MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization

Yifan Sun[1] Trinayan Baruah[1] Saiful A. Mojumder[2] Shi Dong[1] Xiang Gong[1] Shane Treadway[1]
Yuhui Bao[1] Spencer Hance[1] Carter McCardwell[1] Vincent Zhao[1] Harrison Barclay[1]
Amir Kavyan Ziabari[3] Zhongliang Chen[3] Rafael Ubal[1] José L. Abellán[4] John Kim[5] Ajay Joshi[2]
David Kaeli[1]

(yifansun, tbaruah, shidong, xgong, streadwa, ybao, shance, cmccardw, vzhao, hbarclay, ubal, kaeli)@ece.neu.edu,
(msam, joshi)@bu.edu, (aziabari, zhongliang.chen)@amd.com, jlabellan@ucam.edu, jjk12@kaist.edu

[1]Northeastern University, [2]Boston University, [3]AMD, [4]Universidad Católica San Antonio Murcia, [5]KAIST

ABSTRACT

The rapidly growing popularity and scale of data-parallel workloads demand a corresponding increase in raw computational power of Graphics Processing Units (GPUs). As single-GPU platforms struggle to satisfy these performance demands, multi-GPU platforms have started to dominate the high-performance computing world. The advent of such systems raises a number of design challenges, including the GPU microarchitecture, multi-GPU interconnect fabric, runtime libraries, and associated programming models. The research community currently lacks a publicly available and comprehensive multi-GPU simulation framework to evaluate next-generation multi-GPU system designs.

In this work, we present MGPUSim, a cycle-accurate, extensively validated, multi-GPU simulator, based on AMD's Graphics Core Next 3 (GCN3) instruction set architecture. MGPUSim comes with in-built support for multi-threaded execution to enable fast, parallelized, and accurate simulation. In terms of performance accuracy, MGPUSim differs by only 5.5% on average from the actual GPU hardware. We also achieve a 3.5× and a 2.5× average speedup running functional emulation and detailed timing simulation, respectively, on a 4-core CPU, while delivering the same accuracy as serial simulation.

We illustrate the flexibility and capability of the simulator through two concrete design studies. In the first, we propose the *Locality API*, an API extension that allows the GPU programmer to both avoid the complexity of multi-GPU programming, while precisely controlling data placement in the multi-GPU memory. In the second design study, we propose *Progressive Page Splitting Migration* (PASI), a customized multi-GPU memory management system enabling the hardware to progressively improve data placement. For a discrete 4-GPU system, we observe that the Locality API can speed up the system by 1.6× (geometric mean), and PASI can improve the system performance by 2.6× (geometric mean) across all benchmarks, compared to a unified 4-GPU platform.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322230

CCS CONCEPTS

• **Computing methodologies** → **Simulation tools**; • **Computer systems organization** → **Single instruction, multiple data**.

KEYWORDS

multi-GPU systems, simulation, memory management

ACM Reference Format:
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 *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307650.3322230

1 INTRODUCTION

Today's single GPU systems can support a compute throughput of ≈12.4 TFlops [39] to ≈14.7 TFlops [6], and have been redesigned to efficiently accelerate big data analysis [12, 53], machine learning [26, 41], and large-scale physics simulation workloads [21, 42]. However, due to CMOS technology scaling challenges and manufacturing costs, it is becoming increasingly impractical to add more compute resources to a single GPU system to improve its throughput [7]. As a result, these single GPU systems cannot support the processing needs of future data-centric and scientific applications [16, 22, 54, 56].

One attractive path to sustain historic GPU performance scaling, which is currently being pursued by industry, is the integration of multiple GPUs into a single platform. NVIDIA has recently started offering multi-GPU DGX platforms [18, 38], focusing on accelerating Deep Neural Network (DNN) training. However, recent studies suggest that the performance of multi-GPU systems can be heavily constrained by CPU-to-GPU and GPU-to-GPU synchronization, and limited by multi-GPU memory management overhead [31, 35, 58]. Design of an effective memory management system and cross-GPU communication fabric remain open problems

that need to be addressed to unlock the full potential of future multi-GPU platforms. To support design exploration of next-generation multi-GPU platforms, we need fast and accurate simulation tools and frameworks.

Existing publicly available GPU simulators, such as GPGPU-Sim [10] and Multi2Sim [50], were originally developed for single-GPU platforms and do not provide support for simulating state-of-the-art multi-GPU platforms. This is at least in part because: 1) Existing GPU simulators simulate out-dated GPU architectures. Newer GPUs add special features such as system-level atomics and GPUDirect [37] to facilitate collaborative execution across multiple GPUs; 2) Existing simulators lack modularity, which makes modeling and configuring a multi-GPU platform a tedious task; and 3) Existing simulators are not very efficient in terms of simulation speed. A few seconds of execution on real GPUs may take a few days to simulate. This issue is further exacerbated when simulating a multi-GPU platform due to the increased number of components in the modeled platform. As a result, computer architects are handicapped when studying multi-GPU platforms. Although the recent AMD GEM5 APU simulator [19] solves the first problem, a solution that delivers both high modularity in a simulator and high-performance simulations is still desired. In addition, as AMD GEM5 APU simulator focuses on APU devices, the simulator cannot easily simulate large-scale high-performance computing environment where discrete GPUs are commonly used.

We therefore present MGPUSim, a GPU simulator designed for multi-GPU platform simulation. MGPUSim faithfully simulates the AMD GCN3 ISA[2], a state-of-art and widely adopted GPU ISA. Central to MGPUSim are features including high flexibility/configurability, ease of extensibility, and multi-threading capability. High configurability enables users to easily model platforms with different instruction scheduling algorithms, memory hierarchy designs, and number of GPUs in the system. High extensibility allows researchers to add new features to the simulator without significantly modifying the simulator itself. For example, as we will show in Section 7, we extend the simulator by considering a new approach for handling remote memory accesses without modifying the existing code. Exploiting multi-threading in our simulator enables both efficient functional emulation and detailed timing simulation that can leverage the multiple hardware threads, thereby improving simulation speeds by 3.5× and 2.5× in functional emulation and detailed timing simulation, respectively. Our simulator has also been designed to produce high fidelity simulation results, differing by only 5.5% on average when compared to execution on the real AMD R9 Nano hardware.

We illustrate some of the key benefits of MGPUSim using two concrete design studies: designing a new *Locality API*, and designing a highly customized multi-GPU memory management system we call *Progressive Page Splitting Migration* (PASI). The goal of the design studies are two-fold. First, we demonstrate the capabilities of MGPUSim and how straightforward it is to model new multi-GPU platforms. Extending GPGPU-Sim or Multi2Sim to support multi-GPU simulation would be difficult and would require a major rewrite of the code base. Second, we propose and validate our software-based and hardware-based solutions to improve the scalability of multi-GPU platforms through these design studies. We observe that compared to a unified 4-GPU platform the Locality API can speed up a discrete 4-GPU platform by 1.6×, on average, and PASI can also improve the performance of the discrete 4-GPU by 2.6×, on average.

The contributions of this paper include:

- MGPUSim, a new parallel multi-GPU architectural simulator that delivers high fidelity, high flexibility, and high performance;
- Locality API, an API extension that allows explicit data and compute placement control in a multi-GPU platform without kernel modification;
- Progressive Page Splitting Migration, a hardware-based approach that allows the multiple GPUs to gradually improve the data placement at runtime.

2 BACKGROUND

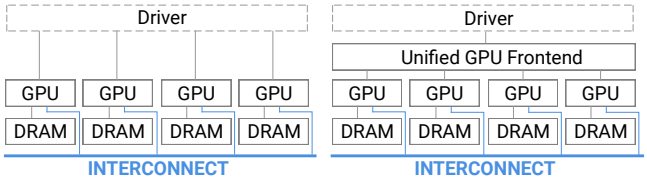
In this section, we discuss background on the four key aspects of our proposed simulator: 1) the GPU execution model that needs to be faithfully modeled by the simulator; 2) the properties of the current state-of-the-art in multi-GPU platforms that need to be supported by a multi-GPU simulator; 3) the design requirements that one should adopt when designing an architecture simulator; and 4) the mechanisms that one can use to parallelize and accelerate simulations.

2.1 GPU Execution Model

A typical GPU system is made up of one or more CPUs, and one or more GPUs (current state-of-the-art multi-GPU platforms can have up to 8 GPUs per node [18, 38]). Traditionally, the GPUs are managed by CPU. More specifically, the host program that runs on the CPU copies the data to the GPUs' memories and launches GPU programs (kernels) on GPUs. After that, the CPU copies back the computed results from GPUs' memories to system memory. A vendor-specific GPU driver, running at the operating system level on the CPU receives API calls from the host program and transfers data from/to the GPUs and launches kernels on the GPUs.

Newer GPU features give GPU more autonomy. With kernel-side enqueueing, the GPU can launch kernels to itself without the help of a CPU. Unified memory and demand paging empower the GPU to read and write system memory directly without explicit data movement between the CPU and the GPU device. In addition, GPUDirect allows the GPU to read and write the memory that is located on another GPU. Recent studies have explored enabling the GPU to request service from the CPU [47], performing system calls [51], initiating network communication [27], and enqueuing tasks on remote devices [28]. Each new feature that was not present on an earlier GPU generation requires new support from both the ISA and the microarchitecture. GPUs have continually evolved or changed the ISA at a rapid pace.

A kernel can launch a 1-, 2-, or 3-dimensional grid of work-items running on a GPU. One work-item is comparable to a thread on a CPU and has its own register state. A grid can be divided into work-groups and wavefronts. On an AMD GCN3 GPU, a wavefront consists of 64 work-items that execute the same instruction in lockstep. A work-group contains 1-8 wavefronts that can be synchronized using barriers.



(a) Discrete Multi-GPU Model. **(b) Unified Multi-GPU Model.**

Figure 1: Multi-GPU Configurations.

Current GPU design supports a very high throughput of up to ≈12.4 TFlops [39] to ≈14.7 TFlops [6]. For example, the AMD Radeon Instinct MI60 [6] GPU leverages 64 Compute Units (CU) to execute instructions in parallel. A single CU incorporates 4 Single-Instruction Multiple-Data (SIMD) units. Each SIMD unit has 16 lanes, with each lane providing a single-precision floating point unit. Hence, a single SIMD unit can execute 16 instructions in parallel in a single clock cycle. Equipped with 64 CUs, the R9 Nano GPU can execute up to $64 \times 4 \times 16 \times 2 = 8,192$ operations (fused multiply-add instructions are considered as two operations) per cycle. As the MI60 GPU runs at a 1.8GHz clock rate, it can support a peak throughput of $8,192 \times 1.8G = 14.7$ TFLOPs.

2.2 Multi-GPU Platforms

While today's GPUs are quite powerful, for many emerging applications such as computer vision analysis on videos and large-scale neural network training, a single GPU cannot meet the required processing demands due to: (1) limited compute capabilities, and (2) limited memory space. For example, VGGNet [43], a popular deep neural network framework, requires ≈ 40 Giga operations (GOps) to process a single image through a DNN model [11]. If an application requires a throughput of 1000 images per second (i.e., 40 TFlops), we need, in theory, at least 3 (40/14.7) MI60 GPUs to fulfill this requirement. On the other hand, training a DNN may require a multi-terabyte dataset [15], dwarfing the memory capacity of a single GPU. If we can increase the storage in GPU-based systems, we have the potential of allowing one kernel launch to process more data and can potentially accelerate the training process.

Multi-GPU platforms can provide both more compute resources and more memory storage. The industry has already started to explore the true potential of multi-GPU platforms [18, 24, 38]. The most commonly used GPU programming frameworks, including OpenCL [45] and CUDA [36], support multi-GPU programming following the discrete multi-GPU model shown in Figure 1a. Both programming frameworks expose all of the GPUs to users, enabling them to select where data is stored and how kernels are mapped to devices. For example, in OpenCL, a command queue is associated with a GPU and all the commands (e.g., memory copy, kernel launch) in the queue run on the associated GPU. In CUDA, a developer can select the GPU with the `cudaSetDevice` API. Exposing all GPUs to the user delivers the maximum flexibility. However, it can be difficult to adapt single-GPU applications to use a multi-GPU platform [7, 25].

The computer architecture community commonly adopts a unified multi-GPU model by hiding multiple GPUs behind a single GPU interface [7, 25, 34, 58], as shown in Figure 1b. A single kernel launch can map to all the GPUs. Therefore, the unified multi-GPU model provides better programmability, as the programmer does not need to modify the GPU program for multi-GPU platforms. However, the unified multi-GPU model may suffer from high-latency inter-GPU communication and non-scalable performance [7, 34]. With the development of new GPU features such as unified memory, demand paging, and system-level atomics, efficient programming of a multi-GPU platform, as if it were a single large GPU, is becoming a reality.

To provide a flexible multi-GPU simulation framework, MGPUSim supports both (unified and discrete) multi-GPU models, letting the user select which model to use. To the best of our knowledge, there is no prior microarchitectural research that focused on a discrete multi-GPU model due to the lack of simulator support. Even with a unified multi-GPU model, other simulators cannot achieve the same level of flexibility. We discuss this in Sections 6 and 7.

2.3 Simulator Design Requirements

An architectural simulator is a necessary tool that enables exploration of various microarchitecture design tradeoffs, performance optimizations, and design space exploration. We summarize the key simulator design choices made by past simulator designers and discuss how we satisfy these requirements in 3.1.

DR-1: Extension without modification. Users of computer architecture simulators usually need to extend the simulator to model new structures. Implementing extensions to a simulator should be simple and should not involve "heavy lifting", as major modifications not only impede the progress of researchers but also impact reproducibility.

DR-2: No magic. Simulators should avoid using "magic" that allows one component to directly access the data (i.e., class field access, getter/setter function call, or any other function call) of another component as much as possible. Using "magic" introduces inaccuracy in the overall system modeling since it ignores the communication overhead between the two components. It also compromises modularity, as the two component are coupled together, making it difficult to replace one of them (similar in nature to the reason why encapsulation is so powerful in software development).

DR-3: Tracking data with timing. Simulators should utilize real data values in the simulator, rather than allowing the emulator to "magically" access the data. Separating data flow and time calculations hides errors in system design and making value-dependent modeling difficult.

DR-4: Simulate parallel hardware in parallel. Digital circuits work in parallel. For example, a DRAM controller and a compute unit update their states simultaneously and independently. Therefore, a computer architecture simulator should also be able to update the component status with multi-threaded simulation, without introducing inaccuracy.

DR-5: No busy ticking. Simulators tend to visit each modeled component to update the state, even if such updates are not required. We should avoid unnecessary state updates to maximize performance.

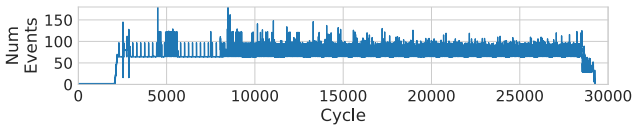


Figure 2: The number of events that can be parallelized, without interrupting the chronological order. An event is a state update of a component.

2.4 Parallel GPU Simulation

Architectural simulation is generally much slower than real hardware. For example, execution of applications on a GPU system modeled using Multi2Sim [50] is reported to be 44,000× slower than native execution, which translates to more than a day to simulate 2 seconds of native execution. Malhotra et al. [32] report that GPGPU-Sim is 480,000× slower than a real system – this translates to GPGPU-Sim needing 11 days to simulate 2 seconds of native execution. This limited simulation speed makes it impossible to simulate large-scale systems and workloads in existing simulators. To successfully simulate multi-GPUs with large-scale workloads, we need a new simulation philosophy.

Prior research has explored using multi-threading to accelerate architectural simulation. Two common approaches to parallelizing simulation are 1) conservative simulation; and 2) optimistic simulation [17]. Using a conservative approach, the chronological order of the events is not interrupted, which requires global synchronization after each cycle. An optimistic approach supports reordering events to avoid frequent synchronizations, reducing simulation time, though at the cost of fidelity of the simulation.

We elect to adopt a conservative parallel simulation approach in MGPUSim because we do not want to compromise simulation accuracy in exchange for faster simulation. Figure 2 shows the number of events scheduled at the same time during simulation of the AES benchmark using MGPUSim. The value for the minimum number of concurrent events observed is 64. This number is due to the fact that we have 64 compute units in the system. The number of events does not constantly go higher than 64 because we avoid updating the state in every cycle for the component in the cache system, interconnects, and the memory controllers to improve performance. Overall, this number of events varies between 60 and 100 for most of the time, providing sufficient parallelism to keep a 4- to 8-core system busy.

3 MGPUSIM

MGPUSim is a highly-configurable GPU simulator that is open-source [1] under the terms of the MIT license [40]. We have developed the simulator using the Go programming language [48]. We selected Go because Go provides both reasonable performance [55] and ease of programmability. It also provides native language-level support for multi-threaded programming, allowing us to spawn a large number of Goroutines (i.e., light-weight threads) [48] to process events with very low overhead.

[1] The source code and the issue tracker are available at: https://gitlab.com/akita/gcn3

3.1 Simulator Framework

Central to our design is the simulator framework. We embrace a domain-agnostic design approach so that the framework can be used to model any component such as a different GPU model, a CPU, or an accelerator device. Our framework consists of the following four parts:

1. The Event-Driven Simulation Engine: We define an event as a state update of a component. Our event-driven simulation engine maintains a queue of events for the whole simulation and triggers events in chronological order.

2. Components: Every entity of a multi-GPU platform that MGPUSim simulates is a component. In our case, a GPU, a CU, and a cache module are examples of components.

3. Connections: Two components can only communicate with each other through connections using requests. Connections are also used to model the intra-chip interconnect network and inter-chip interconnect network.

4. Hooks: Hooks are small pieces of software that can be attached to the simulator to either read the simulator state or update the simulator state. The event-driven simulation engine, all the components, and the connections are hookable. Hooks can perform non-critical tasks such as collecting execution traces, dumping debugging information, calculating performance metrics, recording reasons for stalls, and injecting faults (for reliability studies).

The MGPUSim event engine supports parallel simulation, fulfilling *DR-4*. Leveraging the fact that the events that are scheduled at the same time do not depend on each other, the event-driven simulation engine harnesses multiple CPU threads to process events. We embrace a conservative parallel event-driven scheme (see Section 2.4), so that we guarantee the simulation results will match a serial simulation.

The component system and the request-connection system enforce strict encapsulation of components. We restrict a component from scheduling events for other components, and at the same time, we do not allow a component to access another component's state (by reading/writing field values, using getter/setter functions or function calls). All communication must use the request-connection system. This design choice forces the developer to explicitly declare protocols between components. The benefits of this design are three-fold. *First*, a developer can implement a component, considering only the communication protocol. *Second*, we gain flexibility since we can replace a component with any other component following the same protocol. As we will see in Section 7, extending the simulator just involves adding a new component that implements the same protocol and wiring the new component with other components. By adopting this model, we fulfill the requirement of *DR-1*. *Third*, we can improve simulation accuracy as no information can "magically" flow from one component to another, without being explicitly transferred through the interconnect. Therefore, we can satisfy both *DR-2* and *DR-3* with our design approach.

The event-driven simulation and the connection system can help avoid busy ticking (*DR-5*). For example, a DRAM controller may be able to calculate that a request takes 300 cycles to complete when the request arrives, and nothing needs to be modeled in detail during the 300 cycles. So the DRAM controller can schedule an event in the event-driven simulation engine after 300 cycles and skip

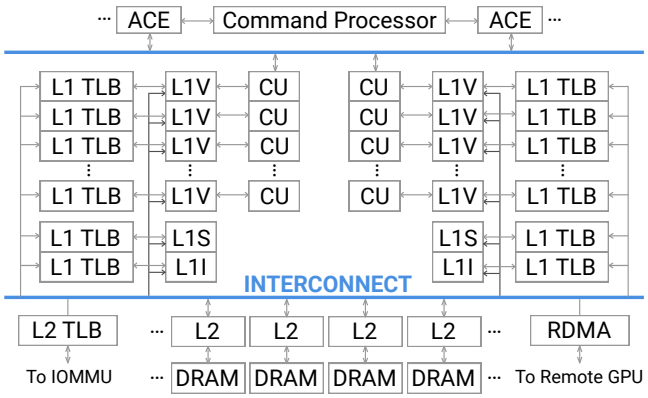


Figure 3: The Modeled GPU Architecture.

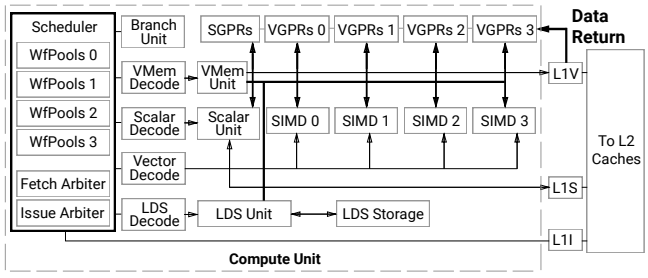


Figure 4: The Compute Unit Model.

state updates until then. In addition, another type of busy ticking in GPU architectures is caused by components that repeatedly try to send data. Since a component has no information about when a connected connection becomes available, the component has to retry each cycle. To avoid this type of busy ticking, we allow the connections to explicitly notify connected components when the connection is available. Therefore, a component can avoid updating the state if all of the out-going connections are busy, and update its state after a connection is available.

3.2 GPU Modeling

MGPUSim's GPU model, as shown in Figure 3, faithfully supports the Graphics Core Next 3 (GCN3) ISA. We configure the model of the GPU according to the publicly available AMD documentations and through microbenchmarking. While the latest ISA on AMD Vega GPU's runs GCN5 [4], GCN5 only extends the memory access instructions. The compute instructions in GCN5 are the same as for GCN3. Simulating GCN5 can be achieved in MGPUSim by just adding support for the new memory access instructions. We do not need to change the remaining core components of MGPUSim.

The GPU architecture is composed of a Command Processor (CP), Asynchronous Compute Engines (ACEs), Compute Units (CUs), caches, and memory controllers. The CP is responsible for communicating with the GPU driver and starting kernels with the help of ACEs. The ACEs dispatch wavefronts of kernels to run on the CUs.

In our model, a CU (as shown in Figure 4) incorporates a scheduler, a set of decoders, a set of execution units, and a set of storage units. The CU includes a scalar register file (SGPRs), vector register files (VGPRs), and a local data share (LDS) storage. A fetch arbiter and an issue arbiter decide which wavefront can fetch instructions and issue instructions, respectively. Decoders require 1 cycle to decode each instruction, before sending the instruction to the execution unit (e.g., SIMD unit). Each execution unit has a pipelined design that includes read, execute, and write stages.

MGPUSim includes a set of cache controllers, including a write-through cache, a write-around cache, a write-back cache, and a memory controller. By default, the L1 caches and the L2 caches use a write-through and write-back policy, respectively. The cache controllers do not enforce coherence as the GPU memory model is fully relaxed. The compute units send virtual addresses for read and write requests to the L1 cache. Virtual addresses are translated to physical addresses at L1 cache with the help of two levels of TLBs. We show the default configuration in Figure 3. However, both the number of layers of caching and the number of layers of TLB are fully configurable. Finally, we equip each GPU with a Remote Direct Memory Access (RDMA) engine to manage the inter-GPU communication.

3.3 Simulator APIs

MGPUSim can run in two different modes, native mode and Go mode. In native mode, we provide a customized implementation of the OpenCL runtime library in the C programming language. Users can link the MGPUSim-provided OpenCL library with the workload executables so that the customized OpenCL library can redirect the API calls to MGPUSim and run the GPU kernels on the simulated GPUs. In Go mode, we allow user to write a main program in Go to define memory layout and launch kernels.

MGPUSim's GPU driver provides a set of OpenCL-like APIs to allow workloads to control the simulated GPUs in Go mode. Each user workload should start by calling an `Init` function to create an execution context for the following API calls. Then, the workload can invoke device discovery functions and use the `SelectGPU` function to specify the GPU to use. Finally, the main body of the workload can be implemented by using memory allocation, memory copy and kernel launch APIs. Since the APIs are similar to OpenCL, an experienced OpenCL programmer should feel very comfortable when using the MGPUSim APIs. In addition, we let each workload, the driver, and the simulation each are run in individual threads, allowing multiple workloads to run in parallel in the simulator.

4 METHODOLOGY

In this section, we describe the simulation configurations, the full set of microbenchmarks and full benchmarks that we utilize for validation and evaluation of the design studies. For validation of MGPUSim, we compare the execution of microbenchmarks and full benchmarks against a multi-GPU hardware platform that has 2 AMD R9 Nano GPUs (Section 5). We also use these workloads to evaluate the benefits of new multi-GPU features in two case studies (discussed in Sections 6 and 7), demonstrating the utility and potential of MGPUSim for multi-GPU research.

Parameter	Property	# per GPU
CU	1.0 GHz	64
L1 Vector Cache	16KB 4-way	64
L1 Inst Cache	32KB 4-way	16
L1 Scalar Cache	16KB 4-way	16
L2 Cache	256KB 16-way	8
DRAM	512MB	8
L1 TLB	1 set, 32-way	96
L2 TLB	32 sets, 32-way	8
IOMMU	shared by all GPUs	-
Intra-GPU Network	Single-stage XBar	1
Inter-GPU Network	PCIe-v3 16GB/s	-

Table 1: Specifications of the Modeled R9 Nano GPU.

4.1 Simulation Configuration

We validate MGPUSim against a multi-GPU platform with 2 Intel Xeon E2560 v4 CPUs and 2 AMD R9 Nano GPUs (details provided in Table 1) using execution time as the validation metric. The CPUs and the GPUs are connected via a 16GB/s PCIe 3.0 interconnect. The system runs the Radeon Open Compute Platform (ROCm) 1.7 software stack on a Linux Ubuntu 16.04.4 operating system. We lock the GPUs to run at the maximum frequency to avoid the impact of Dynamic Voltage and Frequency Scaling (DVFS). All the kernels are compiled with official ROCm compiler. All the timing results are collected using the Radeon Compute Profiler [5]. All the experiments in this paper are performed in the Go mode.

We evaluate the simulator speed and multi-threaded scalability using a host platform with a 4-core Intel Core i7-4770 CPU. We use the environment variable `GOMAXPROCS` to set the number of CPU cores that the simulator can use.

4.2 Microbenchmarks

To fine-tune the GPU model in MGPUSim, we develop a set of 57 microbenchmarks that cover a wide range of instruction types and memory access patterns. Each microbenchmark is composed of a manually written or script-generated GCN3 assembly kernel, a C++ host program used in native execution, and an additional host program written in Go for simulation. For the sake of brevity, out of the 57 microbenchmarks used in this paper, below we discuss four microbenchmarks that serve as a good representative of the complete set:

ALU-focused microbenchmark: This Python-generated microbenchmark generates kernels with a varying number of ALU operations (`v_add_f32 v3, v2, v1`) followed by an `s_endpgm` instruction to terminate the kernel. Using the ALU microbenchmark, we validate instruction scheduling, instruction pipeline, and instruction caches.

L1 Access-focused microbenchmark: This microbenchmark generates a varying number of memory reads to the same address. All accesses, except for the first one are L1 cache hits, which allows us to measure the cache latency.

DRAM Access-focused microbenchmark: This microbenchmark repeatedly accesses the GPU DRAM using a 64-byte stride. Since all cache levels use 64-byte blocks, all accesses are expected to incur both L1 and L2 cache misses, and ultimately read from the DRAM. We use this microbenchmark to measure the DRAM latency.

L2 Access-focused microbenchmark: This microbenchmark first reads each cache line in a 1MB block of memory, loading the whole 1MB into the L2 cache. The L1 cache is expected to retain the last 16KB, which is equal in size to its total capacity. After this, a second scan sweeps the same 1MB of data from the beginning, causing L1 misses and L2 hits. We use this strategy to find the L2 cache latency.

4.3 Full benchmarks

Out of the wide variety of full benchmarks available in the AMD APP SDK [3] and the Hetero-Mark [46] suite, we select a set of representative benchmarks (listed in Table 2) for both simulator validation and our 2 case studies. We select these benchmarks to ensure a wide coverage on the inter-GPU memory access patterns. We modify the benchmarks to run on multi-GPU platforms. For validation experiments, we duplicate the workload to run on two GPUs, while for the design studies, the workloads remain the same, and we dispatch portions of the workload to each individual GPU. We use a different approach during the design studies versus the validation experiments, because we want to focus primarily on multi-GPU collaboration in the design studies. Therefore, we let the multiple GPUs work on a single set of data.

5 SIMULATOR VALIDATION

In this section, we discuss the results of validating MGPUSim against real hardware. For emulation results, by running MGPUSim in either functional emulation mode or timing simulation mode, we are able to compare the simulation results with the results on AMD GPU hardware at bit-level granularity. This comparison enables us to build confidence in the correctness of instruction emulation and memory consistency in our simulator. For timing simulation accuracy, we show the results of running microbenchmarks and full-benchmarks in the following subsections.

Microbenchmark Validation: Figure 5 shows a comparison of the execution time of the different microbenchmarks discussed in Section 4 when runing on an R9 Nano GPU and in MGPUSim. MGPUSim achieves very high accuracy when running these microbenchmarks. For the L1 Vector Cache, L2 Cache and DRAM microbenchmarks, the two curves overlap indicating the high accuracy of our simulator. In the ALU benchmark, there is an offset of several microseconds between the two lines, which is introduced by random DRAM refreshes. We also validate our simulator with microbenchmarks that test other important components such as the ACEs, the L1 constant cache, and the TLBs. Since these experiments all produce similar results, as the simulator estimated execution time curves fully overlap or track closely with the real-GPU execution time curve, they are not included here. In light of all our simulation results using the microbenchmarks, we can confirm that MGPUSim can model the key GPU components with high fidelity.

Full-benchmark validation: Next, we validate our simulator with full benchmarks running on 2 R9 Nano GPUs. Figure 6 shows a comparison of the simulator estimated execution time and the real hardware execution time. The difference between the two values

Abbr.	Workload	Multi-GPU Memory Access Pattern
AES	AES-256 Encryption	**Partition**: Each GPU works on its own batch of data. No inter-GPU communication is needed.
BS	Bitonic Sort	**Random**: Any GPU can read/write from/to any other GPU. Memory access patterns are different from kernel to kernel.
FIR	FIR Filter	**Adjacent Access**: The input array is equally divided into batches for each GPU. The filter data, which is small, is duplicated to each GPU. The calculation on each GPU needs to access a small portion of data close to the batch division from another GPU.
KM	KMeans Clustering	**Partition**: KMeans contains two kernels, one matrix transpose and one calculates the distance from each input point to the cluster centroids. In the second kernel, each GPU works on its own batch of data. We have frequent CPU-GPU communication in this benchmark.
MT	Matrix Transpose	**Local-Read Remote-Write**: Each GPU reads data from their local DRAM, but writes data to remote GPUs' DRAM.
MM	Matrix Multiplication	**Remote-Read Local-Write**: Each GPU reads data from local and remote GPUs, but only writes data to local DRAMs.
SC	Simple Convolution	**Adjacent Access**: The input image is divided into sub-images and copied to each GPU. Each GPU needs to access some of the pixels that are copied to another GPU.

Table 2: Full Benchmarks and their multi-GPU memory access patterns.

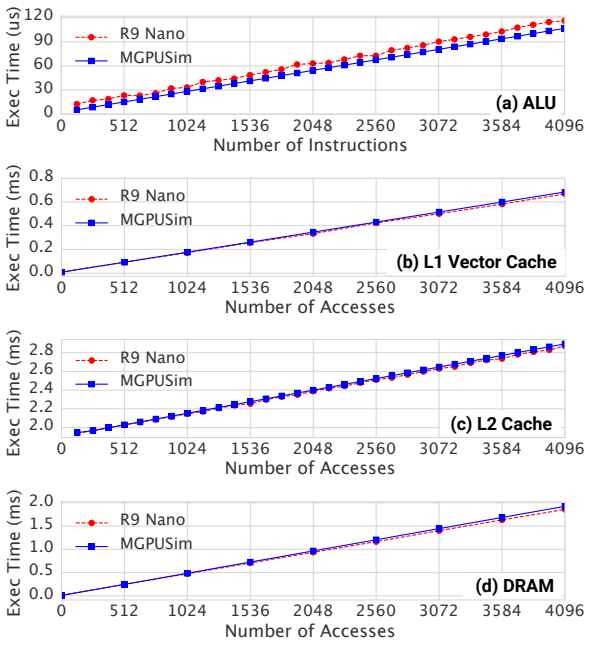


Figure 5: Simulator Validation with Microbenchmarks.

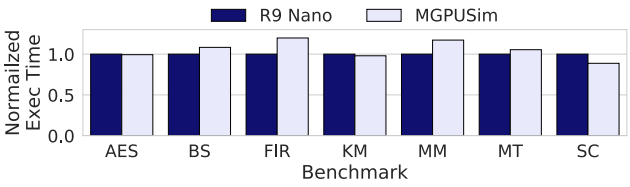


Figure 6: Execution time comparison between R9 Nano and MGPUSim for the benchmarks listed in Table 2.

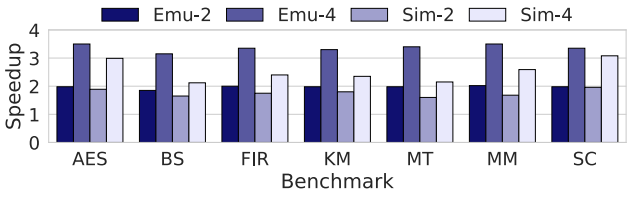


Figure 7: Speedup of Functional Emulation (Emu-), and Detailed Timing Simulation (Sim-) using 2 and 4 CPU Threads.

across all benchmarks has an average value of 5.5% (peak value of ≈20% in FIR and SC benchmarks). After a comprehensive study, we can confirm the differences are mainly due to undocumented GPU hardware details. Although we try to model every individual GPU component, we are not able to capture all the hardware implementation details, such as subtle pipeline structures in the cache modules and sizes of the network buffers.

Parallel Simulation Performance: To compare MGPUSim with Multi2Sim 5.0 and GPGPU-Sim, we run all three simulators configured with a single-GPU running the MT benchmark on an Intel Core i7-4770 CPU. Our experiment reveals that our simulator can reach ≈ 27 Kilo-instruction per second (KIPS) with 4 CPU threads. For Multi2Sim 5.0 and GPGPU-Sim, we obtain a simulation throughput of ≈ 1.6 KIPS and ≈ 0.8 KIPS, respectively. MGPUSim is 16.5× and 33.8× faster than Multi2Sim 5.0 and GPGPU-Sim, respectively.

To support efficient design-space exploration in the context of multi-GPU platforms, unlike contemporary GPU simulators, we designed MGPUSim with built-in multi-threaded execution to further accelerate the speed of simulations. Our simulations can take advantage of the multi-threaded/multi-core capabilities of contemporary CPU platforms. As shown in Figure 7, MGPUSim achieves good scalability when using multiple threads to run simulations. In particular, when 4 cores are used in the Intel Core i7-4770 CPU

platform, MGPUSim can achieve 3.5× and 2.5× speedups in functional emulation and architectural simulation, respectively, while preserving the same level of accuracy as in single-threaded simulation. In addition, our parallelization approach is domain-agnostic, allowing the parallelization approach to remain valid as we extend the features of the simulator.

6 DESIGN STUDY 1: LOCALITY API

Next, we use two design studies to illustrate the use of MGPUSim. Modeling any of the designs with existing open-source GPU simulators can be very difficult due to the lack of multi-GPU support.

As discussed in Section 2.2, the unified multi-GPU model and the discrete multi-GPU model each have unique advantages and disadvantages. The unified multi-GPU model has a simpler programming model, while the discrete multi-GPU model allows for precise control of data and compute placement. The Locality API finds the middle ground of the two approaches by adding a runtime extension to the unified multi-GPU model. Using the Locality API, a programmer can either treat multiple GPUs as a single large GPU (i.e., a *UGPU*) or as individual GPUs (i.e., an *IGPU*). In addition, as the Locality API is a runtime API extension, a similar set of API extensions can be implemented in any GPU programming frameworks, such as OpenCL, CUDA, or HSA [20].

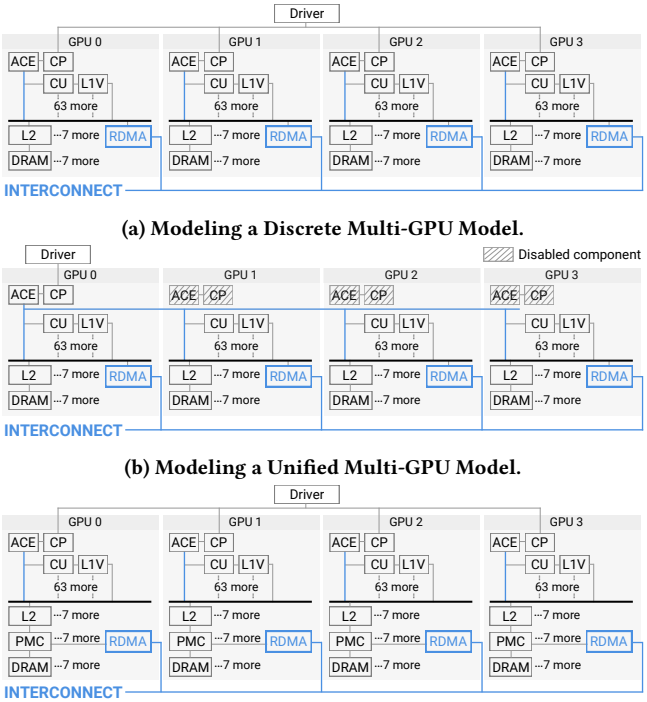
6.1 API Design

The Locality API is based on the observation that a large portion of regular GPU workloads has a regular and predictable memory access pattern. It is common that GPU programmers know exactly what data is accessed by each work-item. In this case, the programmer can utilize algorithm-specific knowledge to ensure most memory accesses reference the local GPU and avoid costly inter-GPU communication. Since this knowledge is algorithm specific, it is very difficult for a pure hardware-based solution to achieve the same level of optimization.

Our Locality API includes a group of three APIs: 1) an extended GPU Discovery API, 2) a Memory Placement API, and 3) a Compute Placement API.

The **extended GPU Discovery API** allows the host program to discover both the UGPU and each IGPU. We assume that there is a memory region associated with each IGPU, so accessing the associated memory is much faster than accessing a remote memory that belongs to another IGPU.

Memory Placement API allows the programmer to explicitly map a range of memory to an IGPU. Since the OS, the Memory Management Unit (MMU), and Input Output Memory Management Unit (IOMMU) manage the Page Table to keep track of both the virtual and physical addresses of pages (a page is usually a contiguous 4KB memory space), we use the GPU driver to modify the page table to map the specified range of virtual addresses to physical addresses on the target IGPU. For example, assuming that we have four IGPUs and each has a 1GB memory space, the physical address space is banked into ranges 0 — 1GB, 1 — 2GB, etc, such that a 16KB vector that has a virtual address of 0 — 16KB can map to a physical address of 0 — 0+4KB, 1GB — 1GB+4KB, etc. If we have a vector-add application, we can simply launch the work-groups that work on the first 4KB to GPU 0 and the wavefronts that work on



(a) Modeling a Discrete Multi-GPU Model.

(b) Modeling a Unified Multi-GPU Model.

(c) Modeling a Multi-GPU Platform with Page-Migration.

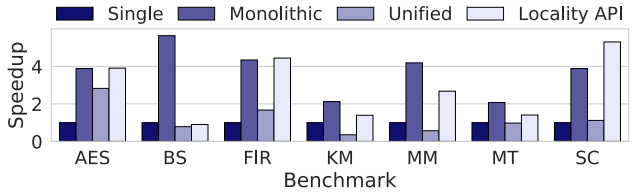
Figure 8: Modeling Different Multi-GPU Configurations.

the second 4KB to GPU 1, and a similar pattern for GPU2 and GPU3, completely avoiding inter-GPU communication. On a typical 4KB-page virtual memory system, we require that a full (versus partial) page is mapped to a specific GPU to guarantee the correctness of address translation.

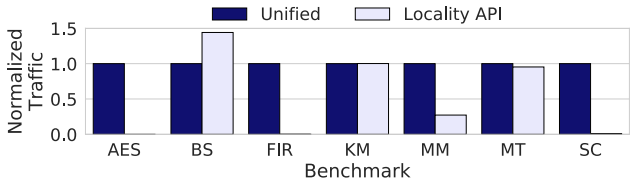
The **Compute Placement API** extends the existing kernel-launching API by allowing programmers to specify the IGPU ID to launch the kernel and the list of work-groups to execute on the submodule. Rather than launching a kernel with a single API call, programmers may need a loop to launch the sub-kernels on each IGPU.

6.2 Simulator Implementation

Modeling discrete multiple GPU platforms is straightforward with MGPUSim, and it is supported out-of-the-box by the simulator. The `BuildNR9NanoPlatforms` function is provided by the framework to build a multi-GPU platform with the desired number of GPUs. More specifically, this function returns an array of GPUs and a driver object that provides information about the GPUs. Internally, the function instantiates multiple GPU objects and configures their internal components and parameters. The function also instantiates a PCIe bus (the PCIe can be replaced with other interconnect fabrics) to connect the RDMA engine to each GPU that is used for GPU-to-GPU memory communication. We present the modeled discrete multi-GPU platform in Figure 8a. Note that instantiating multiple GPU instances is not possible in either GPGPU-Sim or Multi2Sim, due to the heavy coupling between classes.



(a) Locality API Speedup.



(b) Normalized Inter-GPU Traffic.

Figure 9: Locality API Performance.

For a unified multi-GPU model, we need to rewire the GPUs created by `BuildNR9NanoPlatforms` (see Figure 8b). Since we use the ACE of GPU0 to dispatch work-groups to the compute units of all the GPUs, we need to add an extra connection to wire the ACE of GPU0 to all the compute units in the platform, as well as register all the CUs in the ACE. We need to keep the connections between the compute units and the cache modules, the connections between cache modules and memory controllers, and inter-GPU RDMA-RDMA connections unchanged. We accomplish this rewiring of the multi-GPU design, using less than 10 lines of code.

Finally, to implement the Locality API in MGPUSim, we replace the driver component to support the extra driver API functions described in Section 6.1. The new driver component wraps the standard driver to avoid re-implementing the existing driver APIs. Since we define a communication protocol between the driver and the GPU, and between the driver and the MMU, replacement is easy and straightforward.

6.3 Evaluation

To evaluate our new Locality API, we run full benchmarks on four different configurations. We use both single GPU and a monolithic multi-GPU configuration as baselines for comparison. The monolithic GPU is built by integrating the resources of 4 GPUs (CUs, cache modules, memory controllers) into one chip. Note that the monolithic GPU is impractical to build as it requires a large die size $> 2000mm^2$, since each R9 Nano requires a die size of $596mm^2$ [1]. We also compare against a unified 4-GPU configuration, without the Locality API enabled, as another baseline design. The Locality API configuration is based on the same unified 4-GPU configuration, but allows us to apply a locality-based optimization to avoid inter-GPU communication. Using the locality API is equivalent to custom programming for each individual GPU, and therefore, it achieves the same performance as a discrete multi-GPU model.

The inter-GPU traffic, as shown in Figure 9b, can be significantly reduced when the Locality API is used. We see that in benchmarks such as AES, as the programmer can perfectly partition the data, with inter-GPU communication can be fully eliminated. For benchmarks that follow the Adjacent Access pattern (e.g., FIR and SC), the inter-GPU traffic can also be minimized. However, in benchmarks such as MT and BS, manual optimization is not easy to apply, and hence, the Locality API cannot reduce traffic, and sometimes may even introduce more inter-GPU traffic.

In terms of the execution time (see Figure 9a), we observe that the performance of a monolithic GPU generally scales well and can sometimes even provide speedups of more than 4×. This superlinear speedup is due to reduced bank conflicts as we add more L2 cache modules and memory controllers. In benchmarks such as KM and MT, due to an inherent lack of parallelization, a monolithic GPU can only speed up execution by 2×.

The benchmark execution time of the Unified and the Locality-API configuration are correlated with the inter-GPU traffic, indicating that inter-GPU communication is a major bottleneck in the system. We see that in many cases (e.g., AES and FIR), the locality API can nearly obtain the same level of scalability as a monolithic GPU, while a Unified configuration is not able to run as fast as a single-GPU design. In FIR and SC, the Locality API can even outperform a Monolithic GPU. This is because the monolithic GPU has a large network connecting the L1 to L2 caches, and so it is more likely to have congestion in the network and the input buffers of the L2 caches. In most of the other benchmarks (AES, KM, and MM), the Locality API can easily improve the unified multi-GPU model. As a special case here, the AES benchmark shows relatively good scalability on all configurations due to the data-parallel compute-intensive nature of the workload. Finally, the Locality API cannot speed up the MT and BS benchmarks, since we cannot easily split the data on each IGPU.

7 DESIGN STUDY 2: PASI

The Locality API allows programmers to apply their domain-specific knowledge to avoid inter-GPU communication. However, in many cases, allocation of the data properly to each GPU is a difficult task. Also, when a single-GPU application is directly migrated to a multi-GPU platform, before it can be manually optimized, we need a scalable solution.

To allow hardware to help with improving data locality, we propose using *Progressive Page Splitting Migration (PASI)*. The goal of PASI is to enable the GPU hardware to automatically improve the data placement for any workload. We explain the design of PASI in the next 3 subsections.

7.1 Page Migration

When describing Locality API, we assumed that the RDMA engine lies between the L1 and the L2 caches as shown in Figures 8a and 8b. In the case of an L1 miss, depending on the requested address, the L1 cache will either send the request to a local L2 cache or to the RDMA engine. This Direct Cache Access (DCA) design forces all inter-GPU communication packets to have a payload that is smaller than or equal to the cache line size (64B in typical systems), resulting in poor utilization of the interconnect bandwidth and spatial locality.

To address this issue, we employ *Page Migration* by rearchitecting the system as shown in Figure 8c. A Page Migration Controller (PMC) is integrated in each memory controller. A PMC has its internal page directory stored in the GPU DRAM directly. It introduces a very small amount of extra storage. Assuming a 4GB GPU DRAM and a 4KB page size, PMC comprises at most 1M entries to store the tag data (0.2% overhead assuming each entry is 8B). Each entry contains the physical address tag of the page, and a single valid bit, indicating if the page is mapped to current GPU.

In the case of an L2 miss, a memory access request arrives at the PMC. The PMC checks its internal directory to determine whether the page is currently in the local DRAM. If the page is not present in the local DRAM, it communicates with the RDMA engine to send a page migration request to the input-output memory management unit (IOMMU), which is a hardware component located on the CPU. The IOMMU also maintains a table that tracks which page is located on which GPU. The IOMMU identifies the GPU that owns the data and forwards the migration request to the destination GPU. The owning GPU sends the page data to the requester GPU and then marks the page as invalid on the receiver GPU's local PMC. The page invalidation is also followed by a TLB shootdown (invalidating the page in the TLBs) on the receiver to avoid a translation error. Since the IOMMU knows both the source and destination of the page migration request, it updates its internal directory to reflect the migration.

7.2 Cache-only Memory Architecture

Page-migration can help improve utilization of the inter-GPU interconnect by increasing the network packet size and can also increase spatial locality. However, due to the fact that multiple GPUs may share the same data, a single page may ping-pong back and forth between GPUs. This can significantly impact workloads such as matrix multiplication, as all pages containing the input data are accessed by all of the GPUs to calculate the output.

In general-purpose GPU computing, memory access patterns of data items can be categorized into 4 types: 1) Single Read; 2) Multiple Read; 3) Single Write; 4) Multiple Write. Types 1) and 3) are commonly seen in streaming workloads and can be fairly easily resolved by the Locality API and Page Migration approaches. However, page migration does not help address issues with access pattern types 2) and 4). Therefore, we use a *Cache-only multi-GPU memory architecture* and *Page Splitting* to solve type 2) and 4), respectively.

To allow the same piece of data to be shared by multiple GPUs, we extend the page migration approach with a memory coherency protocol to unify the multi-GPU system as a cache-only memory system. The concept of a cache-only memory architecture (COMA) [49] describes memory systems that are only composed of cache modules. In COMA, there is no "root" node in the system to always maintain a copy of all the data. In COMA, any piece of data is stored in at least one GPU. But the exact location depends on which GPU uses the data. In contrast to a page-migration approach, a piece of data is allowed to reside in multiple locations with the support of a memory coherency protocol. When the GPU memory is full because of page sharing, we spill the data to system memory, under the control of the IOMMU.

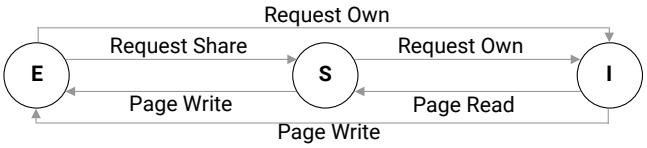


Figure 10: The ESI memory coherency protocol for multi-GPU cache-only memory architecture.

We introduce a light-weight ESI memory coherency protocol, similar to the standard MSI protocol [44], to manage the multi-GPU COMA, where E, S, and I stand for Exclusive, Shared, and Invalid states, respectively. As M in the MSI emphasizes the dirtiness of a cacheline, and the concept of dirtiness does not exist in a cache-only system, we use E to emphasize the write exclusiveness of a page.

Note that this coherency protocol works at a page granularity rather than a cacheline granularity, and only manages a subsystem that is composed of the memory controllers. It is independent of the cache system. The cache system can still apply cache coherency protocols. The memory controllers of the cache-only memory system can collectively be treated as the root node for the cache system since any piece of data is always available in at least one memory controller.

The ESI coherency protocol works as follows. Assuming a page starts with an "Invalid" (I) state, when an L2 cache reads the data from the page, the PMC requests the data to be migrated, as we described in 7.1. The IOMMU maintains a memory coherency directory and checks which GPU owns the data. In case only one GPU owns the data in an "Exclusive" (E) state, the IOMMU requests the owning GPU to send the data to the requesting GPU, while marking the state of the page on each GPU as "Shared" (S) state. On the other hand, if multiple GPUs own the data in the "S" state, the IOMMU will select one of the data owners to send the data to the requesting GPU. The selection algorithm is configurable according to the interconnect topology. In our case, as we use a bus to connect multiple GPUs, we let the IOMMU to randomly select an owner to send the data.

The processor writes take a similar approach. When a processor requests to write to a page that is currently in states I or S, the PMC requests page migration from the IOMMU. The IOMMU invalidates the page from all other owners. The page will also change to the "E" state, as it has acquired exclusively and is ready to be written into.

7.3 Page splitting

The Cache-only System can avoid useless page-migration when multiple GPUs read from the same piece of data (i.e., the "2) Multiple Read case" described earlier). However, when the same page needs to be written by different GPUs (i.e., the "4) Multiple Write" case), a page still needs to be migrated due to the requirement of exclusiveness of writing. In general, different GPUs should not write into the same address unless a system-level atomic write is used. Writing into different parts of the same page from different GPUs (i.e., false sharing) triggers unnecessary page migration, and should be avoided.

Decreasing the page size is a solution to avoid false sharing. However, smaller pages reduce the coverage of the TLBs and may

potentially increase the address translation latency, causing the compute unit to stall. In general, there is no single page size that fits all applications, and even for the same application, different kernels that are part of the application can have a bias towards a particular page size. Therefore, we need a dynamic approach, allowing the hardware to find the best page size during execution.

We use a *Page Splitting* approach, built upon a cache-only memory system and the ESI protocol. Starting from a large page size (e.g., 2MB), when a page needs to transit from state "E" to state "I" or "S", rather than migrating the whole page, we split the page in half and only transfer the requested half of the page. We allow the page to continue to be split down until the smallest page size (e.g., 1KB) is reached. Since we split the page in half, each half becomes a page that has an ESI state and an owner list. The IOMMU and the TLB also need to keep track of the page size to guarantee translation accuracy. In addition, whenever adjacent pages arrive at one GPU, the IOMMU merges them into a larger page.

7.4 Simulator Implementation

With the Locality API, we demonstrate how to reconfigure the driver and connect it to the GPUs. Evaluating PASI serves as a good example of how to change the behavior or the memory system and how multiple GPUs interact.

To implement page migration, we create a new component named the Page Migration Controller (PMC). The PMC has three ports, including the `L2CachePort`, `DRAMPort`, and the `RDMAPort`, which are responsible for handling the communication between the PMC and its connected components. We also add two types of requests, `PageMigrationReq` and `PageMigrationRsp`, which represent both the initial request and the response from a peer. We enable the PMC process read and write requests from the L2 cache, the data ready and write responses from the memory controller, and the `PageMigrationReqs`/`PageMigrationRsps` that are forwarded by the RDMA engine and originated from a PMC of another GPU.

Rewiring is necessary to integrate the new PMC with the existing GPU configuration. After instantiating the PMC, we connect the L2 cache's `DRAMPort` port with the PMC's `L2CachePort`, so that they can communicate. We also connect the RDMA and the memory controller with the PMC. Since we design an explicit protocol for the L2 cache, the memory controller, and the RDMA, the PMC can be inserted as long as it follows the protocols. The L2 cache, the RDMA, and the memory controller are fully unaware of their connection to the PMC. We implement 3 different PMCs by adding features incrementally, including a Page Migration Only PMC (PM), an ESI PMC (ESI), and a Page Splitting PMC (PS). Given the level of modularity that MGPUSim delivers, we can freely swap in the desired type of PMC.

7.5 Evaluation

According to the evaluation results shown in Figure 11, Page Migration alone supports scalability on a unified 4-GPU platform to run the AES, KM, and MM benchmarks. When simulating 4 GPUs, we can achieve a speedup of 3.3×, 3.1×, and 2.7×, as compared to a single GPU execution, when running the AES, KM, and MM benchmarks, respectively, while DCA only achieves a speedup of 2.8× for AES, and a slowdown of 0.35× and 0.56× for KM and MM,

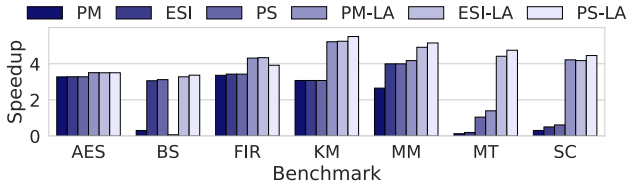


Figure 11: The speedup of PASI on a 4-GPU platform over a single GPU with incrementally added features. Here, PM = Page Migration, PS = Page Splitting, and LA indicates that the Locality-API is used.

respectively. However, in some cases, such as BS, MT, and SC, we see that Page Migration alone slows down the execution up to 4×, as compared to single-GPU execution, mainly because of read-only sharing and false sharing. In these cases, a page cannot reside stably on a single GPU, since ping-ponging between GPUs will occur frequently.

The ESI bars in the Figure 11 show the speed up of an approach that combines the ESI coherency protocol and Page Migration. As the combination of these mechanisms can allow the read-only memory to reside in multiple GPUs, a page only needs to be migrated once to each GPU. We observe the effect of ESI in the BS and MM benchmarks, as their inputs are usually shared by multiple GPUs. However, we also notice that ESI and Page Migration are not able to effectively improve MT and SC performance.

Finally, as we integrate Page Splitting with ESI and Page Migration (the PS and PS-LA bars in Figure 11), we improve performance by up to 4× (in MT), compared to the ESI + Page Migration approach. This is mainly because: 1) a larger initial page size improves TLB coverage; 2) the initial migration takes more time at the beginning of the execution, but we can avoid future small page migrations, and thus reduce the wait time of the ALU pipelines; and 3) for applications that have false sharing, Page Splitting can migrate smaller pages and reduce the inter-GPU traffic. Overall, we see that PASI can improve the performance of a unified 4-GPU platform by 2.65× compared to the DCA approach.

8 RELATED WORK

GPU Simulators: Ever since GPUs were used in the domain of high-performance general-purpose computing, researchers have developed GPU architectural simulators to support the research community to perform GPU architecture exploration. GPGPU-Sim [10] and Multi2Sim [50] are the most popular out of a number of publicly available GPU simulators that model GPUs based on NVIDIA's PTX ISA and AMD's GCN1 ISA, respectively. The AMD GEM5 APU [19] is a recent GPU simulator focused on APU devices developed in parallel with MGPUSim, and is also capable of simulating the GCN3 ISA. While MGPUSim is inspired by these predecessor simulators, MGPUSim emphasizes strong software engineering principles, high-performance parallel simulation, and multi-GPU platform modeling.

Parallel GPU simulators: To accelerate GPU simulation, parallel GPU simulators have been proposed [14, 29, 30, 32]. Barra [14] mainly focuses on parallel functional emulation, while MGPUSim

performs both emulation and timing simulation. GPUTejas [32] is a Java-based, trace-driven, parallel architectural simulator that can achieve high performance and scalability. Instead of trace-driven, MGPUSim is execution-driven in order to support the "no-magic" and "track data with timing" design requirements. The parallel simulator framework proposed by Lee et at. [29, 30] modifies GPGPU-Sim and only synchronizes when the processor accesses the memory system. MAFIA [23] is a multithreaded simulator for running concurrent kernels on GPGPU-Sim. However, in contrast to MGPU-Sim, the simulation of a single kernel is not parallelized by MAFIA. Therefore a single kernel executing in this framework will roughly take the same amount of time as it does in GPGPU-Sim. Different from GPUTejas and Lee et al.'s frameworks, we achieve scalable speedup without compromising simulation accuracy.

Multi-GPU Microarchitecture Research: Recently, the community has started to study how to efficiently accelerate computing with multi-GPU platforms. Given that the inter-GPU network and the supporting memory system inhibit scalability in current multi-GPU platforms, research has focused on optimizing memory organization. Kim et al. propose Scalable Kernel Execution [25], allowing a single kernel to execute on multiple GPUs as if there is only one GPU on the platform. Ziabari et al. [58] propose a unified memory hierarchy (UMH) and NMOESI, using the large GPU DRAMs as cache units for the system memory, achieving CPU and multi-GPU memory coherency. MCM-GPU [7] considers a multi-chip module that encapsulates multiple GPUs in the same package. They introduced an L1.5 cache and used memory affinity scheduling to reduce the cross-GPU traffic. A NUMA-aware multi-GPU system, proposed by Milic et al. [34],also tries to reduce traffic on the interconnect. Young et al. [57] proposed CARVES, which uses a combined hardware and software approach to allow multiple GPUs to share one piece of memory. As the multi-GPU research community is growing larger, it is important to develop powerful tools to further support research in related fields, and the development of MGPUSim fills the gap.

Cache-like Memory Systems: Meswani et al. [33] consider a two-layer memory system and introduce mechanisms to allow users to control where the data is stored in the closer-to-chip memory. CAMEO [13] is another prior study that manages a two-layer memory system with a cache-like memory system. Prior work has been mainly focusing on applying cache-like memory system design for CPUs, while PASI is a cache-like memory management solution tailored for multi-GPU systems.

Locality APIs: Vijaykumar et al. [52] propose using *locality descriptors* to reduce memory movement in both single- and multi-GPU environments. Our solution only tries to reduce inter-GPU data movement, and hence, can use simpler APIs that requires limited code modifications.

Virtual Address Management in GPUs: Ausavarungnirun et al. propose MASK [9], a framework to deal with virtual memory management for multiple applications running on a single GPU. In contrast, we emphasize an address translation-based solution for single applications executing on multiple GPUs. Page splitting for GPUs has been proposed in Mosaic [8]. However, Mosaic is also designed to work within a single GPU system. Efficient Page Splitting on a multi-GPU platform requires new considerations in terms of memory coherence to guarantee correct execution. PASI serves an overall hardware solution that combines page-migration, memory coherency, and page splitting to improve multi-GPU data sharing efficiency.

9 CONCLUSION

With the development of multi-GPU platforms, the research community demands new tools to explore faster and scalable multi-GPU designs. In this paper, we have proposed a new, flexible, and high-performance, parallel multi-GPU simulator called MGPUSim that would facilitate the design of future multi-GPU platforms. We have extensively validated MGPUSim with both microbenchmarks and full workloads against a real multi-GPU platform.

We have used two concrete design studies to both showcase the power and the flexibility of the simulator, and to provide a new solution to improve the scalability of multi-GPU platforms. Equipped with the Locality API introduced in the first design study, a programmer can control the data and compute placement and improve performance without modifying the kernel written for a single GPU implementation. With PASI, the GPU hardware can progressively improve the data placement during application execution. For a discrete 4-GPU platform we achieve a 1.6× and a 2.6× average speedup (geometric mean) with Locality API and PASI, respectively, when compared with a unified 4-GPU platform.

Designing a computer architecture simulator is a long-term effort. Despite the reasonable overall accuracy and flexibility we have achieved, we will continue to support the simulator for the community by adding new features (e.g., supporting atomic operations) and additional workloads. We also plan to explore the multi-GPU design space more thoroughly, including different inter-GPU network architectures and fabrics, DRAM technologies (e.g. HBM, GDDR6), and scaling the GPU count in the system.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for the useful feedback. This work was supported in part by NSF CNS-1525412, NSF CNS-1525474, MINECO TIN2016-78799-P, NRF 2015M3C4A7065647, NRF 2017R1A2B4011457, and HSA Foundation.

REFERENCES

[1] AMD. 2015. AMD Radeon R9 Series Gaming Graphics Cards with High-Bandwidth Memory.
[2] AMD. 2016. Graphics Core Next Architecture, Generation 3, Reference Guide. (2016).
[3] AMD. 2017. AMD APP SDK 3.0 Getting Started. (2017).
[4] AMD. 2017. Vega Instruction Set Architecture, Reference Guide. (2017).
[5] AMD. 2018. Radeon Compute Profiler. https://github.com/GPUOpen-Tools/RCP
[6] AMD. 2018. Radeon Instinct MI60 Accelerator. https://www.amd.com/en/products/professional-graphics/instinct-mi60
[7] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 320–332.
[8] 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*. ACM, 136–150.
[9] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. 2018. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 503–518.

[10] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 163–174.

[11] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. 2016. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678* (2016).

[12] Cen Chen, Kenli Li, Aijia Ouyang, Zhuo Tang, and Keqin Li. 2017. Gpu-accelerated parallel hierarchical extreme learning machine on flink for big data. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 47, 10 (2017), 2740–2753.

[13] Chia Chen Chou, Aamer Jaleel, and Moinuddin K Qureshi. 2014. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 1–12.

[14] Sylvain Collange, David Defour, and David Parello. 2009. