HyFiSS: A Hybrid Fidelity Stall-Aware Simulator for GPGPUs

Jianchao Yang[†], Mei Wen[†✉], Dong Chen[§], Zhaoyun Chen[†], Zeyu Xue[†], Yuhang Li[†], Junzhong Shen[†], Yang Shi[†]
[†]*Key Laboratory of Advanced Microprocessor Chips and Systems, College of Computer, National University of Defense Technology, Changsha, China*
{*yangjianchao16*, *meiwen*, *chenzhaoyun*, *xuezeyu18*, *liyuhang*, *shenjunzhong*, *shiyang14*}*@nudt.edu.cn*
[§]*Huawei Technologies Co., Ltd, Beijing, China*
jameschennerd@gmail.com

***Abstract*—The widespread adoption of GPUs has driven the development of GPU simulators, which, in turn, lead advancements in both GPU architectures and software optimization. Trace-driven cycle-accurate Cycle-accurate simulators, which provide detailed microarchitectural models and clock-level precision, come at the cost of extended simulation times and require high computational resources. Their scalability has become a bottleneck. A growing trend is the adoption of cycle-approximate simulators, which introduce mathematical modeling of partial hardware units and utilize sampling to accelerate simulation. However, this approach faces challenges regarding the accuracy of performance predictions.**

To address these limitations, we introduce HyFiSS, a hybrid fidelity stall-aware GPU simulator. HyFiSS features fine-grained stall events tracking and attribution by constructing a detailed execution pipeline model for various stall events on *Streaming Multiprocessors* (SMs). It accurately emulates the thread block scheduler behavior using real-time scheduling logs and utilizes sampling based on thread block sets to minimize the precision loss due to fine-grained sampling points on the microarchitectural state. We achieve a balance between reliability, speed, and the level of simulation detail, especially regarding bottlenecks. By evaluating a diverse set of benchmarks, HyFiSS achieves a mean absolute percentage error in predicting active cycles that is comparable to the state-of-the-art cycle-accurate simulator Accel-Sim. Moreover, HyFiSS achieves a substantial 12.8× speedup in the simulation efficiency compared to Accel-Sim. HyFiSS also requires at least 3.2× less disk storage than both Accel-Sim and another state-of-the-art cycle-approximate simulator PPT-GPU due to its efficient *SASS* (*Streaming Assembler*) traces compression. With precise, per-cycle stall events statistics, HyFiSS can provide accurate GPU performance metrics and stall cause reporting. This significantly simplifies performance analysis, bottleneck identification, and performance optimization tasks for researchers, making it easier to enhance GPU performance effectively.

***Index Terms*—GPGPU Simulation, Workload Characterization**

I. Introduction

Graphics Processing Units (GPUs) are extensively used in compute-hungry domains, such as high-performance computing and artificial intelligence. To optimize GPU hardware or software running on it [1]–[5], GPU simulators played a key role in understanding the execution details and identifying performance bottlenecks. Various industrial and academic GPU simulators [6]–[13] have been meticulously designed, encompassing the associated performance analysis models [14]–[24]. These simulation frameworks[1] typically allow users to describe GPU architectures with high-level parameters, and provide a performance model for the specified architecture to guide system optimizations.

Despite the advancements in GPU simulation frameworks, achieving a comprehensive balance between simulation fidelity and speed remains challenging for the existing open-source GPU simulation frameworks [9]. A cycle-accurate simulation demands to emulate the behavior of all components in the GPU architecture and their interactions in each cycle. It usually incurs significant overheads, which usually grows exponentially with the size or complexity of the simulated workload. Such overheads could lead to substantial time costs for iterative simulations in system optimizations [21], [25], [26]. Meanwhile, researchers often need rapid, iterative performance feedback. The slow speed of cycle-accurate simulators can hardly meet these needs [22], [27], [28].

A growing trend is the adoption of cycle-approximate simulators, which simulate compute instructions and memory instructions separately [8], [29]–[31]. They utilize mathematical modeling and multi-level sampling strategies to improve simulation speed by orders of magnitude. Unlike comprehensive, high-fidelity cycle-accurate simulations, cycle-approximate approaches abstract local details through mathematical modeling. These models include the roofline model [14], [15], parallelism based on latency hiding [16], [17], interval-based analysis [29]–[31], and compilation-based analysis of data or control flow graphs [18], [21]–[23]. However, these models often lack a robust methodology for accurately representing complex hardware states. Consequently, they may overlook established prior knowledge such as well-documented hardware features or parameters familiar to hardware architects and software optimization researchers. In particular, they do not adequately account for the impact of hardware stalls on

✉ Corresponding author.

[1]In this paper, we refer to GPU simulators and performance analysis models collectively as simulation frameworks.

2379-315/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00022

performance, which has an important impact on the application execution on GPUs. As a result, cycle-approximate simulators may encounter limitations in achieving precise performance evaluation and detailed bottleneck analysis [21].

To overcome these drawbacks, we introduce HyFiSS, a cycle-approximate, hybrid fidelity stall-aware simulator. Figure 1 illustrates our design principles, which begin with identifying typical stall events through abstract analysis from real hardware ❶. We then choose different fidelities of hardware pipelines to construct the simulator ❷. Via simulation and stall event attribution of sampled real application traces, HyFiSS provides stall event stacks ❸, which are similar to CPI stacks in CPUs that indicate the distribution of execution time for instructions [32]–[35]. The abstracted stall events are further refined based on this feedback, thereby influencing the design of HyFiSS ❹. Once the stall events are established and stabilized, HyFiSS can identify hardware bottlenecks through attribution analysis, enabling optimization of the hardware design ❺. Figure 2 quantitatively compares the prediction performance of two state-of-the-art simulators—Accel-Sim [6], [7] (a cycle-accurate simulator, denoted as *ASIM*), and PPT-GPU [8] (a cycle-approximate simulator, denoted as *PPT*)[2]—and HyFiSS across a suite of 35 applications consisting of 1,784 kernels. These are compared against performance metrics obtained from a real NVIDIA QUADRO GV100 GPU (hereafter referred to as the GV100 GPU, denoted as *NCU*) using *Nsight Compute* [36].

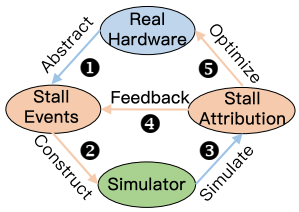


Fig. 1: Conceptual Overview of Our Design Principles.

As shown in Figure 2a Accel-Sim's simulation of a program is up to ten million times slower than real execution on GPU, resulting in significant cumulative delays during the iterative simulation to evaluate numerous design strategies [37]. Conversely, PPT-GPU offers markedly faster simulations—up to three orders of magnitude faster than Accel-Sim—attributable to a reuse distance analysis model and a parallelized compute model with lower fidelity. However, PPT-GPU suffers from larger prediction errors in performance metrics. Figure 2b-Figure 2c exhibit that PPT-GPU's mean absolute percentage errors (MAPEs) in active cycles, instructions per cycle (hereafter referred to as IPC) and occupancy reach 255.8%, 72.0%, and 27.39%, respectively. In contrast, the detailed microarchitectural modeling of Accel-Sim results in much lower errors of only 21.9%, 33.3%, and 3.62%.

Whereas, HyFiSS makes up for the gap between them in terms of simulation speed and prediction accuracy. HyFiSS demonstrates the MAPE of only 39.9% in predicting active cycles, outperforming PPT-GPU. It achieves an average 12.8× speedup over Accel-Sim. Further experimental results show that HyFiSS offers an average 3.2× reduction in trace disk storage compared to both Accel-Sim and PPT-GPU. Furthermore, HyFiSS demonstrates remarkable accuracy in predicting critical cache performance metrics. The mean absolute error (MAE) for the L1 and L2 cache hit rates, and occupancy is only 5.68%, 13.53%, and 10.26%, respectively. Furthermore, HyFiSS offers comprehensive bottleneck analysis for programs running on NVIDIA GPUs, which gives hardware designers and software programmers valuable feedback to optimize the hardware or program due to its stall-aware feature.

The contributions of this paper are listed as follows:

- We analyzed stall events closely linked to hardware constraints and focuses on the primary stall types impacting pipeline execution. This inspired us to develop a hybrid fidelity simulation. It incorporates various levels of fidelity—from highly detailed to highly abstracted—to forge an effective and simulation-efficient abstraction of the entire GPU hardware.
- Our approach enables hybrid-fidelity stall events tracking and accurately attributes a stall cycle to its root cause. We have introduced an attribution algorithm, which builds the foundation for detailed analysis of performance bottlenecks and effective workload characterization.
- We propose a Streaming Multiprocessors sampling algorithm based on *Cooperative Thread Array-Sets*. This algorithm relies on *SASS* (*Streaming Assembler*) traces to rectify the inaccuracies introduced by fine-grained sampling, such as simulating only one warp or thread block, which often diminishes the resource contention in execution pipelines. Our sampling not only improves the simulation accuracy but also significantly enhances the efficiency.
- We have open-sourced our stall-aware GPU simulator [38], HyFiSS. It is a clock-driven simulator for pipeline execution with a parallelized cache hierarchy model based on reuse distance analysis. Besides the simulator, we also provide a set of tools for *SASS* traces extraction, sampling, statistical analysis, etc. The encouraging experimental results suggest the effectiveness of our approach.

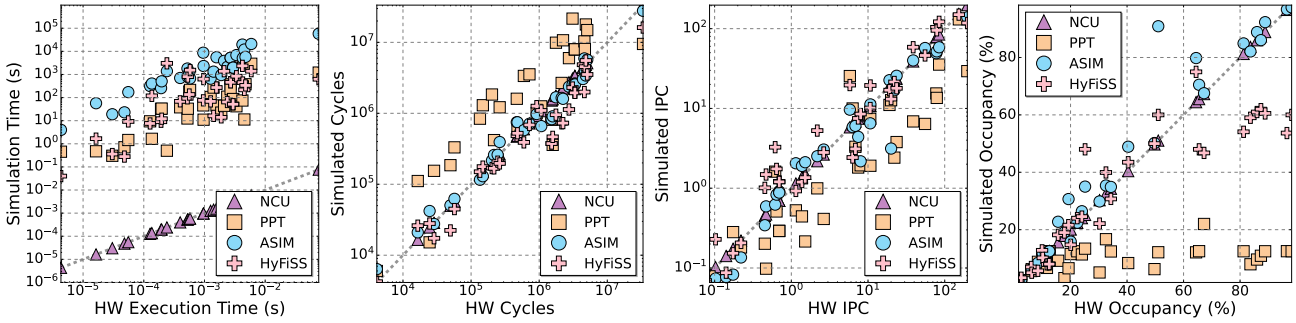
II. Background & Motivation

This section provides the necessary theoretical background to understand our design principles and motivations.

A. GPU Hardware Architecture

We briefly introduce the NVIDIA GPU architecture organization using the example of the GV100 GPU [6], [7], [39]–[41], as illustrated in Figure 3. NVIDIA GPUs consist of an array of *Streaming Multiprocessors* (SMs) that execute in parallel. Each SM contains a variety of scalar and vector ALU units, as well as specialized cores like Tensor Cores [1]. Each SM includes a private L1 cache, which also serves as shared memory. All SMs access a unified L2 cache through a high-bandwidth interconnect network.

[2]We select PPT-GPU as the baseline because it is—to our knowledge—the only open-source and executable cycle-approximate GPU simulator.



(a) Simulation Time. (b) Active Cycles (c) Instructions Per Cycle. (d) Achieved Occupancy.

Fig. 2: Prediction Performance Comparison of Accel-Sim [7] and PPT-GPU [8].

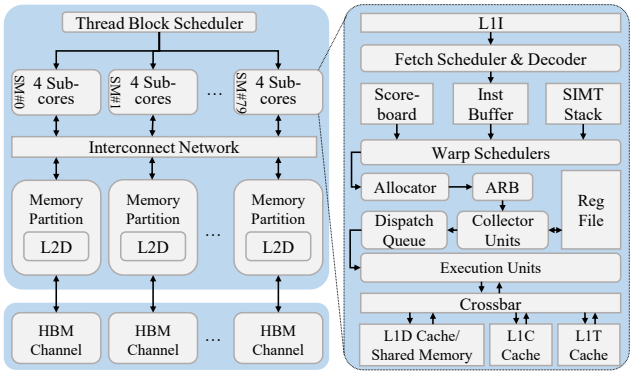


Fig. 3: Architecture Organization of the GV100 GPU.

