NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs

Yuhui Bao
bao.yu@northeastern.edu
Northeastern University
Boston, MA, USA

Yifan Sun
yifan@cs.wm.edu
William & Mary
Williamsburg, VA, USA

Zlatan Feric
feric.z@northeastern.edu
Northeastern University
Boston, MA, USA

Michael Tian Shen
shen.mich@northeastern.edu
Northeastern University
Boston, MA, USA

Micah Weston
weston.m@northeastern.edu
Northeastern University
Boston, MA, USA

José L. Abellán
jlabellan@ucam.edu
Universidad Católica de Murcia
Murcia, Spain

Trinayan Baruah
tbaruah@amd.com
AMD
Santa Clara, CA, USA

John Kim
jjk12@kaist.edu
KAIST
Daejeon, South Korea

Ajay Joshi
joshi@bu.edu
Boston University
Boston, MA, USA

David Kaeli
kaeli@ece.neu.edu
Northeastern University
Boston, MA, USA

ABSTRACT

As GPUs continue to grow in popularity for accelerating demanding applications, such as high-performance computing and machine learning, GPU architects need to deliver more powerful devices with updated instruction set architectures (ISAs) and new microarchitectural features. The introduction of the AMD RDNA architecture is one example where the GPU architecture was dramatically changed, modifying the underlying programming model, the core architecture, and the cache hierarchy. To date, no publicly-available simulator infrastructure can model the AMD RDNA GPU, preventing researchers from exploring new GPU designs based on the state-of-the-art RDNA architecture.

In this paper, we present the NaviSim simulator, the first cycle-level GPU simulator framework that models AMD RDNA GPUs. NaviSim faithfully emulates the new RDNA ISA. We extensively tune and validate NaviSim using several microbenchmarks and 10 full workloads. Our evaluation shows that NaviSim can accurately model the GPU's kernel execution time, achieving similar performance to hardware execution within 9.92% (on average), as measured on an AMD RX 5500 XT GPU and an AMD Radeon Pro W6800 GPU.

To demonstrate the full utility of the NaviSim simulator, we carry out a performance study of the impact of individual RDNA features, attempting to understand better the design decisions behind these features. We carry out a number of experiments to isolate each RDNA feature and evaluate its impact on overall performance, as well as demonstrate the usability and flexibility of NaviSim.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; • **Computing methodologies → Modeling methodologies**; *Model verification and validation.*

KEYWORDS

GPU, Simulation, Computer Architecture

ACM Reference Format:
Yuhui Bao, Yifan Sun, Zlatan Feric, Michael Tian Shen, Micah Weston, José L. Abellán, Trinayan Baruah, John Kim, Ajay Joshi, and David Kaeli. 2022. NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 8–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3559009.3569666

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PACT '22, October 8–12, 2022, Chicago, IL, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9868-8/22/10…$15.00
https://doi.org/10.1145/3559009.3569666

1 INTRODUCTION

GPUs have been used to accelerate a wide range of modern data-centric applications (e.g., artificial intelligence [26], big-data analytics [1], and high-performance computing workloads [18]), leveraging their ever-increasing computing capabilities [32]. With the continuous demand for higher performance, GPU vendors (e.g., NVIDIA and AMD) have been pushing the envelope of GPU performance in every new GPU generation. While some GPU generations are mainly "spec bumps", other GPU generations introduce major

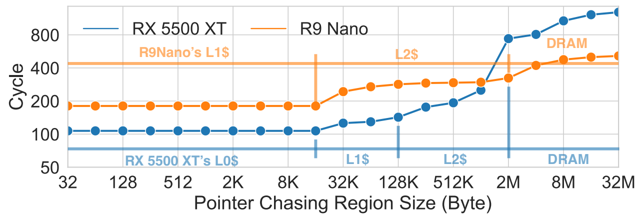

Figure 1: Comparing cache latencies between the GCN and RDNA GPUs, while running the pointer chasing microbenchmark. The results suggest that, other than the publicly announced changes in the specifications (e.g., cache sizes), AMD has made many unannounced design improvements (e.g., reducing cache latencies).

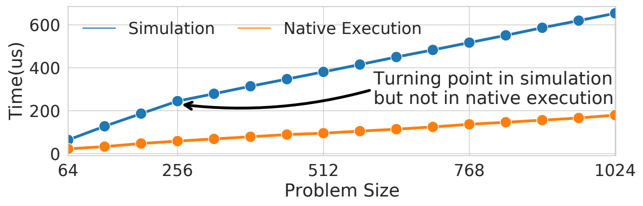

Figure 2: A comparison of a simulator's reported kernel execution time and the native RX 5500 XT GPU execution time when running the BICG benchmark from PolyBench [30]. The experiment is performed using MGPUSim [33], after only modifying the publicly available parameters (core count, frequency, cache sizes, DRAM bandwidth, etc.) based on the validated R9 Nano model. Not only are the performance discrepancies high (up to 318%), but the performance does not follow the trends seen on the RX 5500 XT hardware.

architectural overhauls, changing many aspects of the GPU's organization. For example, in 2019, AMD moved away from the GCN architecture (used by almost all AMD GPUs since 2011) and started to roll out a new RDNA architecture to be used on their 7nm GPUs.

AMD's RDNA architecture was a major redesign, as AMD modified nearly every aspect of the GPU's architecture. This included changes to the instruction set architecture (ISA), core architecture, and memory hierarchy. In the cores, the RDNA architecture reduced the number of co-scheduled threads (i.e., the wavefront) from 64 to 32 to cope with a higher level of thread divergence in modern workloads. The memory hierarchy was also extended, adding an extra layer of cache between the original L1 and L2 caches. This reduced the burden of the L2 caches and simplified the massive L1 to L2 network. In addition to these publicly known changes, enhancements were also made under the hood. For example, as we evaluate the AMD RX 5500 XT GPU (RDNA architecture) against the R9 Nano GPU (GCN3 Architecture) with a pointer chasing microbenchmark (see Figure 1), the results suggested that the first-level cache latency was reduced from ≈190 cycles to ≈110 cycles. This type of hidden change are often ignored when configuring the baseline GPU implementation, which can impact the validity of GPU architecture studies.

To date, GPU architecture simulator development in academia has significantly lagged behind industry schedules. The state-of-the-art AMD GPU simulators are still modeling the AMD GCN3 architecture [15, 33] (released in 2015), and no publicly available simulators model the more recent AMD RDNA architectures. The lack of up-to-date simulator infrastructures can impact a GPU architect's ability to explore new innovations targeting next-generation GPUs.

This lack of GPU simulation tools within academia is potentially harmful to the computer architecture research community in the long run. Research papers often describe their research methodology using sentences such as "we modify simulator *S* that is originally validated against product *A* to model product *B* by changing the number of cores and the cache sizes." Typically, this product *B* is a few generations later than product *A*, and the study implicitly assumed that the simulator could still correctly model the new architecture. However, the nuances in the architecture design and parameter selection may significantly change the performance characteristics. For example, simply modifying MGPUSim [33] (validated against a 2015-released R9 Nano GPU) to model the RX 5500 XT GPU (RDNA architecture), as shown in Figure 2, can result in simulation errors as large as 318% when running the BICG benchmark from PolyBench [30]. The major inaccuracies reported above suggests that simply adjusting publicly known parameters is insufficient to model a brand new architecture and can lead to wrong conclusions. A careful redesign and re-calibration of the simulator infrastructure are necessary to provide a trustworthy baseline model for the next generation of GPU architecture research.

To address these issues, we present NaviSim, a GPU simulator that models AMD RDNA GPUs. NaviSim faithfully emulates the new AMD RDNA ISA and produces exact application outputs as recorded on the GPU hardware. Utilizing the Akita simulation engine [33], the same simulation engine that drove MGPUSim [33], we enable modularity and high-performance parallel simulation. We integrate DRAMSim3 [27] to accurately model different DRAM technologies, including HBM [25] and GDDR [19]. NaviSim is also fully compatible with the Daisen GPU visualization framework [36], allowing users to easily understand the GPU's performance issues identified by the simulator. We extensively validate NaviSim against an AMD RX 5500 XT GPU (RDNA architecture). The average simulation error on execution time, as compared to real hardware, is less than 10% across a suite of 10 workloads.

Given that we have a validated simulator, we then use NaviSim to analyze the effect of different RDNA features on a GPU's overall performance, attempting to make sense of the design decisions behind the design of the RDNA architecture. We carry out a number of experiments to isolate each RDNA feature and evaluate its impact. These experiments also serve as case studies, demonstrating the usability and flexibility of NaviSim.

