697.3037709

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

[21] T. Lühnen, T. Marschner, and S. Lal, "Benchmarking thread block cluster," in *28th Annual IEEE High Performance Extreme Computing Conference, HPEC 2024*, 2024.

[22] W. Luo, R. Fan, Z. Li, D. Du, H. Liu, Q. Wang, and X. Chu, "Dissecting the nvidia chitecture through microbenchmarking and multiple level analysis," 2025. [Online]. Available: https://arxiv.org/abs/2501.12084

[23] 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]. [Online]. Available: http://arxiv.org/abs/2402.13499

[24] G. Malhotra, S. Goel, and S. R. Sarangi, "GpuTejas: A parallel simulator for GPU architectures," in *2014 21st International Conference on High Performance Computing (HiPC)*, Dec. 2014, pp. 1–10, iSSN: 1094-7256. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7116897

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

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

[27] ——, "Cuda c++ programming guide," 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide

[28] ——, "NVIDIA H100 Tensor Core GPU Architecture," 2023. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core

[29] ——, "Parallel thread execution isa version 8.2," 2023. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[30] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[31] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A Heterogeneous CPU-GPU Simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, Jan. 2015, conference Name: IEEE Computer Architecture Letters. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6709764

[32] M. A. Raihan, N. Goli, and T. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs," Feb. 2019, arXiv:1811.08309 [cs]. [Online]. Available: http://arxiv.org/abs/1811.08309

[33] B. Shan and M. Araya-Polo, "Evaluation of Programming Models and Performance for Stencil Computation on Current GPU Architectures," Apr. 2024, arXiv:2404.04441 [cs]. [Online]. Available: http://arxiv.org/abs/2404.04441

[34] O. Villa, D. Lustig, Z. Yan, E. Bolotin, Y. Fu, N. Chatterjee, N. Jiang, and D. Nellans, "Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2021, pp. 868–880, iSSN: 2378-203X. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9407154