B. Parallel Execution Pattern of CUDA Programs

To effectively simulate the programs on GPUs, it is crucial to understand both the complexity of GPU hardware and the multilayered parallel execution patterns employed by programs. In the CUDA programming model, users define the grids and thread blocks (aka *Cooperative Thread Arrays*, hereafter referred to as CTAs) organization and the distribution of computational tasks across available threads [42]. The device code written by users is compiled into kernels, which execute concurrently on the SMs following the *Single Instruction, Multiple Threads* (SIMT) paradigm. Threads within each CTA are grouped into warps, which constitute the basic scheduling unit. At its core, the simulators simulate the complex multi-level parallelism of CTAs on multi-SMs, capturing both the simultaneous execution across SMs and the concurrent warp processing within pipelines, with warps as the fundamental granularity of simulation.

The execution of threads is managed by a multi-tiered hardware scheduler. The scheduling of instructions execution on GPUs is conducted at two levels: the warp schedulers dispatch warps to time-multiplexed execution units [43]–[48], whereas the CTA scheduler allocates CTAs to execute on SMs [49]–[51]. Typical simulation frameworks for GPGPUs adopt a Round-Robin (hereafter referred to as *RR*) strategy for CTA scheduling, which sequentially allocates consecutive CTAs to different SMs when there are sufficient resources available [6]–[8], [49], [50]. However, our observations of the CTA scheduler behavior on a real GV100 GPU reveal deviations from the *RR* strategy.

We build our tracing tool on top of NVIDIA's *NVBit* dynamic binary instrumentation tool [52] to collect the runtime CTA scheduler behavior on a real GV100 GPU. Figure 4 illustrates the normalized distribution of CTAs and warp instructions across different SMs for 9 applications from widely-used benchmark suites including Rodinia [53] and PolyBench [54]. The *SM IDs* have been sorted based on the distribution of CTAs. A general observation suggests that the distributions of CTAs and warp instructions for all the applications exhibit varying degrees of imbalance. Figure 5 presents a time-view of CTA scheduling behavior for the *b+tree* application. It displays the IDs of the CTAs allocated to the first four SMs, based on information traced from the CTA scheduler. This reveals a non-sequential and non-uniform pattern of CTA allocation during runtime. The actual hardware likely maintains a schedulable queue influenced by the on-the-fly information, resulting in a non-uniform CTA distribution across SMs.

C. Stalls in Execution

Stall refers to the phenomenon where the GPU pipeline experiences a pause during execution and is not able to continue executing. GPUs aim at hiding pipeline latency with *Thread-Level Parallelism* (TLP) and *Instruction-Level Parallelism* (ILP), while stalls create pipeline bubbles, detrimentally affecting the execution efficiency [55]. Stalls not only directly lead to pipeline resource wastage and performance loss but also amplify performance bottlenecks through their cascading effects, significantly reducing the performance of the entire system. Morpheus [56] reported the performance degradation in K-means clustering applications due to stalls from the reduced L2 cache hit rates. These cache misses cause a cascading effect, stalling the pipeline and preventing the issue of subsequent instructions, leaving most SMs to be idle. Consequently, the performance with 68 SMs barely exceeds that with 10 SMs and is 50% less than with 20

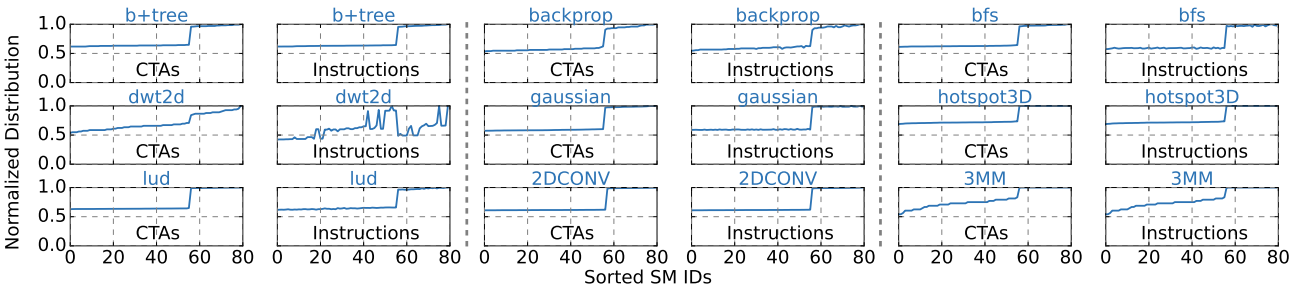


Fig. 4: Normalized Distribution of CTAs and Warp Instructions on SMs.

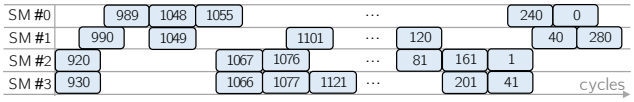


Fig. 5: Time-View of CTA Scheduling Across SMs for *b+tree*.

SMs, highlighting the amplified systemic effects of stalls. Thus, stalls are the key issue in GPU's performance bottleneck analysis [14]–[17], [21], [22], [29]–[31], [46], [57]–[60].

Generally, the stalls are categorized into seven types:

- **Memory Structural Stall** (*MemStruct*) is caused by the resource contention within the memory subsystem.
- **Memory Data Stall** (*MemData*) arises from latency in data access and data dependencies among operands for memory instructions.
- **Compute Structural Stall** (*ComStruct*) arises from resources contention within GPU compute units.
- **Compute Data Stall** (*ComData*) results from data dependencies among operands for compute instructions.
- **Synchronization Stall** (*Sync*) arises from thread synchronization and inter-thread communication.
- **Control Stall** (*Ctrl*) occurs when branch divergence invalidates the instructions queued in the instruction buffer.
- **Idle Stall** (*Idle*) happens in the absence of active warps ready for instruction issue.

Motivation. The hardware complexity of GPUs is not only high but also continually increasing as their architectures evolve to support advancements in performance and features. For example, the GV100 GPU contains 80 SMs and 5,120 CUDA cores, while later generations of GPUs, such as Tesla A100 and Hopper H100 GPUs have greater numbers of SMs and CUDA cores. In addition, GPU hardware architecture involves a hierarchical mapping and scheduling process, along with elaborate interactions between various execution units. Numerous concurrently executing threads, each maintaining independent execution contexts, require synchronization and communication among them. These intertwining factors pose significant challenges to simulation accuracy and speed. We believe that the ideal simulator should achieve a comprehensive balance between reliability, speed, and bottleneck analysis capability; this is the goal of our work. Designing a GPU simulator from the perspective of stalls might be an effective approach not only for revealing performance bottlenecks but also for enabling potential optimization opportunities in the simulation of GPU systems, improving both accuracy and speed. The following section III and section IV will discuss our design principles and implementation.

III. Design Principles and Key Algorithms

This section outlines the design principles of HyFiSS, a simulation employing hybrid fidelity built on multiple levels of abstraction. Our design principles relies on two key algorithms. One is stall attribution, which tracks stall events and attributes them to a stall cycle. The other is SM sampling based on *Cooperative Thread Array-Sets*, which makes it sufficient to simulate one SM instead of all SMs.

A. Simulation with Hybrid Fidelity

Simulation frameworks inevitably involve accepting a certain level of inaccuracy, and fidelity grading is essential for faster simulation [9]. We thoroughly analyze the root causes of all stall events for GPU hardware. Based on this analysis, we present a hybrid fidelity simulation design that integrates various levels of hardware fidelity, thus enabling an effective abstraction of the entire hardware system.

Figure 6 illustrates our simulator design, featuring a single SM pipeline with key multi-SM shared components like L2 cache and HBM. The GPU execution pipeline contains a variety of units, each serving a specific function. The detailed explanations of these units are as follows:

- *SP* stands for Single-Precision Floating Point Units;
- *DP* denotes Double-Precision Units;
- *SFU* is the Special Function Units;
- *INT* refers to the Integer-Precision Units;
- *TC* stands for Tensor Cores;
- *LDST* represents the Memory Pipeline;
- *CU* is identified as the Collector Units;
- *ARB* refers to the arbitrator selecting non-conflicting accesses for the register files.

For each unit, we use three fidelity levels: high, medium, and low fidelities to simulate the GPU execution pipeline. High-fidelity modules simulate every hardware state for maximum precision. Medium fidelity, in contrast, relies on latency models that approximate hardware behavior, sacrificing some accuracy for efficiency. Low-fidelity simulation uses empirical

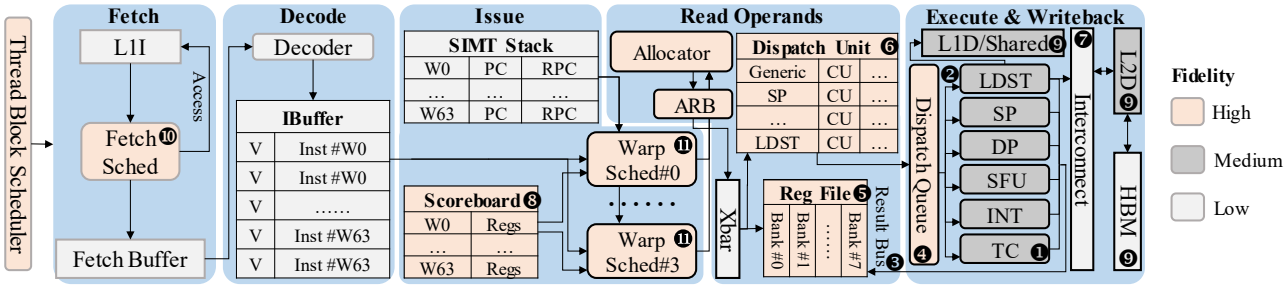


Fig. 6: The Illustration of GPU Execution Pipeline with Hybrid Fidelity.

latency models, providing a rough estimation and occasionally omitting certain hardware states for simplicity. Table I lists the key stall events, such as data dependency, resource contention, synchronization of warp instructions, etc., all of which are explicitly associated with the hardware units. As shown in Figure 6 and Table I, most hardware units related to stall events are designed to be at least medium fidelity.

Key components directly related to stalls, such as the fetch scheduler, scoreboard, warp schedulers, operand collectors, and register files, are simulated with high fidelity, as depicted in Figure 6. Execution units like *SP Unit*, *INT Unit*, and *DP Unit* are modeled with medium fidelity, encapsulating their execution with latency models that simulate instruction delays. For the memory hierarchy, including L1 and L2 caches, we employ medium-fidelity simulations based on the established *Reuse Distance* model. Reuse distance, reflecting the lifespan of cache blocks, provides a quantitative estimation of the cache hits rates in a sequence of memory references [8], [47], [61]–[68]. To ensure a realistic simulation, we use full *Streaming Multiprocessor* (full-SM) memory access sequences to estimate the shared L2 cache's hit rates, with a detailed explanation provided in subsection IV-E.

We simulate the interconnect network and HBM in low fidelity due to their inherent complexity and the relative infrequency of interconnect network congestion as a stall event [9]. Accurate simulation of the interconnect network can typically be achieved using latency and bandwidth models. For HBM accessed by global memory requests, we adopt an estimated *Average Memory Access Time* (*AMAT*) [69], [70], which will be detailed in subsection IV-E.

B. Stall Events Tracking and Attribution

Our simulator provides a detailed tracking of stall events that occur in the GPU execution pipeline. To systematically investigate the impact of stalls on GPU performance, it is essential to establish the connections between stalls and their underlying causes. Since different stall events can occur simultaneously and may belong to various stall types, identifying the primary root stall type can be challenging.

For instance, bank conflicts in the write-back stage in an earlier cycle can cause execution units to remain occupied as they need to retry the write-back in subsequent cycles, thereby delaying subsequent instruction issues. The direct factor for this stall was the execution pipeline saturation; however, the underlying indirect root cause can be traced back to bank conflicts from the preceding cycle. Similarly, high-latency memory accesses might not stall the instruction issue immediately but can prevent it in a later cycle. These indirect stall events, therefore, are a result of accumulated delays from various saturation points or contention in the system, ending in a stall cycle. We hereby formally define the following terms:

- *stall cycle*, a cycle in which at least one warp scheduler is unable to issue an instruction from its issue queue. In an ideal case, each warp scheduler would reach its peak issue rate without any stalls.
- *root stall type*, the type of the initial or the primary event responsible for triggering subsequent stalls in the pipeline.
- *direct stall events*, the stall events that immediately prevent instructions from being issued.
- *indirect cascade-triggering stall events*, the indirect stall events that trigger additional stall events in later cycles, induced by interactions at different pipeline units.

The goal of our stall attribution algorithm is to accurately pinpoint the *root stall type* for each *stall cycle* by analyzing both the two stall events (as previously defined) across the *stall cycle* and several of its prior cycles.

The identification of *direct stall events* is prioritized as follows: **(1)** The instruction buffer (*IBuffer*) has no active warps available for the current warp scheduler ⑩, directly preventing instructions issue. **(2)** Synchronization ⑪, data dependencies ⑧, or unsatisfied execution unit issuing mutual exclusion ② may prevent the issuing of queued instructions. The stall event ② means that it is prohibited to issue two consecutive instructions to the same execution unit to weaken contention and improve overall instruction throughput. **(3)** A shortage of free operand collectors (*CUs*) ⑥ likely prevents the instruction from obtaining necessary operands, blocking its progression in the pipeline.

Stall events triggered by pipeline stages after *CUs* in an earlier cycle will exacerbate the release blockage of the *CUs* at the current *stall cycle*. Considering the time cost of the algorithm, we designate a set of four consecutive cycles $\langle$*cycle#0*, *cycle#1*, *cycle#2*, *stall cycle*$\rangle$ for backtracking to identify *indirect cascade-triggering stall events*. The prioritization sequence is as follows: **(1)** Bank conflicts ⑤ at *cycle #2*

TABLE I: Stall Events, Their Occurrence Units, and Types.

Stall Events and Occurrence Units	Stall Type
Execution Unit Pipeline Saturation ❶* Execution Unit Issuing Mutual Exclusion ❷ Result Bus Saturation ❸ Dispatch Queue Saturation ❹ Bank Conflict ❺ No Free Operands Collector Units ❻	MemStruct ComStruct
Interconnect Network Congestion ❼	MemStruct
Data Dependence in Scoreboard ❽	MemData/ComData
Cache/Global Memory Access ❾	MemData
No Active Warps ❿	Idle
Barrier synchronization ⓫	Sync

* Numerical labels mark event numbers and occurrence units in Figure 6.

result in the *CUs* remaining occupied during the current *stall cycle*. **(2)** Execution unit pipeline saturation ❶ or the dispatch queue saturation ❹ at *cycle #2*. **(3)** High-latency memory accesses ❾, result bus saturation during the writeback ❸, and interconnect network congestion ❼ at *cycle #1*. Given that most optimized applications efficiently mitigate congestion in the interconnect network [9], and the result bus saturation is a direct consequence of bank conflicts at *cycle #0* (which has been factored into our analysis during the stall attribution of *cycle #1*). Therefore, we identify high-latency memory accesses, typically occurring in the *LDST Unit*, as the primary contributors to the current *stall cycle*.

algorithm 1 prioritizes *direct stall events* as the probable root cause of a *stall cycle*. However, if there are underlying *indirect cascade-triggering stall events* potentially driving these direct events, it will be investigated further to determine the root cause. Additionally, when multiple *direct stall events* or *indirect cascade-triggering stall events* occur, we determine the root cause of the *stall cycle* based on the extent to which the events affect the instructions issue.

It is crucial to note that the stall events listed in Table I were not abstracted in a single step during the construction of the simulator. Initially, only a subset of the events in Table I was constructed at the early design stage. Through the process of stall events tracking and attribution, we observed that some *stall cycles* could not be attributed to one of the initially defined stall events. This finding prompted an expansion of the stall event abstractions based on the current observed *stall cycles*. Therefore, Table I presents the finalized set of stall events after the stabilization of the iterative *Feedback* process depicted in Figure 1. This iterative approach can be extended to the construction of simulators for other hardware platforms, and we will demonstrate the comprehensiveness of the finalized set of stall events in subsection VI-A.

C. SM Sampling Based on CTA-Sets

A widely adopted technique to accelerate architectural simulation is to sample the *phases* of a workload that exhibit repeatable patterns (e.g., IPC) across intervals [27], [37], [71]. We seek to determine which SM has the longest predicted execution time by analyzing the non-stalled execution of its *CTA-Set*—the collection of all CTAs belonging to that SM.

Algorithm 1: Stall Attribution for Warp Schedulers.

Input: Warp scheduler *$sched*, a set of consecutive cycles ⟨*cycle#1*, *cycle#2*, *stall cycle*⟩.
Output: *Root Stall Type* of the current *Stall Cycle*.

```
// Lines 2-13 represent stall events at stall cycle.
if $sched issue an instruction then
    return No Stall;
else if No Active Warps ❿ then
    return Idle Stall;
else if meet conditions ❷❽⓫ then
    if Barrier Synchronization ⓫ then
        return Sync Stall;
    else if Data Dependence in Scoreboard ❽ then
        return MemData/ComData Stall;
    else if Issuing Mutual Exclusion ❷ then
        return MemStruct/ComStruct Stall;
else if No Free Operands Collector Units ❻ then
    // Lines 15-18 represent stall events at cycle#2.
    if Bank Conflict ❺ then
        return MemStruct/ComStruct Stall;
    else if Execution Unit Pipeline Saturation ❶ or
            Dispatch Queue Saturation ❹ then
        // Lines 20-24 represent stall events at cycle#1.
        if Cache/Global Memory Access ❾ then
            return MemData Stall;
        else if Interconnect Network Congestion❼ then
            return MemStruct Stall;
        else if Result Bus Saturation ❸ then
            return MemStruct/ComStruct Stall;
// Attribution Failure: Stall events have not been abstracted.
return Other Stall;
```

This significantly reduces the time cost by eliminating the necessity of simulating every SM.

GPU workload sampling commonly employs different granularity, such as warp, basic block, or thread block [37], [71], [72]. However, these methods can not accurately evaluate performance stalls caused by resource contention within SMs, for the following reasons:

- Instructions executed before the sampled section alter the microarchitectural state, particularly the caches. This significantly impacts program performance. Simulators must perform a warming-up process before simulating the sampled section. This presents a major challenge [73].
- Sampling at a finer granularity, like basic block or warp, might miss the inherent resource contention due to the decreased number of concurrent instructions [27], [71], [72].
- Without structural information generated during compilation, *SASS* instructions extracted via binary instrumentation struggle to accurately rebuild Control-Flow-Graphs (CFGs) for basic block sampling.

To address these limitations, we propose SM sampling with the granularity of thread block sets (*CTA-Sets*). The offline sampling process runs concurrently with the actual GPU execution when extracting traces, ensuring that the time overhead for sampling is negligible. Unlike other methods [37] that use

clustering to assess the similarity among basic blocks, warps, or CTAs, we select the SM anticipated to have the maximal execution duration as the sampling point for simulation. These *CTA-Sets* with longer non-stall execution time, when execution units are fully utilized with a saturated issue rate, tend to consume more cycles in a resource-constrained SM pipeline. We predict the maximal anticipated execution duration of the *CTA-Set* on each SM by their non-stall execution time.

Let the set of execution units categories be U_{eu}, with the i-th category of execution unit eu^i (such as an *SP unit*) having a count of N^i_{eu} and an individual instruction execution latency of Lat^i_{eu}. Furthermore, let the number of instructions extracted from an SM be NI^i_{sm}, if these instructions can be dispatched to eu^i. Then, disregarding the issue rate of instructions, the execution time of eu^i can be calculated as:

$$C^i_{NonStall} = \frac{NI^i_{sm}}{N^i_{eu}} + (Lat^i_{eu} - 1). \tag{1}$$

And the total execution time $C_{NonStall}$ for all instructions can be calculated as $\max_{i \in U_{eu}} C^i_{NonStall}$, disregarding the instruction issue rate. Assume that *IRate* denotes the saturated issue rate of all warp schedulers within an SM, representing the maximum number of instructions that can be issued per cycle by a single SM. Next, we consider the total execution time $C^{FullIR}_{NonStall}$ with warp schedulers operating at full issue rate. Therefore, $C^{FullIR}_{NonStall}$ can be calculated as:

$$C^{FullIR}_{NonStall} = \max_{i \in U_{eu}} \frac{C^i_{NonStall} \times N^i_{eu}}{\min(N^i_{eu}, IRate)}. \tag{2}$$

We calculate $C^{FullIR}_{NonStall}$ for each *CTA-Sets* and select the SM with the maximum execution time as the sampling point to be fed into our execution pipeline model for simulation.

IV. HyFiSS Simulator Implementation

This section describes the implementation of HyFiSS simulator. HyFiSS utilizes MPI for process-based parallelism and employs a hybrid-fidelity execution pipeline model with simulation gating for balancing fully event-driven and cycle-driven simulation. It is highly parameterized, enabling users to model various generations of NVIDIA GPUs by simulating *SASS* traces. We set configurable parameters derived from publicly available documentation [39], [40], [42] and a micro-benchmark suite from Accel-Sim [74].

A. Overview of HyFiSS

Figure 7 shows the HyFiSS architecture, which consists of (1) A *Tracing Tool*, built on top of *NVBit* [52], which records application configurations, processes SM sampling based on CTA-Sets, compresses *SASS* instructions, and captures the real CTA scheduler behavior. (2) A suite of *Parsers* interprets application configurations, the traced real CTA scheduler behavior, and user-defined hardware parameters. (3) A fast *Unpacker* decodes warp instructions for the *Execution Pipeline* and *Reuse Distance* model. (4) A *Reuse Distance* model predicts L1 and L2 cache hit rates. (5) A hybrid fidelity and stall-aware *Execution Pipeline* model tracks the stall events

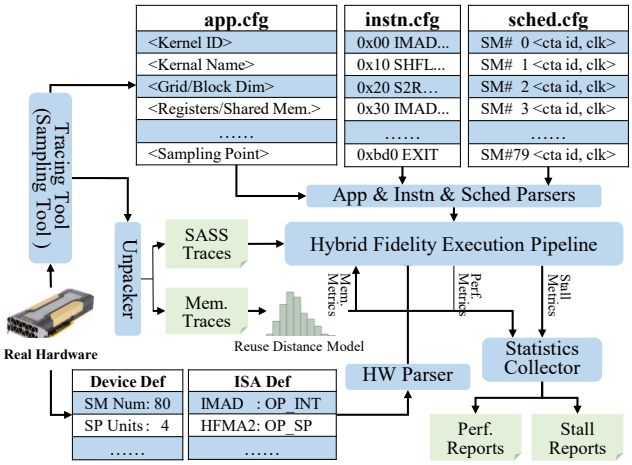


Fig. 7: Architecture of HyFiSS Simulator.

and performs the stall event attribution, and (6) A *Statistics Collector* provides a detailed simulation summary. We then provide more detailed descriptions of the key components, highlighting their roles and interplay within HyFiSS.

B. Tracing Tool

Our *Tracing Tool*, built on top of *NVBit* APIs [52], captures runtime information with minimal performance interference.

1) Application Configurations: The *Tracing Tool* generates an application configuration (*app.cfg*) that includes kernel details, as shown in Figure 7. This information guides the initialization of all warp instances with the grid and block dimensions and manages the concurrency of CTAs. Additionally, it reports the sampling points—the SM ID used during subsequent simulation—as outlined in subsection IV-C, exemplified by the last entry in *app.cfg*.

2) Traces Compression: Traces compression featured in the *Tracing Tool* reduces the size of traces extracted from real GPUs. Although the instructions executed by different warps within a kernel may vary in order, they share identical *program counters* (PCs) and instruction sequences. Compared to Accel-Sim [7] and PPT-GPU [8], HyFiSS significantly reduces traces storage requirements by retaining unique $\langle pc, mask \rangle$ pairs for instructions within each warp. It stores the instruction strings including opcodes, operands, and memory addresses in *instn.config*. This enables efficient restoration of instruction strings during unpacking by querying *instn.config*.

3) CTA Scheduling Information: HyFiSS accurately captures the dynamic CTA scheduler behavior on real GPUs with an integrated monitor in the *Tracing Tool*. This monitor logs each CTA's issue with SM IDs and timestamps into *sched.cfg*. HyFiSS then reproduces CTA scheduling and execution from *sched.cfg*, mirroring the real hardware behavior.

C. Sampling Tool

Our *Sampling Tool*, integrated within the *Tracing Tool*, collects the count of instructions dispatched to various execution units within each CTA. To avoid the computational

TABLE II: Benchmarks Evaluated for 35 Applications Across 7 Suites, Comprising 1,784 Kernels.

	Applications (#kernels)	Traces Size (ASIM, PPT, HyFiSS)	Simulation Time	Applications (#kernels)	Traces Size (ASIM, PPT, HyFiSS)	Simulation Time	Domain
[53]	*backprop* (2)	311M, 200M, 75M	3m, 2s, 7s	nw (255)	577M, 374M, 241M	11m, 2m, 3m	ML/Bioinformatics
	b+tree (2)	783M, 884M, 183M	7m, 2s, 27s	*dwt2d* (10)	299M, 246M, 77M	5m, 33s, 17s	Data Structure/DSP
	hotspot (1)	161M, 24M, 41M	57s, 1s, 3s	*hotspot3D* (100)	11G, 23G, 4.7G	6h, 2m, 36m	Physics Simulation
	gaussian (300)	32G, 8.5G, 2.1G	65m, 44s, 23m	lud (300)	27G, 11G, 4.6G	2h, 42s, 14m	Linear Algebra
	huffman (46)	1.5G, 283M, 351M	12m, 38s, 2m	nn (1)	2.0M, 3.3M, 1.1M	4s, 1s, 1s	Data Compress./Mining
	bfs (24)	2.1G, 1.4G, 1.5G	39m, 11s, 7m	*pathfinder* (5)	738M, 217M, 161M	5m, 9s, 11s	Graph Algorithms
	cfd (10)	679M, 9G, 220M	9m, 33s, 21s	lavaMD (1)	20G, 648M, 4.5G	3h, 11s, 50s	Physics Simulation
[54]	*2DConv* (1)	1.1G, 2.6G, 233M	23m, 1s, 30m	*3DConv* (99)	1.3G, 3.6G, 795M	13m, 33s, 15m	Image Processing
	3mm (3)	3.3G, 18G, 2.5G	92m, 46s, 4m	*gemm* (1)	1.3G, 6G, 898M	30m, 12s, 1m	Linear Algebra
	atax (2)	264M, 2G, 212M	33m, 26s, 47s	*gesummv* (1)	350M, 3G, 283M	35m, 2m, 2m	Linear Algebra
	bicg (2)	264M, 2G, 212M	32m, 1m, 48s	*gramschm* (3)	79M, 506M, 62M	16m, 22s, 18m	Linear Algebra
	mvt (2)	264M, 2G, 212M	32m, 1m, 48s				Linear Algebra
[75]	gemm_train (8)	11G, 6.1G, 2.1G	2h, 6m, 9m	gemm_inf (8)	11G, 6.2G, 2.1G	2h, 7m, 9m	CNN
	rnn_inf (133)	25G, 33G, 11G	2h, 6m, 7m	rnn_train (291)	40G, 40G, 14G	88m, 91m, 8m	RNN
	conv_inf (11)	1.5G, 1.3G, 448M	22m, 4m, 28s				CNN
[76]	AlexNet (22)	32G, 30G, 12G	6h, 48m, 16m	SqueezeNet (30)	29G, 44G, 12G	16h, 20m, 11m	CNN
	GRU (2)	676K, 2.4M, 428K	19s, 1s, 3s	*LSTM* (1)	296K, 1.1M, 208K	23s, 1s, 2s	RNN
[77]	GemmEx (1)	9.3G, 3.2G, 2.2G	3h, 45s, 1m				Linear Algebra
[78] [79]	LULESH (81)	857M, 611M, 273M	20m, 3m, 4m	PENNANT (25)	4.8G, 3.6G, 1.1G	11m, 4m, 19m	Mesh Hydrodynamics

complexity of clustering methods, we leverage user-defined hardware parameters, such as the number of execution units and instruction execution latency, to predict the execution time of an SM and select the candidate to simulate. The offline sampling strategy effectively distributes the sampling overhead across the runtime of each real GPU thread, making it virtually imperceptible.

D. Traces Unpacking & Parsing

A fast *Unpacker* is developed to convert the instructions from the compressed format to HyFiSS's execution format. It retrieves and parses the instructions from the compressed file (*kernel.sass*) based on the unique identifier $\langle kernel\_id, pc \rangle$ defined in *instn.config*, where additional metadata such as instruction types, operand types, and operands are also recorded. Then, the *Trace-Parser* categorizes instructions and directs them using $\langle kernel\_id, pc, global\_warp\_id \rangle$ to the simulator for subsequent simulation.

E. Reuse Distance Model

We utilize multiprocessing to simulate the hit rates of L1 and L2 caches for global memory access across each SM, leveraging memory instructions. The calculation of the *Average Memory Access Time* (*AMAT*) of HyFiSS is similar to that of other simulators [8]. To accurately simulate the queuing overhead within the interconnect network, we regulate the instruction throughput rate of the latency model for *LDST Unit*. This ensures that it does not exceed the maximum bandwidth capacity of the interconnect network. In addition, the latency of global memory accesses reflects the queuing overhead for off-chip memory.

V. Evaluation

A. Experimental Setup

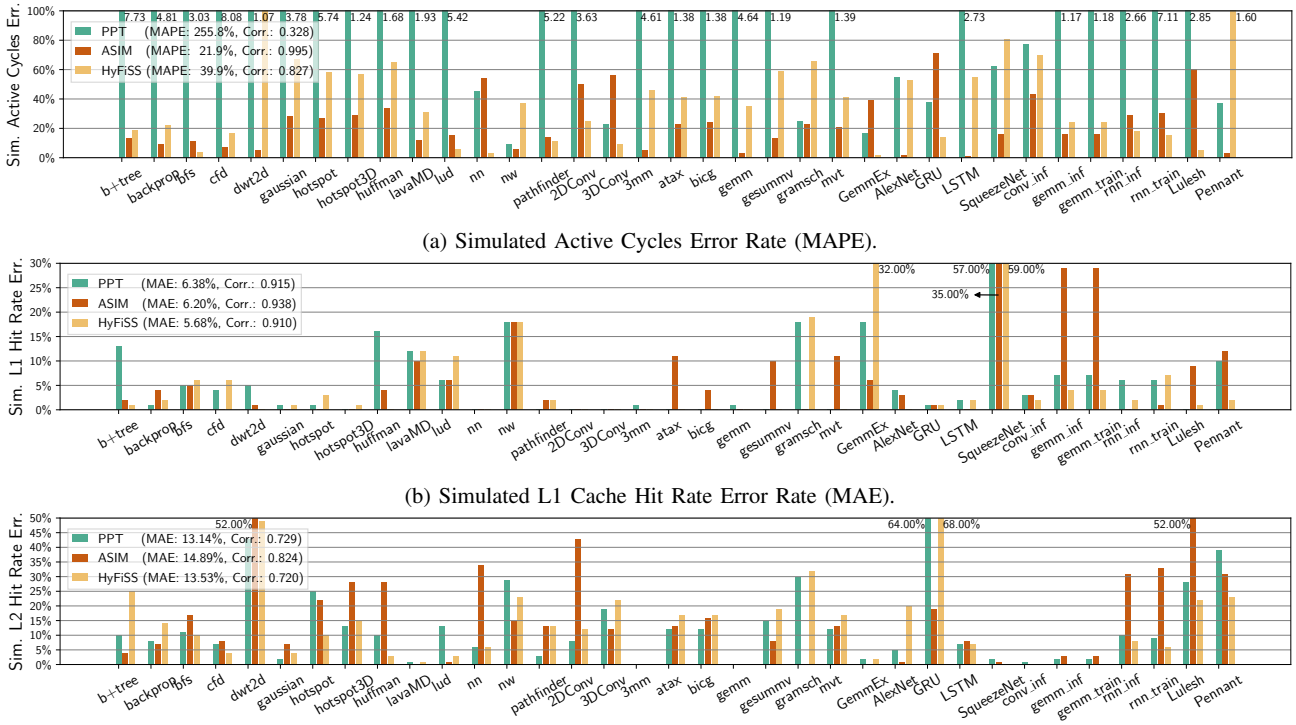
Our experimental system is equipped with two 32-core Intel® Xeon® processors with hyper-threading capability.

TABLE III: NVIDIA QUADRO GV100 GPU Configurations.

Parameter	Value
Number of SMs	80
Warp Schedulers per SM	4 Two-Level [80], [81] Round-Robin Schedulers
Max Warps, Threads, CTAs per SM	64 / 2048 / 32
Core Clock Frequency	1447 MHz
Shared Memory Size per SM	Configurable up to 96 KB
L1 Data Cache	32 KB, 4 sets, 32B line, 4 Banks, LRU
L2 Data Cache	6 MB, 24 Sets, 64B line, 64 Banks, LRU
Registers per SM	65536, 8 Banks

We use the hardware configuration of the widely-modeled NVIDIA QUADRO GV100 GPU [39]. Its simulation parameters are listed in Table III. We compare HyFiSS to the state-of-the-art cycle-accurate simulator Accel-Sim [7] and cycle-approximate simulator PPT-GPU [8], all simulating *SASS* instructions. Both Accel-Sim and PPT-GPU use *NVBit* APIs [52] to extract *SASS* instructions, a feature that HyFiSS also possesses while providing further capabilities. As mentioned earlier, HyFiSS supports CTA scheduler behavior extraction, traces compression, and sampling based on *CTA-Sets*. Performance metrics measured on a real GV100 GPU using NVIDIA's *Nsight Compute* [36] serve as the baseline.

Table II lists the benchmark applications from various domains used for validation. We run 35 applications with a total of 1,784 kernels. These applications are from the *cuBLAS* library [77], the heterogeneous computing benchmark suite PolyBench [54], Rodinia [53], the fluid dynamics benchmark LULESH [78], [82], the deep learning basic operators benchmark suite DeepBench [75], the DNN benchmark suite Tango [76] and the unstructured mesh mini-app benchmark PENNANT [79]. The convolutional layer training program from DeepBench [75] and the CifarNet model from Tango [76]



(a) Simulated Active Cycles Error Rate (MAPE).

(b) Simulated L1 Cache Hit Rate Error Rate (MAE).

(c) Simulated L2 Cache Hit Rate Error Rate (MAE).

Fig. 8: Prediction Performance Comparison of Active Cycles and Cache Hit Rates.

are omitted because their traces are too large, making simulation time unacceptable for Accel-Sim. All workloads are compiled using CUDA 11.0 with the compute capability *sm70* for the Volta architecture.

B. Prediction Accuracy

We use the mean absolute error (MAE) to evaluate the prediction accuracy of percentage metrics, and the mean absolute percentage error (MAPE) for that of non-percentage metrics. We use Pearson's correlation coefficient (Corr.) to show the correlation between the predicted and measured metrics. Figure 8 and Table IV illustrates the prediction performance of Accel-Sim, PPT-GPU, and HyFiSS for active cycles and cache hit rate, compared to the real hardware results.

The error rates for active cycle predictions are detailed in Figure 8a. Note that we sum up the active cycles across all simulated kernels as the total active cycles for that application. Accel-Sim shows the best performance in predicting active cycles, with the lowest MAPE of 21.9% and a prediction correlation near 1. HyFiSS, benefiting from a detailed stall analysis-based compute model, achieves the MAPE of 39.9% and a prediction correlation of 0.827. This performance significantly surpasses that of PPT-GPU, which has a MAPE of 255.8% and a correlation coefficient of 0.328 (refer to Figure 2b for an application-specific comparison).

Predictions of L1 and L2 cache hit rates are compared in Figure 8b and Figure 8c respectively. For each application, the hit rate is calculated as the sum of hit requests divided by the total requests for all simulated kernels. HyFiSS achieves the lowest MAE of 5.68% in predicting the L1 cache hit rate, slightly outperforming Accel-Sim (6.20% MAE) and PPT-GPU (6.38% MAE). All simulators exhibit prediction correlations above 0.910 for L1 cache hit rate. For L2 cache hit rate, HyFiSS has an MAE of 13.53%, slightly higher than PPT-GPU's 13.14% but lower than Accel-Sim's 14.89%.

TABLE IV: Error Rate and Corr. Comparison of Metrics.

(MAPE, Corr.)	**Accel-Sim**	**PPT-GPU**	**HyFiSS**
Active Cycles	21.9%, 0.995	255.8%, 0.328	39.9%, 0.827
GMEM Requests	269.4%, 0.173	88.0%, 0.999	82.8%, 0.998
IPC	33.3%, 0.901	72.0%, 0.711	65.5%, 0.928
(MAE, Corr.)	**Accel-Sim**	**PPT-GPU**	**HyFiSS**
L1 Hit Rate	6.20%, 0.938	6.38%, 0.915	5.68%, 0.910
L2 Hit Rate	14.89%, 0.824	13.14%, 0.729	13.53%, 0.720
Occupancy	3.62%, 0.974	27.39%, 0.416	10.26%, 0.910

* GMEM Requests: Global Memory Requests.

Table IV also illustrates the prediction performance for other metrics besides active cycles, L1, and L2 hit rates. Global memory (GMEM) requests predictions are best anticipated by HyFiSS with a 82.8% MAPE and 0.998 correlation, outperforming PPT-GPU's MAPE and closely following its high correlation. Accel-SIM's MAPE is significantly higher at 269.4%, indicating HyFiSS's superiority in predicting global memory requests. For each application, we calculate the av-

erage achieved occupancy across all kernels as the program's achieved occupancy. HyFiSS's 10.26% MAE is better than PPT-GPU's and only second to Accel-Sim, which leads with a 3.62% MAE. HyFiSS also holds a strong second with a 0.910 correlation, trailing behind Accel-Sim's 0.974 but notably ahead of PPT-GPU's 0.416 (see Figure 2d for an application-specific comparison). The IPC prediction comparisons in Table IV reveal that Accel-Sim leads in accuracy and correlation, while HyFiSS performs well but with room for improvement (see Figure 2c). IPC is a comprehensive metric influenced by numerous factors, such as HyFiSS's inaccurate estimation of the total number of instructions in non-sampled *CTA-Sets*.

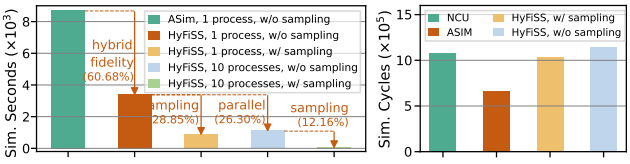
Overall, the prediction accuracy of HyFiSS closely aligns with the results achieved by Accel-Sim.

C. Simulation Time & Traces Disk Storage

TABLE V: Simulation Time & Traces Disk Storage.

	Simulation Time	Traces Disk Storage
Accel-Sim	2d 2h 21m 35s	269.7 GB
PPT-GPU	3h 24m 20s	263.4 GB
HyFiSS	3h 55m 12s	81.3 GB

Table V compares the simulation time and traces disk storage of Accel-Sim, PPT-GPU, and HyFiSS across all benchmarks. HyFiSS achieves a 12.8× speedup compared to Accel-Sim, due to its hybrid-fidelity strategy that reduces simulation overhead unrelated to stalls tracking. However, it is 15.1% slower than PPT-GPU, because PPT-GPU utilizes optimized parallel discrete event simulation (PDES) [83] and process-based parallelization [84]. In contrast, HyFiSS only employs a simple MPI framework for multiprocessing acceleration. Besides speedups, HyFiSS also exhibits a high trace compression ratio, consuming 3.3× and 3.2× less disk storage than Accel-Sim and PPT-GPU, respectively. It means that HyFiSS can support the simulation with larger trace sizes.



(a) Simulation Speedup. (b) Accuracy Loss.

Fig. 9: Ablation Study to Quantify HyFiSS Benefits.

To understand the benefits of hybrid fidelity, SM sampling based on *CTA-Sets*, and process-based parallelism employed by HyFiSS, Figure 9a compares the simulation times of Accel-Sim and HyFiSS using a *cuBLAS* [77] matrix multiplication program ($M, K, N = 2048$). Compared to Accel-Sim's serial simulation, hybrid fidelity alone reduces HyFiSS simulation time by 60.68%. Independently applying SM sampling and 10-process parallel simulation further reduce HyFiSS simulation time by 28.85% and 26.30%, respectively. By integrating all these techniques, HyFiSS achieves a 116× speedup over Accel-Sim with only minor accuracy loss as Figure 9b shows.

D. Irregular Applications and Alternative Microarchitectures

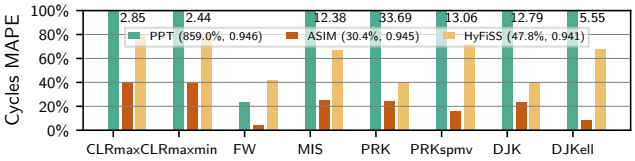


Fig. 10: Prediction Performance Comparison for Pannotia.

To demonstrate HyFiSS's applicability in predicting the accuracy of irregular applications, we compare its accuracy against Accel-Sim and PPT-GPU using the Pannotia benchmark suite [85]–[87] with the GV100 configuration. As shown in Figure 10, HyFiSS demonstrates competitive accuracy against Accel-Sim and significantly surpasses PPT-GPU.

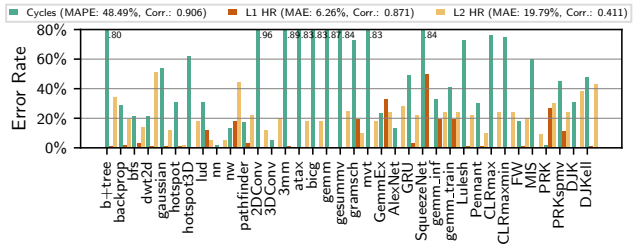


Fig. 11: Prediction Performance on the Ampere A100 GPU.

We then modeled the Ampere A100 GPU [88], which is equipped with 108 SMs and features 40 MB of L2 cache along with 192 KB/SM of L1 cache. The predicted accuracy of HyFiSS across all applications, including the Pannotia benchmark suite, is shown in Figure 11. HyFiSS demonstrates a MAPE of 48.49% in predicting simulated cycles, which is less accurate compared to the GV100 GPU simulation. Note that we did not alter the simulator design and only adjusted the hardware and *ISA* configuration based on publicly available documents [88], [89]. This might not fully capture the actual hardware details. Our future work involves calibrating HyFiSS with the A100 GPU. The increased number of SMs in the A100 GPU enhances its simulation efficiency, because the number of CTAs allocated to each sampled SM decreases.

E. SASS vs. PTX Simulation

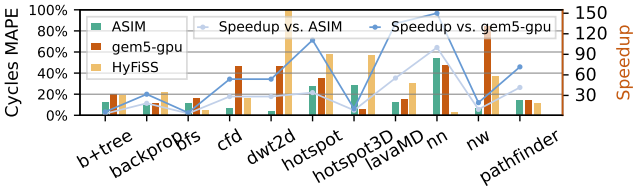


Fig. 12: Prediction Accuracy of *SASS* and *PTX* Simulation.

We compared the prediction performance of HyFiSS and Accel-Sim (*SASS* simulation) against gem5-gpu [10] (*PTX*

simulation) using the Rodinia benchmark suite [53], as illustrated in Figure 12. In most applications, *SASS* simulation achieves higher accuracy than *PTX* simulation, as the latter may oversimplify microarchitectural behaviors [90].

VI. Case Studies

This section details how HyFiSS reveals performance bottlenecks, which is useful for potential system optimization.

A. Stall Distribution

Figure 13a illustrates the distribution across 9 different stall types for various applications. It reveals that only a few applications (*backprop* and *gaussian*) have a *No Stall* percentage exceeding 30%, indicating that most programs experience significant stalls throughout their execution. Nearly all applications are primarily affected by four stall categories: *ComStruct*, *ComData*, *MemStruct*, and *MemData*. Based on their proportions, we classify applications as either *memory-bound* or *compute-bound*, as shown in Table II (*memory-bound* applications are underlined and italicized). Furthermore, *Sync*, *Ctrl* and *Idle* stalls have a minimal impact on all applications.

Our stall attribution methodology enables accounting for almost all *stall cycles* by attributing them to specific stall events, with only a negligible fraction classified as *Others*. This indicates that our defined stall events are comprehensive.

Figure 13b presents the distribution of *ComStruct* and *MemStruct* stall events. *GRU* and *LSTM* did not exhibit these two stall types, hence they are not presented. The data shows that *Execution Unit Pipeline Saturation*, *Dispatch Queue Saturation*, *Bank Conflict*, and *No Free Operand Collector Units* are the primary causes of *ComStruct* and *MemStruct* stalls. The SM experiences the highest percentage of stalls during the operands collection, writeback, and instructions dispatch stages. This insight suggests that architectural design should focus on optimizing these critical stages.

Figure 13c presents the distribution of *MemData* stall events. This reveals that most applications experience few stalls due to *Data Dependencies* and can effectively utilize the cache, resulting in fewer *Global Memory Access* stalls for applications with high cache hit rates. However, applications such as *cfd*, *dwt2d*, *nn*, *nw*, etc., face notable stalls due to their patterns of non-contiguous and random memory access. Optimizing cache utilization remains a key strategy for improving performance across many applications. For those encountering extensive *Global Memory Access* stalls, it is crucial to consider alternative approaches, such as redesigning access patterns, to improve memory bandwidth efficiency and reduce latency.

ComData stall events are all generated by the *Scoreboard*, we will not present a separate distribution for *ComData* stalls.

B. Execution Time and Instructions Distribution Analysis

Figure 14 illustrates the execution time and executed instructions distribution of execution units. This reveals a positive correlation between the execution time of an execution unit and the total number of instructions belonging to it. However, it is crucial to note that the workload of execution units varies among applications due to program characteristics such as data precision and computation patterns. For example, in the *nw* application, *SP Units* are rarely utilized. Therefore, monitoring the workload of execution units is essential for performance optimization. Execution units used by high-frequency instructions should be prioritized for performance tuning and resource scaling to prevent them from becoming bottlenecks and impacting the overall execution efficiency.

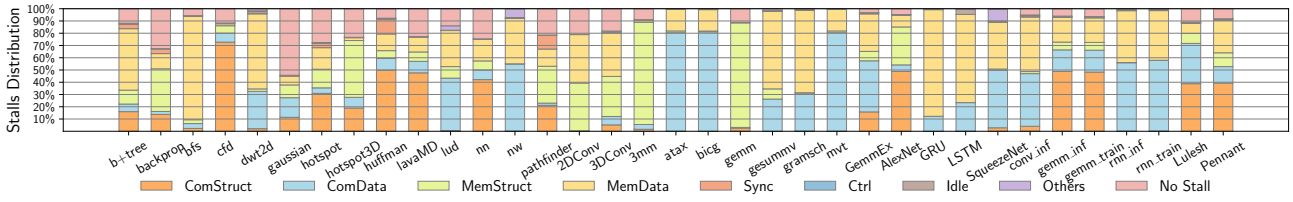
C. Accuracy of Stall Events Distribution

We select and compare three stall metrics from *Nsight* [36] that align with HyFiSS's counting logic: the average latency between two consecutive instructions waiting for scoreboard dependencies, CTA barriers, and available execution units. Figure 15 shows the results using Rodinia [53]. For the vast majority of applications, HyFiSS provides accurate metrics against *Nsight*. The discrepancies arise from inaccurate conflict management entries in HyFiSS to capture certain conflicts, such as scoreboard entries, barrier synchronization monitoring entries, and dispatch queue entries. During periods of rapid drop in conflicts (e.g., scoreboard dependencies of *bfs*), these entries become overly redundant and cause the smaller delays in issuing related instructions. This results in deviations from real hardware in terms of the average latency between two consecutive instructions and subsequently leads to lower average values. However, this situation has minimal impact on capturing most stall events in the pipeline and further, on execution cycle prediction.

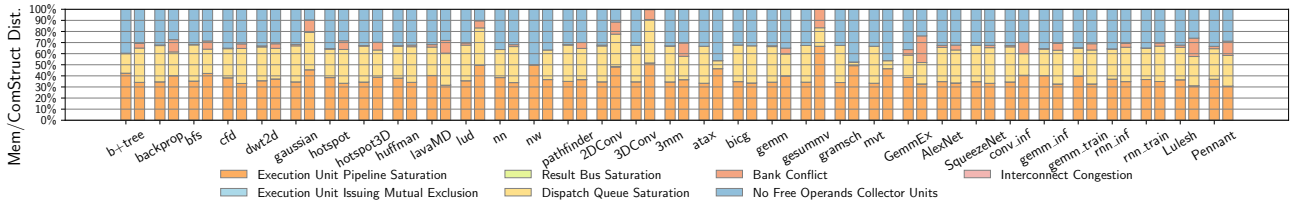
D. Simulation of Concurrent Kernels

In Figure 16, we simulate the concurrent performance of eight *cuBLAS* kernels with the GV100 GPU configuration. Each performs a 1024×1024 matrix multiplication concurrently to achieve a 2048×2048 matrix multiplication. Our simulation shows, concurrency reduces the simulation cycles by 32.87% and increases the occupancy by 14.12%, resulting in a 48.13% IPC increase. Since L1 cache stores data copies from multiple concurrent kernels, its hit rate decreases slightly. Conversely, the L2 cache hit rate improves because it stores a single copy of data reused among concurrent kernels. By comparing the performance of concurrent kernels obtained through *Nsight*'s *Range Replay* mode, HyFiSS shows competitive accuracy.

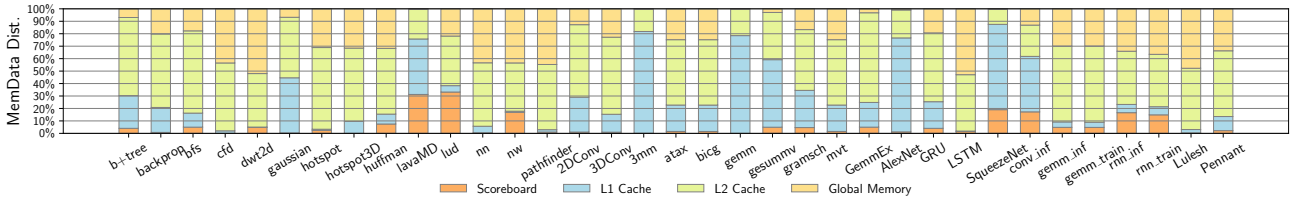
Figure 16f illustrates the stall distribution of concurrent kernels execution. With the increase in the number of concurrent instructions, the proportions of *MemStruct* and *ConStruct* stalls rise. Figure 16g indicates that concurrent instructions further complicate register mapping and significantly increase bank conflicts. Meanwhile, Figure 16h shows that the distribution of *ComStruct* stalls remains relatively stable because the proportion of compute instructions has not changed significantly. Here we primarily present stall events distribution rather than their absolute cycle count, as the former provides more intuitive insights for optimizing concurrent kernel execution.



(a) Stall Distribution Across Different Stall Types for Various Applications.



(b) Distribution of MemStruct (left bar) and ComStruct (right bar) Stall Events.



(c) Distribution of MemData Stall Events.

Fig. 13: Distribution of Stall Types and Stall Events.

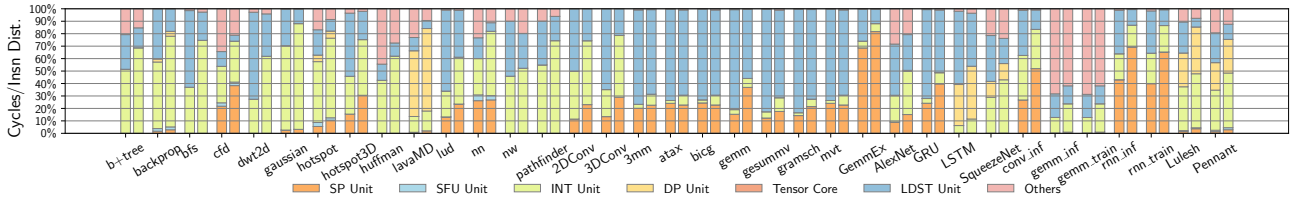
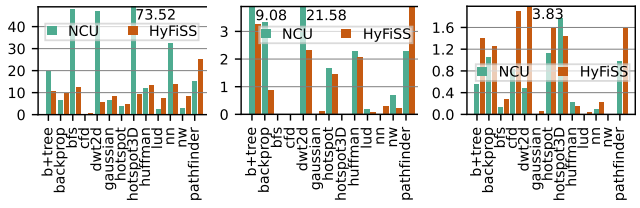
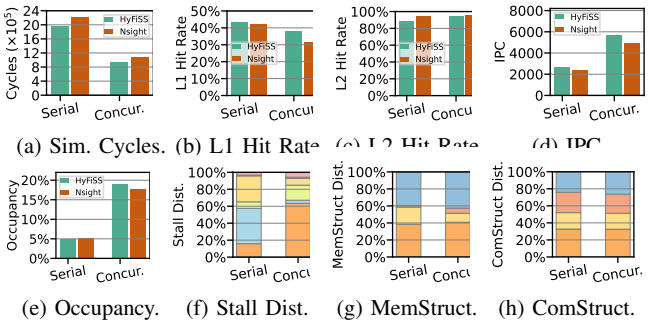


Fig. 14: Execution Time (left bar) and Executed Instructions (right bar) Distribution of Execution Units.



(a) Scoreboard. (b) Barrier. (c) Execution Units.

Fig. 15: Average Latency Between Consecutive Instructions.



(a) Sim. Cycles. (b) L1 Hit Rate (c) L2 Hit Rate (d) IPC

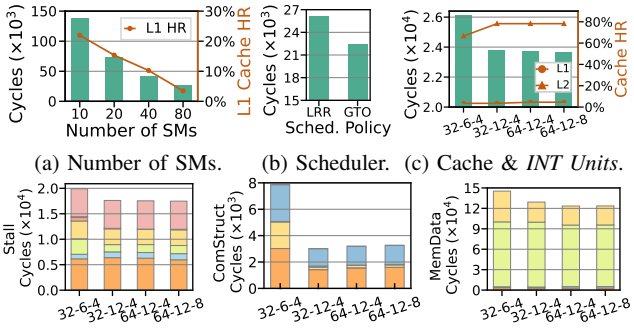
(e) Occupancy. (f) Stall Dist. (g) MemStruct. (h) ComStruct.

Fig. 16: Prediction of Concurrent Kernels on GV100. The legends in Figure 16f through Figure 16h match those in Figure 13.

E. Optimizing Architecture Design with HyFiSS

For a specific architecture and application, HyFiSS captures the root stall events in each cycle and provides the representative stall event stacks. Besides bottleneck detection, they also offer valuable insights to guide architecture design improvement. The fast and precise simulation of HyFiSS enables efficient comparison of different design choices. As a demonstration, Figure 17 shows the effect of changing (a) the number of SMs, (b) the warp scheduling policy from *LRR* to *Greedy-Then-Oldest* [91] (*GTO*), and (c) the L1 cache size with *hotspot*. Instead of enumeration of parameters, architects

can leverage the stall event stacks to quickly and accurately identify the main bottlenecks and optimization opportunities. Figure 13a identifies the primary stall types of *hotspot*: *ComStruct* and *MemData*. In Figure 17, we try to optimize the architecture to reduce the above two stall types for *hotspot*. For *ComStruct* stalls, we simply increase the number of *INT Units* to mitigate *Execution Unit Pipeline Saturation*, as the high distribution of INT Unit indicated by Figure 14. For *MemData* stalls, Figure 13c highlights that the root stall event is high L2 cache access latency. Our goal is to enhance L1 and L2 cache hit rates to reduce high memory access latencies. L1 cache enlargement shows minimal impact due to *hotspot*'s weak temporal locality, leading us to enlarge the L2 cache size. Figure 17 confirms these optimizations decreased *MemData* stall cycles by 15.45%, cut global memory access latency by 38.16%, and lowered the *Sync* stalls overhead. Furthermore, boosting the number of *INT Units* reduced stall cycles of *Execution Unit Pipeline Saturation* by 47.04%. Collectively, the three steps—doubling the L1 and L2 cache sizes, and the number of INT Units—illustrated in Figure 17c represent the improvements that reduced the execution time of *hotspot* on the GV100 GPU by 9.29%.



(a) Number of SMs. (b) Scheduler. (c) Cache & *INT Units*.

(d) Stall Cycles. (e) *ComStruct* Cycles. (f) *MemData* Cycles.

Fig. 17: Performance Effect of GPU Adjustments for *hotspot*. The *x*-axis values in Figure 17c through Figure 17f represent the L1 cache size (KB), L2 cache size (MB), and the number of *INT Units*, respectively. The legends in Figure 17d through Figure 17f match those in Figure 13.

VII. Related Work

A. Simulation Frameworks

1) With Tightly-Coupled Hardware Parameters: Simulators with tightly-coupled hardware parameters primarily use clock-driven simulation and heavily depend on detailed hardware parameters, covering general-purpose simulators like GPGPU-Sim [6], Accel-Sim [7], NVArchSim [9], MacSim [12] for Intel GPUs, MGPUSim [11] for AMD GPUs, and the FPGA-based simulator Vortex [92]. Some application-specific simulators, such as Vulkan-Sim [93] with ray-tracing support, [94] with CuDNN and PyTorch support, [1] with Tensor Core units, and graphics rendering simulators like Emerald [95] and ATTILA [96] with OpenGL support, also require detailed hardware parameters. However, such detail-oriented simulators often trade-off simulation speed for accuracy, hindering extensive application analysis. PPT-GPU [8] simplifies the compute model but fails to capture stalls accurately due to lacking detailed microarchitectural modeling.

2) With Loosely-Coupled Hardware Parameters: Advanced simulation models with loosely-coupled hardware parameters, like roofline [14], [15] and interval-based methodologies such as GPUMech [29], GCoM [31], and MDM [30], as well as latency hiding models [16], [17], enable faster performance predictions. However, their less detailed nature can obscure understanding performance bottlenecks. Data and control flow graph-based approaches [18], [21]–[23] attempt to uncover data dependencies but may fail to account for dynamic stalls. Moreover, critical microarchitectural behaviors are often over-simplified in intermediate representations [90], prompting the need for low-level (*SASS*) instruction analysis. Neural network-based models [19], [20] offer quick performance estimates but lack transparency in bottleneck diagnosis.

HyFiSS is a simulator with tightly-coupled hardware parameters based on *SASS* instructions, and the first to construct a hybrid-fidelity hardware model based on stall events.

B. Sampling Techniques

For simulation acceleration, *phase*-based sampling of workloads has been extensively researched [97]–[101], utilizing *phase*-detection techniques based on instruction segments similarities [27]. TBPoint [72] extends the *phase*-detection to the kernel and thread block levels. Photon [71] utilizes basic block feature vectors (BBVs) for *phase*-detection and sampling at the kernel, warp, and basic block levels. PKA [25] selects the kernels with the most significant impact on execution time at the kernel level, while Sieve [102] achieves kernel-level sampling using an instruction count-based selection method. Scale-Model [103] employs Sieve [102] for sampling representative kernels and predicting the performance of target GPU system through both scale-modeling and LLC miss rate curves. GT-Pin [37] evaluates the error rates across different sampling granularities. HyFiSS differentiates by performing SM sampling based on *CTA-Sets* to minimize the impact of sampling points on microarchitectural states.

VIII. Conclusion

This work introduces HyFiSS, a cycle-approximate system-level NVIDIA GPU simulator. HyFiSS constructs a hybrid fidelity model according to the correlation between stall events and hardware units. Through tracking and attributing various stall types, HyFiSS can capture their impact during the trace simulation process. Furthermore, the SM sampling and traces compression employed by HyFiSS provide better scalability compared to state-of-the-art simulators. We have open-sourced the complete HyFiSS toolchain. Our results show that HyFiSS achieves a balance between reliability and speed, efficiently supporting bottleneck analysis for various benchmarks.

HyFiSS currently faces a few limitations. It reproduces the dynamic CTA scheduling behavior of real hardware, which

restricts the implementation of user-defined CTA scheduling policies. We plan to support custom CTA scheduling policies in future work. Additionally, further calibration is needed to improve HyFiSS's accuracy, particularly for the Ampere A100 GPU. Based on feedback from *Nsight*, HyFiSS also needs to adjust its conflict management entries. Modern architectures bring significant advancements, such as the *Transformer Engine* in Hopper and uGPUs connected via *NV-HBI* in Blackwell. It is crucial that HyFiSS be extended to support these innovative components. Future improvements will also include adding simulation support for *PTX* instructions to enhance HyFiSS's compatibility with contemporary GPUs.

ACKNOWLEDGMENT

We express our gratitude to the anonymous reviewers and shepherd for their constructive feedback, which has undoubtedly strengthened our paper. We also extend our thanks to all mentors who provided guidance for our work. Finally, we are grateful to our group members who contributed to the development of HyFiSS. This work was supported by the National Natural Science Foundation of China (Grant No.U23A20301) and the NUDT Foundation (Grant No.ZK2023-16).

ARTIFACT APPENDIX

A. Abstract

This artifact includes the source code of HyFiSS along with the necessary scripts and instructions to reproduce the key experimental results presented in our paper, such as Figure 2, Figure 8, Figure 11, Figure 13, and Figure 14. Additionally, we provide a fully tested docker image with a pre-configured experimental environment for user convenience. To avoid runtime environment issues, we strongly recommend using HyFiSS within our pre-configured docker image.

This artifact also includes necessary scripts to reproduce the results of Accel-Sim, PPT-GPU, and *Nsight Compute*. However, to avoid the need for reproducing the results of these other simulators and tools, and storing their required traces, the results of these tools are provided within our artifact. We have pushed the stable docker image and code of benchmark suites to Zenodo repository [104], and the HyFiSS code along with potentially updatable scripts to GitHub repository [38]. This way, users can easily access and use our artifact while ensuring that it is always up to date.

B. Artifact check-list (meta-information)

- **Compilation:** GCC v7.3 or above, CUDA v11.0 or above, Nsight Compute v2023.1.1 (comes with CUDA v12.0), Make v4.2 or above, Boost C++ Libraries v1.74.0 or above, MPICH v3.3.2.
- **Binary:** NVBit v1.5.1.
- **Run-time environment:** NVIDIA Driver v535.113.01 or above, Python 3.7 or above.
- **Hardware:** Real GV100 and A100 GPUs.
- **Metrics:** *Simulation Time (seconds)*, *Active Cycles*, *IPC*, *Achieved Occupancy*, *Mean Absolute Percentage Error (MAPE)*, *Mean Absolute Error (MAE)*, *Pearson's Correlation Coefficient (Corr.)*, *Distribution of Stall Types and Stall Events*, *Execution Time Distribution of Execution Units*, and *Executed Instructions Distribution of Execution Units*.
- **Output:** Detailed summary Excel files and figures in the paper.
- **Experiments:** Generate experiments using supplied scripts.
- **How much disk space required (approximately)?:** 500 GB.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours. Mostly depends on downloading bandwidth.
- **How much time is needed to complete experiments (approximately)?:** 1 week. Mostly depends on CPU performance.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** None.
- **Archived (provide DOI)?:** 10.5281/zenodo.13150677

C. Description

1) How to access: GitHub [38] and Zenodo [104].

2) Hardware dependencies: Any Linux system with real GV100 and A100 GPUs and at least 500 GB of free disk space.

3) Software dependencies:

- `libboost_mpi.so >= 1.74.0`
- `git >= 1.8.3.1`
- `gzip >= 1.5`
- `wget >= 1.14`
- `g++ >= 7.3`
- `make >= 4.2`
- `docker >= 19.03`
- `ncu >= 2023.1.1`
- `nvcc >= 11.0`
- `mpirun == 3.3.2`
- `python >= 3.7`
  - `pandas == 1.1.5`
  - `Pillow == 9.5.0`
  - `numpy == 1.21.6`
  - `matplotlib == 3.5.3`
  - `scipy == 1.1.0`
  - `xlrd == 1.1.0`
- `mpiexec == 3.3.2`

4) Data sets or models: None.

D. Installation using our pre-configured docker image

1) Download our pre-configured docker image micro57.tar.gz—the non-root username and password are both *micro57*, and the root password is also *micro57*—from Zenodo repository, save it to the <Download>/ directory, unzip it, and load it:

```
$ wget -O <Download>/micro57.tar.gz https://zenodo.org/records/
  13150677/files/micro57.tar.gz?download=1
$ gzip -d <Download>/micro57.tar.gz
$ docker load -i <Download>/micro57.tar
```

2) Check the <IMAGE_ID> of the loaded docker image, then create a container, and run it:

```
$ docker image list    # check the <IMAGE_ID>
$ docker run --gpus all -it <IMAGE_ID> /bin/bash
```

3) We have already included the files that need to be cloned and downloaded in the docker image, so users only need to check for updates in the corresponding repositories and build them:

```
$ su micro57    # enter the password: micro57
$ source ~/.bashrc
$ cd ~/HyFiSS && git pull && make clean && make all -j
$ cd ~/HyFiSS/tracing-tool && make clean && make
$ cd ~/HyFiSS/sass-split && make clean && make
$ cd ~/Paper-Figures && git pull
$ cd ~/Results-Process-Scripts && git pull
$ cd ~/Self-Simulated-Results && git pull
$ cd ~/simulator-apps && ./cleanall.sh && ./buildall.sh
```

E. Installation from the source code

We strongly recommend that users follow the directory tree structure provided below, including placing all directories that need to be cloned and downloaded under the ~/ directory, because our scripts work according to this structure.

1) Clone HyFiSS from GitHub repository and build:

```
$ git clone https://github.com/ConvolutedDog/HyFiSS.git ~
```

```
$ cd ~/HyFiSS && make clean && make all -j
$ cd ~/HyFiSS/tracing-tool && make clean && make
$ cd ~/HyFiSS/sass-split && make clean && make
```

2) Clone Paper-Figures from GitHub repository:

```
$ git clone https://github.com/ConvolutedDog/Paper-Figures.git ~
$ mkdir ~/Paper-Figures/figs
```

3) Download Pre-Provided-Results.tar.gz from Zenodo repository, save it to the <Download>/ directory, move it into the ~/ directory, and unzip it:

```
$ wget -O <Download>/Pre-Provided-Results.tar.gz
  https://zenodo.org/records/13150677/files/Pre-Provided-Results
  .tar.gz?download=1
$ mv <Download>/Pre-Provided-Results.tar.gz ~/
$ cd ~/ && tar zxvf Pre-Provided-Results.tar.gz
$ rm ~/Pre-Provided-Results.tar.gz
```

4) Clone Results-Process-Scripts from GitHub repository:

```
$ git clone https://github.com/ConvolutedDog/Results-Process-
  Scripts.git ~
```

5) Clone Self-Simulated-Results from GitHub repository:

```
$ git clone https://github.com/ConvolutedDog/Self-Simulated-
  Results.git ~
```

6) Download simulator-apps.tar.gz from Zenodo repository, save it to the <Download>/ directory, move it into the ~/ directory, unzip it, and build:

```
$ wget -O <Download>/simulator-apps.tar.gz https://zenodo.org/
  records/13150677/files/simulator-apps.tar.gz?download=1
$ mv <Download>/simulator-apps.tar.gz ~/
$ cd ~/ && tar zxvf simulator-apps.tar.gz
$ rm ~/simulator-apps.tar.gz
$ cd ~/simulator-apps && ./cleanall.sh && ./buildall.sh
```

F. Experiment workflow

1) We have already provided a smoke-test script—named smoke-test.sh—in the ~/Self-Simulated-Results/Self-Simulated-GV100-Results/ directory, which simulates the *hotspot* application and can be used to verify whether HyFiSS can execute successfully. Users need to check the <GPU_ID> of the GV100 GPU, make it visible, and execute the smoke-test script:

```
$ nvidia-smi    # check the <GPU_ID> of the GV100 GPU
$ export CUDA_VISIBLE_DEVICES=<GPU_ID>
$ cd ~/Self-Simulated-Results/Self-Simulated-GV100-Results/
$ bash smoke-test.sh > smoke-test.log
```

   If the records in smoke-test.log show successful generation of reports, it indicates that the smoke-test has passed.

2) After the smoke test is successful, we will use configurations for the GV100 GPU and A100 GPU to simulate all applications in the ~/simulator-apps directory. Please note that real GV100 and A100 GPUs are required to extract the application's traces. This step may take a considerable amount of time, with each script running for 1 to 3 days, primarily depending on CPU performance:

```
$ nvidia-smi    # check the <GPU_ID> of the GV100 GPU
$ export CUDA_VISIBLE_DEVICES=<GPU_ID>
$ cd ~/Self-Simulated-Results/Self-Simulated-GV100-Results/
$ bash Self-Simulated-GV100-Results.sh > Self-Simulated-
  GV100-Results.log
$ nvidia-smi    # check the <GPU_ID> of the A100 GPU
$ export CUDA_VISIBLE_DEVICES=<GPU_ID>
$ cd ~/Self-Simulated-Results/Self-Simulated-A100-Results/
$ bash Self-Simulated-A100-Results.sh > Self-Simulated-A100-
  Results.log
```

3) Next, we will compare the simulation results of HyFiSS with our pre-provided results from Accel-Sim, PPT-GPU, and Nsight Compute, generating Excel files for the GV100 and A100 GPU configurations. Each Excel file will have a maximum of 7 sheets, and each sheet will contain 143 metrics for thousands of kernels. These files will be saved in the ~/Self-Simulated-Results/ directory, named compare_for_self_simulated_GV100_results.xlsx and compare_for_self_simulated_A100_results.xlsx, respectively:

```
$ cd ~/Self-Simulated-Results/Self-Simulated-GV100-Results/
$ ln -s ~/Pre-Provided-Results/Pre-Provided-GV100-Results/
  Accel-Sim-Results ./
$ ln -s ~/Pre-Provided-Results/Pre-Provided-GV100-Results/
  PPT-GPU-Results ./
$ ln -s ~/Pre-Provided-Results/Pre-Provided-GV100-Results/
  NsightCollection ./
$ cd ~/Self-Simulated-Results/Self-Simulated-A100-Results/
$ ln -s ~/Pre-Provided-Results/Pre-Provided-A100-Results/
  NsightCollection ./
$ cd ~/Results-Process-Scripts/
$ python3 read_reports_for_self_simulated_GV100_results.py
$ python3 read_reports_for_self_simulated_A100_results.py
```

   Note: If users utilize the provided docker image, executing the symbolic link step is unnecessary, as it has already been completed.

4) To reproduce the figures mentioned in the Artifact Appendix Abstract:

```
$ cd ~/Paper-Figures/
$ bash Paper-Figures-Self-Simulated.sh
```

   The reproduced figures, which are basically consistent with those inserted in our paper, will be located in the ~/Paper-Figures/figs/ directory.

G. Evaluation

We generate Figure 2a-Figure 2d using the absolute values of *Simulation Time (seconds)*, *Active Cycles*, *IPC*, and *Achieved Occupancy*. For other figures, we evaluate the error in predicting *Active Cycles* compared to *Nsight Compute* using the *Mean Absolute Percentage Error (MAPE)* for HyFiSS, Accel-Sim, and PPT-GPU. The *Mean Absolute Error (MAE)* is used to evaluate the error in predicting cache hit rates for the three simulators compared to *Nsight Compute*. We also use *Pearson's Correlation Coefficient (Corr.)* to evaluate the correlation between the predicted and measured metrics for the three simulators. Finally, We utilize normalized percentages to evaluate the *Distribution of Stall Types and Stall Events*, *Execution Time Distribution of Execution Units*, and *Executed Instructions Distribution of Execution Units* in Figure 13-Figure 14.

H. Expected results

All the expected results are illustrated in the figures, which are basically consistent with the results described in our paper.

I. Notes

1) The results of the bottleneck analysis are closely tied to the specific execution environment, particularly the runtime.
2) Utilizing the docker image may incur some loss in simulation speed compared to running on physical machines.

J. Methodology

Submission, reviewing and badging methodology:

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

REFERENCES

[1] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2019, pp. 79–92.

[2] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 359–371. [Online]. Available: https://doi.org/10.1145/3352460.3358269

[3] Y. Wang, C. Zhang, Z. Xie, C. Guo, Y. Liu, and J. Leng, "Dual-side sparse tensor core," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, June 2021, pp. 1083–1095.

[4] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse gpu kernels for deep learning," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2020, pp. 1–14.

[5] X. Fu, B. Zhang, T. Wang, W. Li, Y. Lu, E. Yi, J. Zhao, X. Geng, F. Li, J. Zhang, Z. Jin, and W. Liu, "Pangulu: A scalable regular two-dimensional block-cyclic sparse direct solver on distributed heterogeneous systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607050

[6] 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*, April 2009, pp. 163–174.

[7] 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)*, May 2020, pp. 473–486.

[8] Y. Arafa, A.-H. Badawy, A. ElWazir, A. Barai, A. Eker, G. Chennupati, N. Santhi, and S. Eidenbenz, "Hybrid, scalable, trace-driven performance modeling of gpgpus," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2021, pp. 1–15.

[9] 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.

[10] 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.

[11] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "Mgpusim: Enabling multi-gpu performance modeling and optimization," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, June 2019, pp. 197–209.

[12] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C.-K. Luk, "Performance characterisation and simulation of intel's integrated gpu architecture," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2018, pp. 139–148.

[13] 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.

[14] E. Konstantinidis and Y. Cotronis, "A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling," *Journal of Parallel and Distributed Computing*, vol. 107, pp. 37–56, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517301247

[15] N. Ding and S. Williams, "An instruction roofline model for gpus," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Nov 2019, pp. 7–18.

[16] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in gpgpu applications," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 11–22. [Online]. Available: https://doi.org/10.1145/2145816.2145819

[17] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 152–163. [Online]. Available: https://doi.org/10.1145/1555754.1555775

[18] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," *SIGPLAN Not.*, vol. 45, no. 5, p. 105–114, jan 2010. [Online]. Available: https://doi.org/10.1145/1837853.1693470

[19] C. Hong, Q. Huang, G. Dinh, M. Subedar, and Y. S. Shao, "Dosa: Differentiable model-based one-loop search for dnn accelerators," in *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2023, pp. 209–224.

[20] L. Braun, S. Nikas, C. Song, V. Heuveline, and H. Fröning, "A simple model for portable and fast prediction of execution time and power consumption of gpu kernels," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, dec 2021. [Online]. Available: https://doi.org/10.1145/3431731

[21] C. Bai, J. Huang, X. Wei, Y. Ma, S. Li, H. Zheng, B. Yu, and Y. Xie, "Archexplorer: Microarchitecture exploration via bottleneck analysis," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 268–282. [Online]. Available: https://doi.org/10.1145/3613424.3614289

[22] X. Wang, K. Huang, A. Knoll, and X. Qian, "A hybrid framework for fast and accurate gpu performance estimation through source-level analysis and trace-based simulation," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019, pp. 506–518.

[23] Z. Li, Y. Ye, S. Neuendorffer, and A. Sampson, "Compiler-driven simulation of reconfigurable hardware accelerators," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, April 2022, pp. 619–632.

[24] Y. Xu, M. E. Belviranli, X. Shen, and J. Vetter, "Pccs: Processor-centric contention-aware slowdown model for heterogeneous system-on-chips," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1282–1295. [Online]. Available: https://doi.org/10.1145/3466752.3480101

[25] C. Avalos Baddouh, M. Khairy, R. N. Green, M. Payer, and T. G. Rogers, "Principal kernel analysis: A tractable methodology to simulate scaled gpu workloads," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 724–737. [Online]. Available: https://doi.org/10.1145/3466752.3480100

[26] M. Agbarya, I. Yaniv, J. Gandhi, and D. Tsafrir, "Predicting execution times with partial simulations in virtual memory research: Why and how," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2020, pp. 456–470.

[27] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, June 2003, pp. 336–347.

[28] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and W.-M. Hwu, "An architectural framework for runtime optimization," *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 567–589, June 2001.

[29] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "Gpumech: Gpu performance modeling technique based on interval analysis," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 268–279.

[30] L. Wang, M. Jahre, A. Adileho, and L. Eeckhout, "Mdm: The gpu memory divergence model," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2020, pp. 1009–1021.

[31] J. Lee, Y. Ha, S. Lee, J. Woo, J. Lee, H. Jang, and Y. Kim, "Gcom: a detailed gpu core model for accurate analytical modeling of modern gpus," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA:

Association for Computing Machinery, 2022, p. 424–436. [Online]. Available: https://doi.org/10.1145/3470496.3527384

[32] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: where have all the cycles gone?" *ACM Trans. Comput. Syst.*, vol. 15, no. 4, p. 357–390, nov 1997. [Online]. Available: https://doi.org/10.1145/265924.265925

[33] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, 2006, p. 175–184. [Online]. Available: https://doi.org/10.1145/1168857.1168880

[34] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 35–44.

[35] H. Jang, J.-E. Jo, J. Lee, and J. Kim, "Rpstacks-mt: A high-throughput design evaluation methodology for multi-core processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 586–599.

[36] N. Corporation. (2024) The user guide for nsight compute cli (v2024.1.0). [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

[37] M. Kambadur, S. Hong, J. Cabral, H. Patil, C.-K. Luk, S. Sajid, and M. A. Kim, "Fast computational gpu design with gt-pin," in *2015 IEEE International Symposium on Workload Characterization*, Oct 2015, pp. 76–86.

[38] J. Yang, M. Wen, D. Chen, Z. Chen, Z. Xue, Y. Li, J. Shen, and Y. Shi, "HyFiSS GitHub repository," 2024. [Online]. Available: https://github.com/ConvolutedDog/HyFiSS.git

[39] N. Corporation. (2017) Nvidia tesla v100 gpu architecture. [Online]. Available: https://images.nvidia.cn/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[40] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *CoRR*, vol. abs/1804.06826, 2018. [Online]. Available: http://arxiv.org/abs/1804.06826

[41] W. W. L. F. Tor M. Aamodt and T. G. Rogers, *General-Purpose Graphics Processor Architectures*. Morgan & Claypool Publishers, 2018.

[42] N. Corporation. (2024) Cuda c++ programming guide (v12.4). [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[43] A. ElTantawy and T. M. Aamodt, "Warp scheduling for fine-grained synchronization," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 375–388.

[44] J. Liu, J. Yang, and R. Melhem, "Saws: Synchronization aware gpgpu warp scheduling for multiple independent warp schedulers," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 383–394.

[45] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2012, pp. 72–83.

[46] S.-Y. Lee and C.-J. Wu, "Caws: Criticality-aware warp scheduling for gpgpu workloads," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug 2014, pp. 175–186.

[47] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 515–527.

[48] A. Barnes, F. Shen, and T. G. Rogers, "Mitigating gpu core partitioning performance effects," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb 2023, pp. 530–542.

[49] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving gpgpu resource utilization through alternative thread block scheduling," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 260–271.

[50] S. Park, K. Cho, and H. Bahn, "Analysis of thread block scheduling algorithms for general purpose gpu systems," in *2021 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, Dec 2021, pp. 1–6.

[51] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 3, jun 2021. [Online]. Available: https://doi.org/10.1145/3451164

[52] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "Nvbit: A dynamic binary instrumentation framework for nvidia gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 372–383. [Online]. Available: https://doi.org/10.1145/3352460.3358307

[53] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[54] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10.

[55] V. Volkov, "Understanding latency hiding on gpus," Ph.D. dissertation, EECS Department, University of California, Berkeley, Aug 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html

[56] S. Darabi, M. Sadrosadati, N. Akbarzadeh, J. Lindegger, M. Hosseini, J. Park, J. Gómez-Luna, O. Mutlu, and H. Sarbazi-Azad, "Morpheus: Extending the last level cache capacity in gpu systems using idle gpu core resources," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2022, pp. 228–244.

[57] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular gpu kernels," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2014, pp. 130–139.

[58] Y. Li, Y. Sun, and A. Jog, "Path forward beyond simulators: Fast and accurate gpu execution time prediction for dnn workloads," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 380–394. [Online]. Available: https://doi.org/10.1145/3613424.3614277

[59] J. Alsop, M. D. Sinclair, R. Komuravelli, and S. V. Adve, "Gsi: A gpu stall inspector to characterize the sources of memory stalls for tightly coupled gpus," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 172–182.

[60] C. Yu, Y. Bai, and R. Wang, "Mipsgpu: Minimizing pipeline stalls for gpus with non-blocking execution," *IEEE Transactions on Computers*, vol. 70, no. 11, pp. 1804–1816, Nov 2021.

[61] R. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[62] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," *SIGPLAN Not.*, vol. 38, no. 5, p. 245–257, may 2003. [Online]. Available: https://doi.org/10.1145/780822.781159

[63] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, p. 2–13, jun 2004. [Online]. Available: https://doi.org/10.1145/1012888.1005691

[64] Y. Arafa, G. Chennupati, A. Barai, A.-H. A. Badawy, N. Santhi, and S. Eidenbenz, "Gpus cache performance estimation using reuse distance analysis," in *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, Oct 2019, pp. 1–8.

[65] Q. Wang, X. Liu, and M. Chabbi, "Featherlight reuse-distance measurement," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019, pp. 440–453.

[66] C. CaBcaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proceedings of the 17th Annual International Conference on Supercomputing*, ser. ICS '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 150–159. [Online]. Available: https://doi.org/10.1145/782814.782836

[67] G. Chennupati, N. Santhi, S. Eidenbenz, and S. Thulasidasan, "An analytical memory hierarchy model for performance prediction," in *2017 Winter Simulation Conference (WSC)*, Dec 2017, pp. 908–919.

[68] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "Parda: A fast parallel reuse distance analysis algorithm," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 1284–1294.

[69] S. L. Harris and D. Harris, "8 - memory systems," in *Digital Design and Computer Architecture*, S. L. Harris and D. Harris, Eds. Morgan Kaufmann, 2022, pp. 498–541. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128200643000088

[70] M. Wolf, "Chapter 3 - cpus," in *Computers as Components (Fourth Edition)*, fourth edition ed., ser. The Morgan Kaufmann Series in Computer Architecture and Desi, M. Wolf, Ed. Morgan Kaufmann, 2017, pp. 99–159. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128053874000030

[71] C. Liu, Y. Sun, and T. E. Carlson, "Photon: A fine-grained sampled simulation methodology for gpu workloads," in *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2023, pp. 1227–1241.

[72] J.-C. Huang, L. Nai, H. Kim, and H.-H. S. Lee, "Tbpoint: Reducing simulation time for large-scale gpgpu kernels," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 437–446.

[73] N. Nikoleris, L. Eeckhout, E. Hagersten, and T. E. Carlson, "Directed statistical warming through time traveling," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 1037–1049. [Online]. Available: https://doi.org/10.1145/3352460.3358264

[74] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers. (2021) Gpu microbenchmark. [Online]. Available: https://hub.nuaa.cf/accel-sim/accel-sim-framework/tree/release/util/tuner/GPU_Microbenchmark

[75] B. Research. (2016) Benchmarking deep learning operations on different hardware. [Online]. Available: https://github.com/baidu-research/DeepBench

[76] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Detailed characterization of deep neural networks on gpus and fpgas," in *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 12–21. [Online]. Available: https://doi.org/10.1145/3300053.3319418

[77] N. Corporation. (2024) The api reference guide for cublas, the cuda basic linear algebra subroutine library (v12.4). [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[78] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-641973, August 2013.

[79] C. R. Ferenbaugh, "Pennant: an unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice & Experience*, vol. 27, no. 17, p. 4555–4572, dec 2015. [Online]. Available: https://doi.org/10.1002/cpe.3422

[80] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2011, pp. 308–317.

[81] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 235–246.

[82] L. L. N. Laboratory. (2018) Livermore unstructured lagrangian explicit shock hydrodynamics. [Online]. Available: https://github.com/LLNL/LULESH/tree/2.0.2-dev

[83] N. Santhi, S. Eidenbenz, and J. Liu, "The simian concept: Parallel discrete event simulation with interpreted languages and just-in-time compilation," in *2015 Winter Simulation Conference (WSC)*, Dec 2015, pp. 3013–3024.

[84] J. Developers. (2021) Joblib: running python functions as pipeline jobs. [Online]. Available: https://joblib.readthedocs.io

[85] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2013, pp. 185–195.

[86] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[87] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, *Graph partitioning and graph clustering*. American Mathematical Society Providence, RI, 2013, vol. 588.

[88] N. Corporation. (2020) Nvidia a100 tensor core gpu architecture. [Online]. Available: https://images.nvidia.cn/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[89] H. Abdelkhalik, Y. Arafa, N. Santhi, and A.-H. Badawy, "Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis," 2022. [Online]. Available: https://arxiv.org/abs/2208.11174

[90] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers, "Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 608–619.

[91] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2012, pp. 72–83.

[92] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, "Vortex: Extending the risc-v isa for gpgpu and 3d-graphics," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 754–766. [Online]. Available: https://doi.org/10.1145/3466752.3480128

[93] M. Saed, Y. H. Chou, L. Liu, T. Nowicki, and T. M. Aamodt, "Vulkan-sim: A gpu architecture simulator for ray tracing," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2022, pp. 263–281.

[94] J. Lew, D. A. Shah, S. Pati, S. Cattell, M. Zhang, A. Sandhupatla, C. Ng, N. Goli, M. D. Sinclair, T. G. Rogers, and T. M. Aamodt, "Analyzing machine learning workloads using a detailed gpu simulator," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2019, pp. 151–152.

[95] A. A. Gubran and T. M. Aamodt, "Emerald: Graphics modeling for soc systems," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, June 2019, pp. 169–182.

[96] 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*, March 2006, pp. 231–241.

[97] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2013, pp. 2–12.

[98] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé, "Taskpoint: Sampled simulation of task-based programs," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 296–306.

[99] A. Sabu, H. Patil, W. Heirman, and T. E. Carlson, "Looppoint: Checkpoint-driven sampled simulation for multi-threaded applications," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, April 2022, pp. 604–618.

[100] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, July 2006.

[101] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe, "Smarts: accelerating microarchitecture simulation via rigorous statistical sampling," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, June 2003, pp. 84–95.

[102] M. Naderan-Tahan, H. SeyyedAghaei, and L. Eeckhout, "Sieve: Stratified gpu-compute workload sampling," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2023, pp. 224–234.

[103] H. SeyyedAghaei, M. Naderan-Tahan, and L. Eeckhout, "Gpu scale-model simulation," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, March 2024, pp. 1125–1140.

[104] J. Yang, M. Wen, D. Chen, Z. Chen, Z. Xue, Y. Li, J. Shen, and Y. Shi, "HyFiSS: A Hybrid Fidelity Stall-Aware Simulator for GPGPUs," Aug. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13150677