In summary, this paper makes the following contributions:

- We present NaviSim, a highly configurable and accurate GPU simulator that models the AMD RDNA architecture.
- We present validation results, comparing NaviSim simulation results and hardware execution on two GPUs (AMD RX 5500 XT and AMD Radeon Pro W6800).
- We provide case studies demonstrating the utility of NaviSim, evaluating new architectural and design features targeting the RDNA GPU.

2 BACKGROUND

The AMD GCN architecture [2] has been the reference specification for AMD GPU design over the past decade. However, in recent years, the AMD GCN architecture faced critical scalability challenges, as it struggled to fully exploit the ever-increasing number of transistors provided by today's photolithography technology [8]. In response, AMD developed a new family of RDNA architectures [3], introducing a major architectural overhaul as compared to the GCN architecture. In this section, we provide a brief overview of the GCN architecture and discuss the changes introduced in the RDNA architecture. For more detailed specifications, readers can refer to the AMD GCN Whitepaper [2] and RDNA Whitepaper [3].

GPU Programming Model. Both AMD GCN and RDNA GPUs can execute GPU programs implemented using the same GPU programming model. Here, we introduce the programming model using OpenCL terminologies, though other programming models (e.g., CUDA or HIP) use similar semantics.

A GPU program typically can be separated into the GPU portion and the CPU portion. The portion of the program that executes on the GPU is called a *kernel*. The portion of the program that runs on the CPU (i.e., the host program) launches kernels using vendor-provided runtime APIs. A kernel consists of a number of work-items. Work-items are similar to CPU threads; they execute the same program concurrently, but work on different data. Work-items can be grouped into work-groups. The work-items in a work-group (typically 32–1024 work-items) can be synchronized using barriers and can share a small, but fast, local data share (LDS) memory.

On a GPU, a subset of work-items (typically 32–64) in a work-group are organized into a wavefront. AMD GPUs issue instructions at a wavefront granularity—every time the instruction scheduler issues one instruction, the ALU repeats the operation for each work-item within the wavefront. Due to the use of this form of wavefront-level scheduling, all the work-items in a wavefront are always synchronized. This mechanism is also known as *lock-step execution*. With lock-step execution, all the work-items in a wavefront need to execute an instruction, even if only part of the wavefront needs it when the execution paths of different work-items diverge. Thread divergence negatively impacts GPU performance, as only part of the work-item execution will take effect.

GCN Architecture. The GCN architecture (see Figure 3(a)) adopts a highly modular design that incorporates a Command Processor, Shader Arrays (including Compute Units and L1 caches), an on-chip network connecting the core-side L1 caches and the memory-side L2 caches, and DRAM. The Command Processor (CP) is responsible for handling all communications with the CPU, including memory copying and kernel launch. The CP is also responsible for breaking down kernels into work-groups and wavefronts, as well as dispatching the work-groups and wavefronts to the Compute Units (CUs). In the GCN3 architecture, the CP can dispatch one wavefront per cycle. However, given the fact that GPU devices can hold an ever-increasing number of CUs, especially when moving to 7nm technology, the CP dispatching speed can be a performance bottleneck. While other simulators (e.g., Multi2Sim [38], GPGPUSim [7]) do not typically model the communication between the CP and the CU, we carefully model the dispatching process, as we have observed that the work-group dispatching can be a major performance bottleneck.

A CU (see Figure 3(c)) on a GPU is similar to a CPU core. The CU is responsible for instruction execution and data processing. Each CU includes a scheduler that can fetch and issue instructions for up to 40 wavefronts. During each cycle, the scheduler can decode up to 5 different instructions and issue these 5 instructions to different execution units, including a branch unit (not shown in the figure for clarity), a scalar unit (responsible for executing instructions that manipulate data shared by work-items in a wavefront), a Local Data Share (LDS) unit, a vector memory unit, and four Single-Instruction Multiple-Data (SIMD) units. Each SIMD unit is responsible for executing vectorized floating-point instructions for 10 out of the 40 wavefronts managed by the scheduler. Each SIMD unit is equipped with 16 single-precision Arithmetic Logic Units (ALUs). Therefore, each 64-work-item wavefront takes 4 cycles to finish the execution of one instruction. Since CU behavior will determine instruction throughput and needs to handle data dependencies, modeling and analyzing the behavior of the CU accurately is essential to accurately modeling the overall GPU performance.

The GCN architecture has a two-level cache hierarchy. The L1 cache can be divided into the L1 scalar cache (mainly used for storing constant data, such as kernel arguments and pointers), an L1 instruction cache, and an L1 vector cache (a write-through cache that stores most of the data required by a CU). Each CU has a dedicated L1 vector cache. CUs in a Shader Array (typically 4 CUs) share an L1 scalar cache and an L1 instruction cache. All the L1 caches fetch data from L2 caches (L2s are write-back caches). Each L2 cache interfaces to a DRAM controller (typically implemented in HBM or GDDR technology). The L2 caches and the DRAM controllers are banked, allowing them to service a part of the address space. Since many GPU applications are memory bound, careful modeling of the memory system is critical and a goal of NaviSim.

The L1 caches and the L2 caches are connected with a crossbar. The crossbar design can provide low-latency and high-throughput communication channels for the L1 and the L2 caches. However, as the 7nm technology enables a larger number of CUs, the crossbar design struggles to scale. Therefore, future architectures will require design changes targeting the memory hierarchy in order to enhance the scalability of the GPUs.

RDNA Architecture. AMD's RDNA architecture (see Figure 3(b)) is designed to replace the GCN architecture for better scalability. The RDNA architecture makes changes to many elements of the GPU design, including the programming model, the CU, and the memory hierarchy.

One of the most noteworthy changes is that the RDNA architecture reduces the size of the wavefront from 64 work-items to 32 work-items. By cutting the wavefront size in half, the CUs are expected to better cope with a higher degree of thread divergence in modern workloads. Additionally, as the wavefront size is smaller, fewer memory transactions are expected to be generated by one load/store instruction (although the total number of transactions generated by the kernel is likely to remain unchanged), potentially reducing memory access latencies (as demonstrated in the pointer chasing microbenchmark results in Figure 1). These benefits should help to improve ALU utilization. This paper will further evaluate the effects of narrower wavefronts in Section 6.

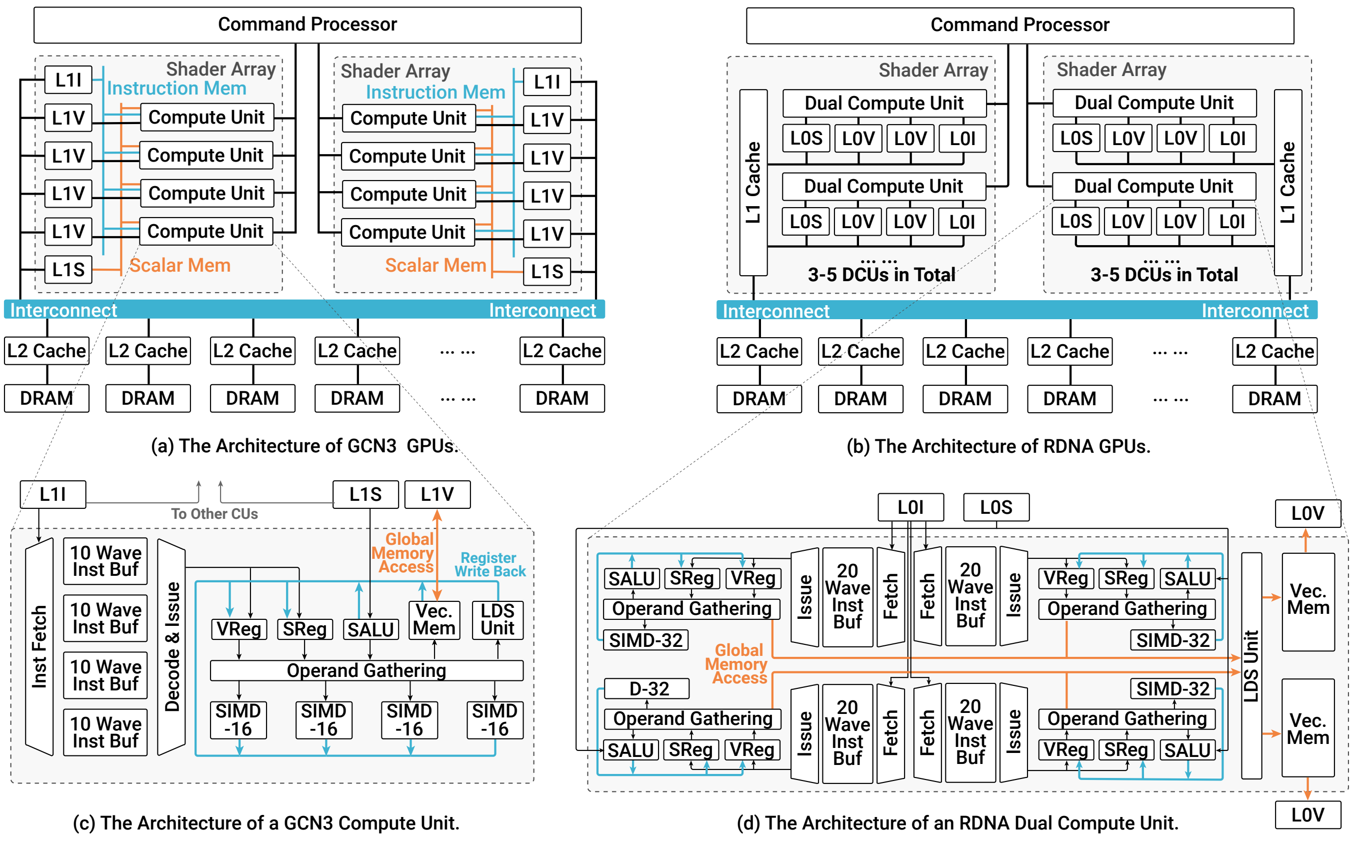

Figure 3: A Comparison between the GCN Architecture [2] and the RDNA Architecture [3].

A second major change in RDNA GPUs is the introduction of Dual Compute Units (DCUs, see Figure 3(d)), replacing the GCN CUs. A DCU contains 4 schedulers. Increasing the number of schedulers from 1 to 4 significantly increases the instruction issue rate. Rather than dispatching instructions across 4 SIMD units, as in the integrated GCN CU, each RDNA scheduler in a DCU dispatches instructions to 1 SIMD unit. One SIMD unit in a DCU has 32 single-precision ALUs, doubling the number of ALUs in a CU. Working together with the narrower 32-work-item wavefronts, each SIMD unit can finish executing one instruction in a single cycle, as compared to 4 cycles in a CU.

Third, the RDNA architecture redefines the cache hierarchy from a 2-layer structure to a 3-layer structure. The caches that are directly connected to the DCUs are renamed as L0 caches (versus L1 caches). Each read-only L0 instruction cache and read-only L0 scalar cache are no longer shared by multiple CUs, but dedicated to a DCU. Each DCU connects with 2 separate write-through L0 vector caches; a group of two schedulers and two SIMD units can use one L0 vector cache. Since now we have two L0 caches connected to one DCU, updating the data in one cache may render the data in the other L0 cache stale. This may cause coherence issues within a DCU and requires explicit cache invalidation instructions (as provided in the RDNA ISA, but not in the GCN ISA).

Additionally, an intermediate level of caching (i.e., the new write-evict L1 cache) is inserted. The L1 cache serves a group of DCUs (typically 4-5) in a Shader Array, and sits between the L0 and L2 caches. The L1 caches can reduce the number of requests arriving at the L2 caches (in the case of L1 hits) and reduce the amount of data that is transmitted across the chip (from L2 to L0), thereby increasing performance and lowering the power consumption caused by cross-chip transmissions. Finally, the cache line size of the L0 vector caches, L1 caches, and the L2 caches is doubled from 64B to 128B, so that a cache line can deliver unique single-precision numbers for all 32 work-items in a wavefront ($4B \times 32 = 128B$).

The Akita Simulation Engine. The Akita Simulation Engine [33] is a computer architecture simulator engine that is implemented in the Go programming language [37]. The Akita Simulation Engine has been used effectively in the MGPUSim simulator [33]. We selected the Akita Simulation Engine because of its high flexibility and optimized multi-threaded simulation performance. While we reuse a few components (e.g., the L0 and L2 caches) from MGPUSim, most of the NaviSim simulator is designed and implemented independently using the Akita Simulation Engine (e.g., the RDNA instruction emulator, the new L1 write-evict cache, and DRAM controllers).

3 NAVISIM

In this section, we present NaviSim, a novel GPU simulator that models the AMD RDNA architecture. NaviSim is open source (link is hidden for double-blind review) under the terms of the MIT license [17].

RDNA ISA Emulation. NaviSim is an execution-driven simulator. The simulator recreates the execution results of GPU instructions during simulation with the help of an instruction emulator for the RDNA ISA. NaviSim can use the MGPUSim's GCN3 instruction emulator as a library. Since the instruction emulator shares the same interface with the GCN3 instruction emulator, this allows users to swap the instruction emulator being used. The virtual driver (a set of APIs that connect the Go-coded host programs and the simulated GPUs) of NaviSim allows users to configure which ISA to emulate and load the corresponding GCN3/RDNA kernel binaries.

NaviSim runs in either emulation mode or timing simulation mode. Instruction emulation mode can recreate execution results, without evaluating detailed timing information for the instruction pipelines, caches, and DRAM controllers. As a result, emulation runs much faster than timing simulation. No matter which mode is used, the emulation results of the benchmarks shown in Table 4 exactly match the output of the applications run on real GPU hardware.

Currently, NaviSim supports both OpenCL [20] kernels and kernels written in the HIP programming language [5]. OpenCL kernels can be compiled by the AMD official `clang-ocl` compiler, which is a standard part of the AMD Radeon Open Compute (ROCm) platform [35]. HIP kernels can be compiled with the `hipcc` compiler, which also ships with the AMD ROCm platform. By using the `-genco` argument, `hipcc` ignores the host program and only generates kernel binaries. NaviSim supports loading kernel binaries compiled by either compiler and emulates the execution of the GPU kernels using actual input data sets.

Wavefront Dispatching. We carefully model the Command Processor to capture the wavefront dispatching process. The modeled Command Processor maintains resource masks that keep track of which resources are occupied in each CU/DCU, including the wavefront slot (wavefront-level resources, such as instruction buffers and the program counter register), scalar registers, vector registers, and LDS memory. The resource masks ensure that no hardware resources are oversubscribed. Since we mask the resources either at the register level (for vector and scalar registers) or at the byte level (for the LDS), we can also model register/LDS fragmentation issues in the CUs/DCUs [29].

NaviSim can also support modeling concurrent kernel execution [43], as the Command Processor has multiple wavefront dispatchers. Each wavefront dispatcher will manage the progress of the currently executing kernel's execution and dispatch a new wavefront when resources free up in the CUs/DCUs. By default, each Command Processor provides 8 dispatchers (the number is configurable), and hence, we allow up to 8 kernels to execute concurrently. The wavefront dispatchers compete for the resources and have equal opportunities to dispatch wavefronts to the CUs/DCUs, ensuring the fairness of the concurrently executing kernels.

DCU modeling. In the RDNA architecture, a dual compute unit replaces the old compute unit. We develop a detailed architectural model of the DCU, which governs how instructions are executed in the simulator. Since the accurate modeling of the DCU is essential for simulation accuracy, NaviSim carefully models the pipeline (see Figure 4) with a multi-stage, multi-issue structure.

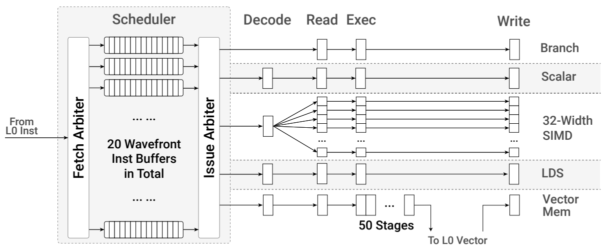

Figure 4: The instruction pipeline model in a NaviSim DCU. Each DCU has four copies of the structure shown.

The pipeline starts with the instruction fetch arbiter, which attempts to fetch instructions for a wavefront that has vacant space in instruction buffers. When multiple wavefronts have space available, the wavefront that received instructions furthest in the past is selected. The issue arbiter monitors the instruction buffers and selects wavefronts that have instructions ready that they can be issued. The arbiter can issue at most 5 instructions per cycle, one instruction to each instruction pipeline.

The Branch, Scalar and LDS pipelines use a fairly regular 6-stage pipeline that includes fetch, issue, decode, read, execute, and write stages. Note that the decode stage happens after the issue stage. This is because the issue arbiter can easily determine the type of each instruction by checking a few encoded bits in the instruction. The branch instruction has no decode stage, as the instructions are very simple. This is in line with publicly available documentation provided by AMD for the DCU architecture. The SIMD unit, which provides most of the computing power of the DCU, also uses a 6-stage pipeline design, but is capable of reading, executing, and writing 32 instructions in parallel. The vector memory has the most complex pipeline structure, adopting a 50-stage pipeline. This may sound unusual, but the model matches our microbenchmarking results. Since we are not aware of the function of each of the pipeline stages, we do not model the behavior of each stage, but only associate a latency value with each instruction.

Memory Hierarchy. We model the 3-level cache hierarchy in RDNA GPUs. L0 scalar and L0 instruction caches are read-only, L0 vector caches use a write-through policy, L1 caches use a write-evict policy, and L2 caches use a write-back policy. We also connect DRAMSim3 [27] to model GDDR5, GDDR5X, GDDR6, GDDR6X, HBM, and HBM2 DRAM controllers. We allow users to configure any number of caches in the hierarchy and allow any type of combinations of the cache policies. Additionally, all of the parameters of a cache (e.g., set count, way associativity, cache line size, directory latency, storage access latency) can be fully configured by users.

Multi-GPU and Interconnect Modeling. Similar to MGPUsim, NaviSim natively supports multi-GPU simulation. The number of GPUs can be easily configured with command line arguments (e.g., `-gpus=1,2,3,4`). There is no limitation on the number of GPUs under simulation, as long as the host machine has enough memory. Additionally, NaviSim natively supports advanced multi-GPU features, such as unified multi-GPU execution [23], GPU-GPU RDMA, and GPU-GPU page migration [13].

User Interface. NaviSim adopts a similar user interface as used in MGPUSim. To run benchmarks, the user can compile either HIP

Table 1: The configurations of validation platforms.

Parameter	Platform1	Platform2
GPU	Radeon RX 5500 XT	Radeon Pro W6800
GPU Core Freq	1845 MHz	2320 MHz
DCU Count	11	30
GPU Memory	GDDR6	GDDR6
Mem Bandwidth	224.0GB/s	512.0GB/s
CPU	AMD Ryzen Threadripper 2950X	AMD EPYC 7302P
OS	Linux Ubuntu 18.04	Linux Ubuntu 20.04
GPU Driver	AMD ROCm 5.0	AMD ROCm 5.1

or OpenCL kernels using the official AMD compiler. A host program written in Go is required to invoke a set of APIs to allocate/copy memory and launch kernels. The host APIs are compatible with MGPUSim and are similar to common GPU programming frameworks (e.g., CUDA, OpenCL). Users can specify either emulation mode (value emulation only) or timing mode (detailed timing simulation that involves instruction pipeline, caches, and memory controllers) as a command line option. Meanwhile, NaviSim uses a few configuration files that are written in Go to define hardware configuration. Users can easily configure the hardware under simulation by modifying parameters and component connections in the configuration code. We use code for configuration because users can easily debug the configuration logic with debuggers.

Simulator Output. NaviSim can generate a wide range of output data to facilitate performance analysis. For high-level metrics, NaviSim outputs the total execution time (kernel time + memory copy time), total kernel execution time, and the per-GPU kernel execution times. For performance metrics related to individual components, NaviSim reports instruction counts, average latency spent accessing each level of cache, transaction counts for each cache (including read misses, read hits, read MSHR hits, write misses, write hits, and write MSHR hits), TLB transaction counts (hits, misses, MSHR hits), DRAM transaction counts and read/write sizes, and transaction counts for the GPU RDMA engines.

Additionally, NaviSim can generate low-level detailed traces, including instruction traces (complete with the register states after executing each instruction) and memory traces (at each level of cache and the DRAM, including the transaction start and end times). NaviSim ships with a graphical user interface (GUI) tool that allows users to navigate the instruction traces and inspect how the registers are updated after each instruction execution. This tool is similar to popular GUI-based MIPS emulators [28] and can be used for educational purposes. Finally, NaviSim can produce traces using the Daisen format [36], so that users can use a visualization tool to inspect the detailed behavior of each component.

4 METHODOLOGY

The studies to be performed in this paper include two parts. First, we calibrate and validate the accuracy of the simulator with a set of microbenchmarks and full benchmarks. With a validated simulator, we then conduct additional experiments to evaluate the RDNA architecture design.

Table 2: Simulator configuration.

Param	P1	P2	P3[*]
Base Model	RX5500XT	R9 Nano	-
# CU/DCU	11(DCU)	64(CU)	32(DCU)
Core Freq	1845MHz	1000MHz	1000MHz
TFLOPS	5.20	8.19	8.19
L0V $[@]	16KB	16KB	16KB
L0V $ Assoc.	4-way	4-way	4-way
L0 Inst $[@]	32KB	32KB	32KB
L0 Scalar $[@]	16KB	16KB	16KB
L0I/L0S $ Org.	Per DCU	4-CU shared	Per DCU
L1 $	128KB	-	128KB
L1 $ Assoc.	16-way	-	16-way
L2 $	1MB	2MB	2MB
L2 $ Assoc.	16-way	16-way	16-way
DRAM Tech	GDDR6	HBM	HBM
DRAM Size	4GB	4GB	4GB
Mem Freq	1750MHz	500MHz	500MHz
Mem Bus	128 bit	4096 bit	4096 bit
Mem BW	224 GB/s	512 GB/s	512 GB/s

[*] P3 is not an off-the-shelf GPU, but it is chosen specifically to analyze the performance impact of individual RDNA features.
[@] We use L0 to name the caches that are directly connected to a CU/DCU. In P2, L0 caches connects to L2 caches directly.

Platforms with GPU Devices. We use an RX 5500 XT GPU and Radeon Pro W6800 (see Table 1) to validate our NaviSim GPU Model. Our platforms run the ROCm 5.0/5.1 software stacks on Linux Ubuntu 18.04/20.04 servers.

Simulator Configuration. We set the baseline GPU configuration using publicly available information and calibrate our results using microbenchmarks. The default configuration of NaviSim for the RX 5500 XT GPUs is recorded as P1 in Table 2.

After validating the NaviSim GPU model, we use the model to conduct a series of use-case experiments to evaluate the performance impact of microarchitectural design features in the RDNA architecture. We compare our simulation results with the default configuration of MGPUSim for the R9 Nano GPUs (denoted as P2 in Table 2). Since the AMD RX 5500 XT and the R9 Nano GPUs belong to two very different markets, comparing them directly would not provide us with a lot of new insights. Therefore, we have configured a GPU as P3 in Table 2. We chose these GPU configurations so that they would have a similar theoretical computing throughput (represented by TFLOPS) and a comparable memory hierarchy. In the experiments, rather than directly comparing P2 and P3, we gradually add features from P2 to build P3, so that we can evaluate the effect of each feature.

Microbenchmarks. We use microbenchmarks to evaluate key parameters of each GPU in order to evaluate the accuracy of NaviSim. We design 7 microbenchmarks (see Table 3) to evaluate individual GPU subsystems, such as the wavefront dispatcher, instruction pipelines, and memory hierarchy. At a high level, we repeat one

Table 3: The list of microbenchmarks used to calibrate NaviSim.

Microbenchmark	Parameter	Description
`empty_kernel`	Work-Group DispatchingSpeed	Executes a kernel, while varying the kernel and work-group sizes. We leave the kernel blank so that we can measure work-group dispatching latency.
`empty_kernel_multi`	Kernel Launching Overhead	Launches a given number of empty kernels. The kernel used is same as the one used in `empty_kernel` microbenchmark.
`single_thread_loop`	Instruction Pipeline Depth	Executes a kernel with only one thread. The kernel has a main loop, and each iteration of the loop executes a single single-precision floating point instruction.
`multi_thread_loop`	Instruction Pipeline Throughput	Executes the same kernel as `single_thread_loop`, but with a large number of threads that are large enough to fully occupy the GPU execution resources.
`pointer_chasing_random`	Cache Sizes and Cache Latencies	Runs the classic pointer chasing microbenchmark [41]. The pointers stored in the buffers are randomized. We only run 1 thread in this kernel.
`pointer_chasing_linear`	Cache Size and Cache Latencies	The same kernel as `pointer_chasing_random`, except that the pointers in the buffers always point to the next pointer (the last pointer points to the first one.
`memory_copy`	Bandwidth of Caches and DRAMs	Copies data from one buffer to another, using a given number of work-groups. When there are only 1-2 work-groups, we evaluate the L0 cache bandwidth. When there are enough work-groups to fill the whole GPU, this microbenchmark can also test the DRAM bandwidth.

operation thousands to millions of times to stress individual components of the GPU. Then we statistically analyze the latency of an operation or evaluate the throughput of a specific GPU component.

As an example of this process, we use the pointer-chasing benchmark to evaluate cache sizes and latencies. The host program first creates a region of memory (i.e., the pointer chasing region) of a given size. Next, the host program divides the region into 8-byte cells and fills each cell with an address that points to the next cell, located at a randomly assigned address within the region. There is no repetition in the addresses stored in the region so that the whole region can be traversed multiple times. The GPU will use a single thread to access the cells many times (at least several times more than the number of cells), following the addresses stored in the region. We eventually divide the kernel execution time by the number of accesses to calculate the average access latency.

Full Benchmarks. We also exploit a list of full GPU benchmarks (see Table 4) from a wide range of benchmark suites including AMDAPPSDK [31], SHOC [10], HeteroMark [34], PolyBench [30], and DNNMark [11]. We use these benchmarks as a set of workloads that cover a wide range of applications to comprise different arithmetic intensities, memory access patterns, and communication patterns. We start with the original OpenCL kernel implementations from the benchmark suite and compile the kernels with the original AMD ROCm compiler (applying default compiler optimizations). We also write host programs in Go, allowing the simulators to call the kernels. We ensure our host programs are equivalent to the original host program from the benchmark suite. For validation experiments, we vary the problem sizes of the benchmarks to make sure that NaviSim can recreate scaling trends. For performance evaluation, we use large problem sizes that are sufficient to stress the whole GPU.

Running Benchmarks. We run OpenCL implementations of the benchmarks, while varying the input size. We use kernel execution time as the performance metric and report the average execution time over 10 runs. The times obtained on the real GPU are recorded using OpenCL events [20].

Table 4: Full Benchmarks.

Abbr.	Suite	Workload
ATAX	PolyBench	Matrix Transpose and Vector Multiplication
BICG	PolyBench	BiCGStab Linear Solver [39]
BS	AMDAPPSDK	Bitonic Sort
FIR	HeteroMark	Finite Impulse Response Filter
FLW	AMDAPPSDK	Floyd-Warshall Algorithm
FWT	AMDAPPSDK	Fast Walsh Transform
KM	HeteroMark	KMeans Clustering
MT	AMDAPPSDK	Matrix Transpose
ReLU	DNNMark	Rectified Linear Unit
SPMV	SHOC	Sparse Matrix-Vector Multiplication

We use kernel execution time as the primary metric to evaluate the accuracy of NaviSim, for two reasons. First, execution time is the most commonly reported metric when considering architectural tradeoffs. Second, the kernel execution time is a high-level metric that summarizes the impact of all features being simulated; we can only achieve a low error in execution time if the fidelity of all the components being modeled in the simulator is high.

5 SIMULATOR VALIDATION

Any computer architecture simulator requires a rigorous validation process before it can serve as a baseline for future research. In this section, we report on our validation efforts for NaviSim, comparing simulation results against GPU hardware execution.

First, we verify the correctness (in terms of application outputs) of NaviSim in both emulation mode and timing simulation mode. To this aim, we compare every simulator-generated application output with its corresponding actual hardware execution output.

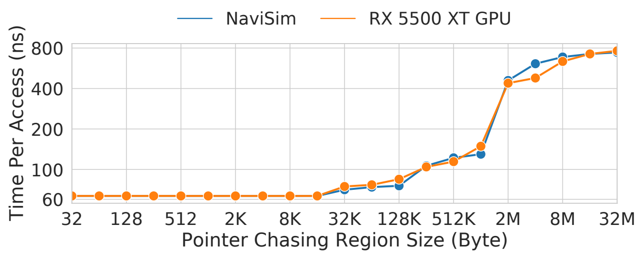

Figure 5: Comparison of the Pointer Chasing microbenchmark between NaviSim and the RX 5500 XT hardware.

We observe that the results match exactly for all applications. The matching results of NaviSim when running in the emulation mode suggests that NaviSim faithfully emulates the RDNA instructions. Additionally, different from other simulators [7, 38], where the instruction emulation and time modeling are independent of each other, NaviSim models data values in every request and every simulated clock cycle. Our simulation approach exposes mistakes in the communication modeling, many times captured as an error in the simulator-generated execution output. Being able to match the results between the timing simulation results and hardware measurements suggests that we faithfully model communication between components. For example, we capture and accurately model all flushes of cache lines, where errors may never be detected by simulators that separate the emulation and timing modeling logic. Blending the instruction and emulation code provides an extra layer of confidence that the components and the communication are modeled accurately.

Second, for validating the timing model, we use both microbenchmarks and full application workloads to validate the accuracy. We configure our simulation using the publicly available parameters of the AMD RX 5500 XT GPU (see Table 1). We then use microbenchmarks (see Table 3) to help us reverse-engineer a wide range of undocumented parameters, such as the work-group dispatch overhead, instruction pipeline depth, cache latency at each level, and DRAM bandwidth and latency. A good example of our strategy is that we use a pointer chasing microbenchmark to figure out the size and latency of each level of the cache. As we can see from the representative results of running the pointer chasing microbenchmark (see Figure 5), NaviSim is able to calibrate the parameters with extremely high accuracy. The calibration results of other microbenchmarks follow similar trends as the pointer chasing results.

With the fully calibrated parameters, we evaluate the accuracy of the simulator by validating it against AMD RX 5500 XT hardware using full benchmarks (Table 4). Figure 6 shows the simulator accuracy (left y-axis) as well as the relative error at each data point by the bar plot (right y-axis) for a range of problem sizes depending upon each benchmark. We observe that for some benchmarks the relative error increases as the problem size increases (e.g., ReLU), and for some other benchmarks the relative error decreases as the problem size increases (e.g., BS, SPMV). For each benchmark, in parenthesis, we also report the average error. As we can see, the error in terms of modeled execution time, averaged across all benchmarks, is just 9.75%. Additionally, we not only model the execution with high fidelity, we clearly capture the patterns and the nuances of the GPU architecture. For example, in the FIR and ReLU benchmarks, we successfully capture the transition in the workload when the execution time starts to increase. This suggests that we properly model any limitations associated with the total amount of computing resources available on a GPU. As another example, we observe steps in some benchmarks (e.g., BS before 32K, FWT after 16K and 32K, KM between 2K and 3K). As these steps are caused by complex interactions between the instruction scheduler, cache hierarchy, and memory transaction handling, being able to model these steps demonstrates that NaviSim can model the subtle features in the RDNA architecture. We have also validated NaviSim against a second GPU model, the AMD Radeon Pro W6800 GPU (see Table 1), which is an RDNA2-based GPU. The average difference between simulated and hardware measured execution time is 10.08%. Our validation results, which are shown in Figure 7, demonstrate that NaviSim can capture the changes of microarchitectures in RDNA across different devices with high fidelity.

We are also aware of discrepancies in a few benchmarks, such as FW and KM. In general, these benchmarks are either short-running benchmarks (FW) or workloads that involve a large number of kernel launches (KM). The discrepancies suggest that NaviSim has difficulties in modeling GPU behaviors at the kernel launch phase. In general, we believe this is not a big problem since the simulators are likely to be used to model large problem sizes, and the differences observed in the kernel launch overhead should not impact the overall accuracy by much. We leave more detailed modeling of the kernel launch behavior as future work.

We also analyze the memory footprint and performance of NaviSim simulation. In the FIR benchmark (with a 4M problem size, as shown in Figure 6), we use 823MB of memory, which fits in the memory of most modern computers. In terms of simulation performance, on an Apple M1 Mac Mini, we achieve 43.5 KIPS and 89.5 KIPS in serial and parallel modes (NaviSim is multi-threaded), respectively. This performance is much faster than MGPUSim, which reported 27 KIPS parallel execution performance in their original paper. Thus, the memory consumption and performance of NaviSim are quite reasonable.

6 CASE STUDIES: UNDERSTANDING THE RDNA ARCHITECTURE FEATURES

With a carefully validated simulator model, we next use NaviSim to perform a set of experiments to analyze the impact of RDNA features on application performance. In particular, we attempt to answer the following questions:

(1) How does the ISA impact the overall performance and how does the DCU architecture impact performance? How is the instruction execution pipeline impacted by executing a different ISA?
(2) What is the effect of the newly added L1 cache?
(3) What percent of the overall performance increase can be attributed to changes in frequency (increased from 1 GHz to 1.845 GHz)?

While we study the impact of the features on the overall performance, we also use this study to demonstrate the utility of NaviSim. We showcase the flexibility and the configurability of NaviSim.

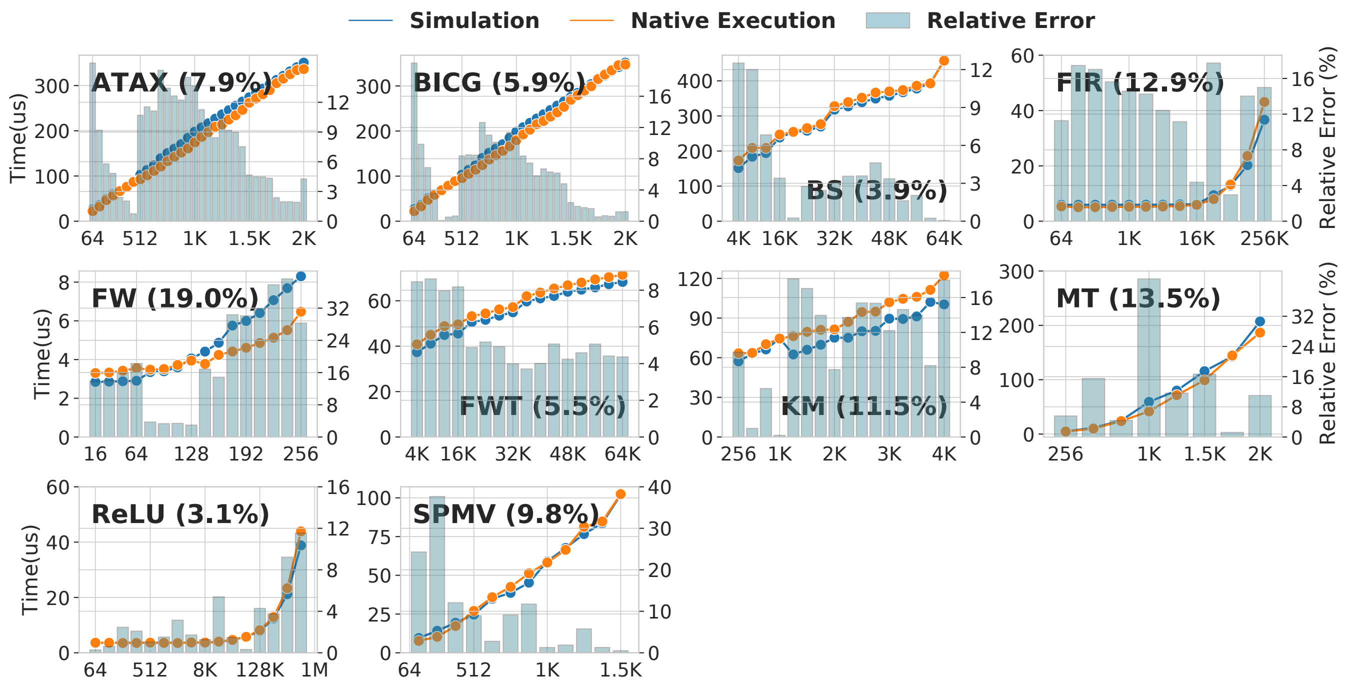

Figure 6: Simulator validation against the AMD RX 5500 XT GPU. The x-axis plots the problem size and the two y-axes plot the kernel execution time and relative error. The numbers in the parentheses represent the average difference between NaviSim simulation and the hardware execution for each application.

Impact of changes in the ISA and the introduction of the DCU. In the first set of experiments, we focus on question (1). We use the P2 configuration (see Table 2, denoted as CU+GFX803 in Figure 8) as the baseline. We have configurations where we either change the ISA to GFX1010 (CU+GFX1010) or change the core to DCU (DCU+GFX803, where the number of DCUs is half of the number in the CU, thus providing a fair comparison of similar computing capabilities). We also provide a configuration that changes both the core and the ISA (DCU+GFX1010).

At a high level, our results (see Figure 8) suggest that many benchmarks (e.g., BS, PR, ReLU) achieve the same performance for all four cases. This is understandable, as the memory system and the overall computing capabilities remain the same. However, we notice major performance differences in ATAX, BICG, FIR, and FLW benchmarks. These changes are caused by the differences in the ISA and the CU/DCU organization.

ATAX and BICG reveal how the CU/DCU organizer impacts the performance of benchmarks that are sensitive to memory bandwidth and latency. ATAX and BICG are workloads with limited parallelism and strong inter-work-item dependencies. Therefore, ATAX and BICG have large work-group sizes and a small number of work-groups in each kernel. In the CU configurations, the number of blocks cannot utilize all the CUs in the GPU and hence, cannot fully utilize the bandwidth between the L0 caches and the CUs. On the contrary, the number of blocks in the ATAX and BICG can fully utilize the DCUs on the third and fourth configurations, since the count of DCUs is halved. As each DCU is connected with two vector L0 caches, the bandwidth between the core and the L0 caches is effectively doubled, causing an increase in the performance by about 2×.

The most critical difference in the ISAs is the wavefront size difference. Kernels compiled to the GFX1010 ISA always use a wavefront size of 32, which is not a perfect match for the CU architecture. When running wavefronts on CUs, because the CU scheduler can only issue 1 instruction to a SIMD unit every 4 cycles, the CU needs to spend 4 cycles to execute 1 instruction. We observe underutilization of the ALUs in ATAX, BICG, FIR, and FLW benchmarks when we match the GFX1010 ISA with the CU microarchitecture. Other benchmarks do not observe this issue because they are bound by memory bandwidth and are not sensitive to ALU utilization. In the FIR benchmark, we also see that matching the GFX803 benchmark and the DCU microarchitecture causes the SIMD unit not to be able to catch up with the instruction issuing speed, leading to significant pipeline stalling and even more slowdown. The problem does not exist when the DCU microarchitecture and GFX1010 ISA are used together, suggesting that the ISA and the microarchitecture are co-designed to achieve the best performance.

Impact of the L1 cache. Next, we focus on question (2) and try to understand the effect of the new L1 cache. We use the P3 configuration (see Table 1, denoted as P3 w/ L1 in Figure 9) as the baseline, and compare the performance with a configuration that removes the L1 cache (P3 w/o L1). The results are shown in Figure 9.

Similar to the earlier experiments, as we only modify a small part of the configuration, most benchmarks that are bounded by

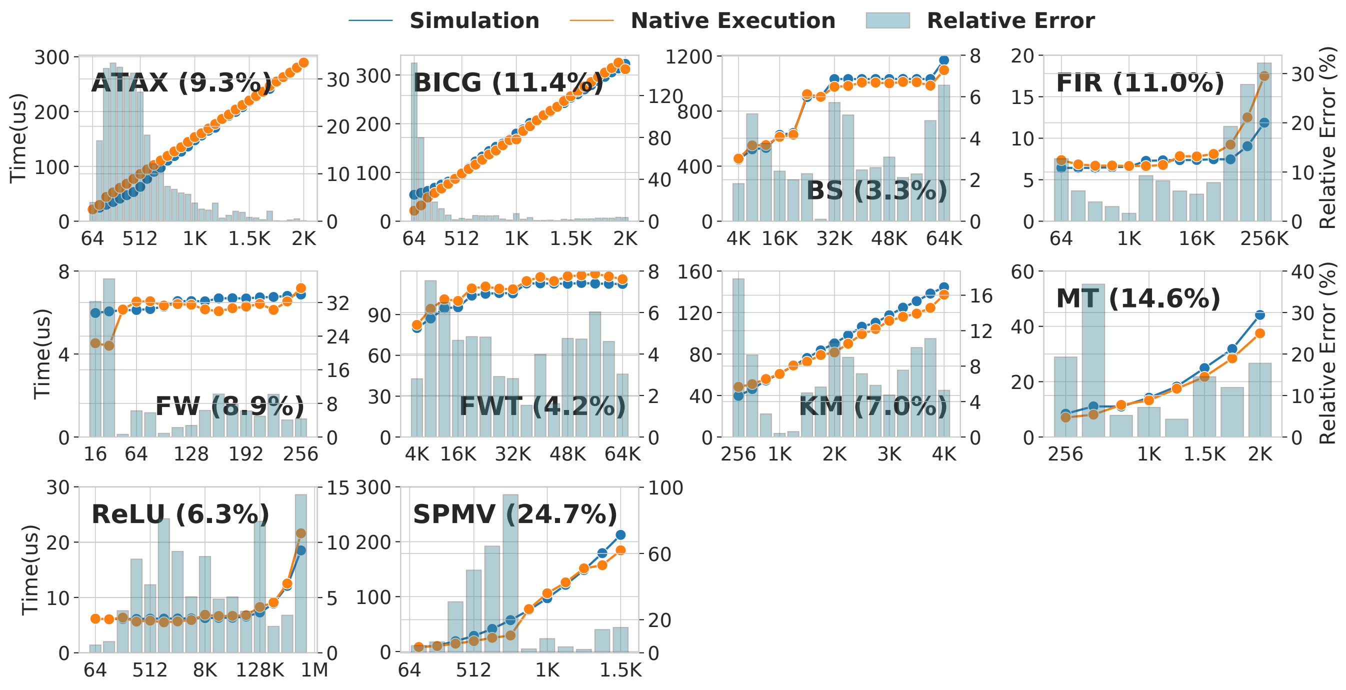

Figure 7: Simulator validation against the AMD Radeon Pro W6800 GPU. The x-axis plots the problem size and the two y-axes plot the kernel execution time and relative error. The numbers in the parentheses represent the average difference between NaviSim simulation and the hardware execution for each application.

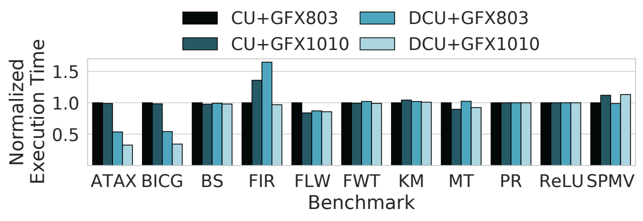

Figure 8: The effect of changing from CUs to DCUs and the ISA from GFX803 (GCN3) to GFX1010 (RDNA) on execution time.

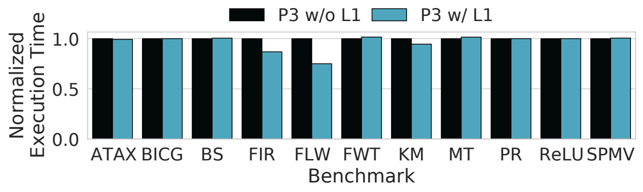

Figure 9: The effect of adding the new write-evict L1 cache on benchmarks' execution time.

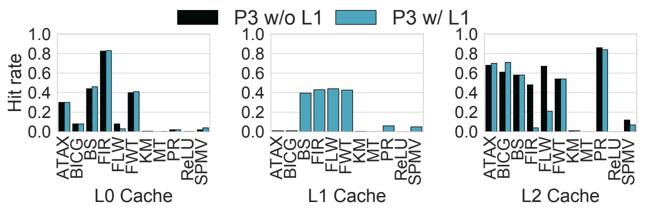

Figure 10: The cache hit rate of each levels of caches before and after the L1 cache is added.

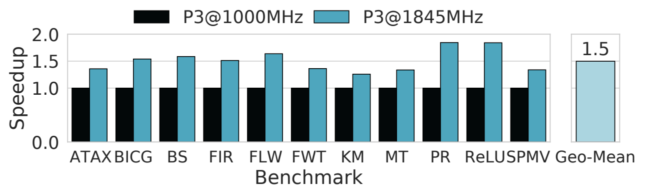

Figure 11: The effect of the increased core frequency on the overall performance. As we increase the core frequency by 1.845×, the performance improved by ≈1.5×.

either compute power or by DRAM bandwidth do not observe a performance difference. However, we did observe performance improvements in the FIR and FLW benchmarks.

To understand why FIR and FLW can benefit from the added L1 cache, we plot the cache hit rate for each level of cache in Figure 10. In general, we see a high degree of diversity across the benchmarks, as these workloads have dramatically different levels of locality. For the BS, FIR, FLW and FWT benchmarks, we observe a relatively high L1 cache hit rate (around 40%). However, the effects of the high L1 cache hit on the L2 caches differ across workloads. For BS and FWT, the L2 cache hit rate remains unchanged, while the L2 cache hit rate drops significantly for FIR and FLW (in a good way). These results suggest that the improved L1 cache hit rates can reduce the number of transactions to the L2 caches for the FIR and the FLW benchmarks. Because the FIR benchmark already has a high L0 hit rate, the speedup caused by the L1 cache is not as high as seen in the FLW benchmark.

Impact of increased frequency. One major change in the RDNA GPUs is the increased frequency. While the increase is not an architectural feature, but instead a benefit of shrinking transistor sizes, we are eager to understand how much the increased frequency can improve performance.

Here, we use P3 as a baseline and increase the GPU core frequency to 1845MHz, which is the frequency that the RX 5500 XT GPU runs at. Note that as we increase the core frequency, we also increase the clock speed that controls the L0, L1, and L2 caches to 1845MHz. In AMD GPUs, the cores and the caches work in the same frequency domain. We do not change the DRAM configuration, so the DRAM latency and bandwidth remain unchanged.

Overall, we see the performance of all the benchmarks improve, with the rate of change ranging from 1.25× to 1.84× (see Figure 11). This suggests that the increase in frequency leads to a marked improvement in GPU performance. This is particularly evident in the PR and ReLU benchmarks, which experience a speedup which is closely correlated with the increase in the core frequency. This suggests that the performance of these two benchmarks is dominated by the GPU frequency. After our further investigation with the Daisen [36] visualization tool, we find that this trend is due to a faster work-group dispatching rate. These two benchmarks have a rather short work-group execution time, so the dispatcher cannot catch up with the work-group retirement speed. Because the CU can complete the work-groups faster than the rate of new work-groups arriving to start execution, the number of work-groups concurrently executing in each CU is limited. This also limits the number of concurrent in-flight memory transactions. Increasing the core frequency accelerates the dispatching speed and increases the occupancy of the CUs, allowing the CUs to better utilize the memory bandwidth. This effect is also observed in the case study discussed in the Daisen paper [36]. Meanwhile, we also notice that the average speedup (geometrical mean) is only 1.5×, suggesting that the increased core frequency still needs to be accompanied by improved DRAM performance.

7 DISCUSSION

Using a simulator to understand the rationale behind architectural changes. Our experiments in Section 6 help us understand part of the design rationale behind changes to AMD GPUs. To the best of our knowledge, our study is the first to perform this type of analysis. Here, we want to reflect on the benefits of having a simulator such as NaviSim available for evaluating the reasons for these changes.

Overall, we find that the flexibility of using a simulator enables us to evaluate specific hardware design choices. We are able to isolate the potential benefits of each architectural feature. For example, we can change the CUs with DCUs, without impacting any other features, including the GPU frequency. This is very challenging to do with RTL and impossible to do on live hardware.

However, we also find some challenges when designing these experiments. One major challenge is determining the validity of the modeled configurations. For example, earlier simulators (e.g., MGPUSim) simply use an ideal network to connect the L1 and L2 caches, only counting the latency of the network as part of the L2 cache latency. However, as we alter the network in our experiments, as shown in Figure 9, we believe that network latency should be lower when the L1 cache is present. Since it is hard to use microbenchmarks to separate out the latency of the network versus the L2 cache for both cases, we may lose some opportunities to identify the true benefits of the L1 cache.

Limitations. While we have shown that NaviSim can achieve high accuracy, there are several unique architectural features that are not presently modeled. For example, we do not support the `CLAUSE` instruction in the RDNA ISA [4], which serves as a performance hint to prevent the DCUs from switching contexts. We also do not implement instructions that explicitly flush the L0 caches. We find these features are rarely used and do not have a major performance impact. We will be implementing both of these features in future work.

Additionally, NaviSim delivers a simulator model for only one generation of AMD GPUs. We do not support simulating other versions of the GPUs and GPUs from other vendors. However, we do not consider this as a disadvantage as focusing on one simulator model allows us to ensure the fidelity of the GPU. Most widely used GPU simulators (e.g., GPGPUSim [7], Multi2Sim [38]) started with a single GPU model and gradually added new models to the simulator infrastructures. Also, considering that NaviSim is developed using the same underlying simulator engine as MGPUSim, users can easily combine the two simulators to simulate both AMD GCN architecture and RDNA architecture.

8 RELATED WORK

GPU Simulators. GPU simulators have been critical infrastructures that enable GPU architecture design validation. To date, the GPU architecture research community has devoted major effort into developing GPU simulators and emulators. Earlier tools include Barra [9] and GPU Ocelot [12, 21]), which provide functional GPU emulation support, though do not support time modeling. Next, GPGPUSim [7] and Multi2Sim [38] were introduced to deliver reliable performance modeling of NVIDIA and AMD GPUs. GPGPUSim has been extended to support additional GPU features, such as virtual addressing [6], concurrent kernel execution [42], parallel simulation [16], and trace-based simulation [22]. Meanwhile, Multi2Sim added support for the NVIDIA Kepler architecture [14].

In recent years, more simulators have been developed, primarily to support newer GPU architectures. For example, Macsim [24] simulates the Intel GPU architecture. The AMD gem5 GPU model [15] is a component added to the gem5 simulator and is dedicated to the AMD GCN3 architecture. MGPUSim [33] is also a high-performance parallel GPU simulator targeting the AMD GCN3 architecture.

Moreover, Accel-Sim [22] is an extension to the GPGPUSim infrastructure that can simulate closed-source GPU programs, utilizing a trace-based simulation method. Accel-Sim also added support for the NVIDIA Kepler, Pascal, Volta, and Turing architectures. Finally, NVArchSim [40] is an internal simulator used by NVIDIA, with much higher performance as compared to GPGPUSim.

The design and development of NaviSim have been inspired by existing simulators. NaviSim shares the simulator core technology with MGPUSim, and hence, inherits the high performance, high flexibility, and multi-GPU simulation capability of MGPUSim. Additionally, to the best of our knowledge, NaviSim is the first simulator that can simulate one of the newest RDNA GPU architectures and is validated with one of the most rigorous validation processes, providing a reliable baseline for future GPU architecture research.

Prior work with GPU simulators has explored and emulating capabilities of different aspects of a GPU. The need for architectural level simulators continuously grows as researchers employ them more within the domain of high performance computing.

9 CONCLUSION

Up-to-date and accurate architectural simulators that can faithfully model today's computing platforms are key toolsets for both developing a comprehensive understanding of current computing design trends, as well as evaluating new design ideas to build forward-looking computing platforms. In the context of AMD GPU platforms, this paper bridges these important gaps by proposing NaviSim, the first cycle-level simulator that targets state-of-the-art RDNA-based GPUs. Through our intensive and rigorous validation methodology, which included developing several microbenchmarks and building on the Akita simulation framework [33], we are able to calibrate NaviSim execution to achieve a small 9.85% average simulation error, as compared to hardware execution on an AMD RX 5500 XT RDNA GPU. To showcase how NaviSim can be used to explore the rationale behind critical architectural design decisions made in the transition from pre-RDNA (GCN-based) GPUs, we carry out a set of use cases to quantitatively analyze the effects of different RDNA features on workload performance. By supporting GPU programs developed in both OpenCL and HIP, NaviSim contributes a valuable simulation framework for further design-space exploration, and enables the research and design of next-generation GPUs based on the RDNA architecture.

ACKNOWLEDGMENTS

This work is supported in part by gifts from AMD, William & Mary start-up funding, and project grant PID2020-112827GB-I00 funded by MCIN/AEI/ 10.13039/501100011033.

REFERENCES

[1] AmirAli Abdolrashidi, Hodjat Asghari Esfeden, Ali Jahanshahi, Kaustubh Singh, Nael Abu-Ghazaleh, and Daniel Wong. 2021. Blockmaestro: Enabling programmer-transparent task-based execution in GPU systems. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, IEEE, New York, NY, 333–346.
[2] AMD Inc. 2012. AMD Graphics Core Next Architecture. https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf
[3] AMD Inc. 2019. Introducing RDNA Architecture, The all new Radeon gaming architecture powering "Navi". https://www.amd.com/system/files/documents/rdna-whitepaper.pdf
[4] AMD Inc. 2020. "RDNA 1.0" Instruction Set Architecture, Reference Guide. https://developer.amd.com/wp-content/resources/RDNA_Shader_ISA.pdf
[5] AMD Inc. 2022. HIP Programming Guide. https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html
[6] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Cambridge, MA, USA, 136–150.
[7] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, IEEE, Boston, MA USA, 163–174.
[8] Tsann-Bim Chiou, Alek C Chen, Mircea Dusa, and Shih-En Tseng. 2016. Impact of EUV patterning scenario on different design styles and their ground rules for 7nm/5nm node BEOL layers. In *Design-Process-Technology Co-optimization for Manufacturability X*, Vol. 9781. International Society for Optics and Photonics, SPIE, Bellingham, Washington USA, 978107.
[9] Caroline Collange, Marc Daumas, David Defour, and David Parello. 2010. Barra: A Parallel Functional Simulator for GPGPU. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, Miami, Florida, USA, 351–360. https://doi.org/10.1109/MASCOTS.2010.43
[10] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1735688.1735702
[11] Shi Dong and David Kaeli. 2017. DNNMark: A Deep Neural Network Benchmark Suite for GPUs. In *Proceedings of the General Purpose GPUs* (Austin, TX, USA) *(GPGPU-10)*. Association for Computing Machinery, New York, NY, USA, 63–72. https://doi.org/10.1145/3038228.3038239
[12] Naila Farooqui, Andrew Kerr, Gregory Diamos, Sudhakar Yalamanchili, and Karsten Schwan. 2011. A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, Newport Beach, CA, 1–9.
[13] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2020. Adaptive page migration for irregular data-intensive applications under GPU memory oversubscription. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, IEEE, New Orleans, Louisiana USA, 451–461.
[14] Xun Gong, Rafael Ubal, and David Kaeli. 2017. Multi2Sim Kepler: A detailed architectural GPU simulator. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, IEEE, Santa Rosa, CA, 269–278.
[15] Anthony Gutierrez, Bradford M Beckmann, Alexandru Dutu, Joseph Gross, Michael LeBeane, John Kalamatianos, Onur Kayiran, Matthew Poremba, Brandon Potter, Sooraj Puthoor, et al. 2018. Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, IEEE, Vienna, Austria, 608–619.
[16] Clayton Hughes, Simon David Hammond, Mengchi Zhang, Yechen Liu, Tim Rogers, and Robert J Hoekstra. 2021. *SST-GPU: A Scalable SST GPU Component for Performance Modeling and Profiling.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).
[17] Open Source Intiative. 1980. The MIT License.
[18] CL Jermain, GE Rowlands, RA Buhrman, and DC Ralph. 2016. GPU-accelerated micromagnetic simulations using cloud computing. *Journal of Magnetism and Magnetic Materials* 401 (2016), 320–322.
[19] JEDEC JESD250. 2017. Graphics double data rate 6 (GDDR6) SGRAM standard. JEDEC Solid State Technology Association.
[20] David R Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. 2015. *Heterogeneous computing with OpenCL 2.0.* Morgan Kaufmann, Burlington,MA,USA.
[21] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2012. Gpu application development, debugging, and performance tuning with gpu ocelot. In *GPU Computing Gems Jade Edition*. Elsevier, Amsterdam, Netherlands, 409–427.
[22] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, IEEE, Valencia, Spain, 473–486.
[23] Gwangsun Kim, Minseok Lee, Jiyun Jeong, and John Kim. 2014. Multi-GPU System Design with Memory Networks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, United Kingdom) *(MICRO-47)*. IEEE Computer Society, USA, 484–495. https://doi.org/10.1109/MICRO.2014.55
[24] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. 2012. *Macsim: A cpu-gpu heterogeneous simulation framework user guide.* Georgia Institute of Technology, Atlanta, GA.
[25] Joonyoung Kim and Younsu Kim. 2014. HBM: Memory solution for bandwidth-hungry processors. In *2014 IEEE Hot Chips 26 Symposium (HCS)*. IEEE, IEEE,

Cupertino, CA, 1–24.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 1097–1105.

[27] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: a cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109.

[28] Mauro Morsiani and Renzo Davoli. 1999. Learning operating systems structure and implementation through the MPS computer system simulator. *ACM SIGCSE Bulletin* 31, 1 (1999), 63–67.

[29] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 527–540. https://doi.org/10.1145/3037697.3037707

[30] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite.

[31] AMD Staff. 2014. Opencl and the AMD App SDK v2. 4.

[32] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David R. Kaeli. 2019. Summarizing CPU and GPU Design Trends with Product Data. *CoRR* abs/1911.11313 (2019), 1–5.

[33] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. 2019. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 197–209. https://doi.org/10.1145/3307650.3322230

[34] Y. Sun, X. Gong, A. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–10. https://doi.org/10.1109/IISWC.2016.7581262

[35] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli. 2018. Evaluating Performance Tradeoffs on the Radeon Open Compute Platform. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, Los Alamitos, CA, USA, 209–218. https://doi.org/10.1109/ISPASS.2018.00034

[36] Yifan Sun, Yixuan Zhang, Ali Mosallaei, Michael D Shah, Cody Dunne, and David Kaeli. 2021. Daisen: A Framework for Visualizing Detailed GPU Execution. *Eurographics Conference on Visualization* 40, 3 (2021), 239–250.

[37] The Go Project. 2019. Effective Go. https://golang.org/doc/effective_go.html.

[38] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (Minneapolis, Minnesota, USA) *(PACT '12)*. Association for Computing Machinery, New York, NY, USA, 335–344. https://doi.org/10.1145/2370816.2370865

[39] Henk A Van der Vorst. 1992. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing* 13, 2 (1992), 631–644.

[40] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. 2021. Need for speed: Experiences building a trustworthy system-level GPU simulator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, IEEE, Seoul, Korea (South), 868–880.

[41] Vasily Volkov and James W Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, IEEE, Austin, TX, USA, 1–11.

[42] Haonan Wang, Fan Luo, Mohamed Ibrahim, Onur Kayiran, and Adwait Jog. 2018. Efficient and fair multi-programming in GPUs via effective bandwidth management. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, IEEE, Vienna, Austria, 247–258.

[43] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. 2011. Exploiting concurrent kernel execution on graphic processing units. In *2011 International Conference on High Performance Computing & Simulation*. IEEE, IEEE, Istanbul, Turkey, 24–32.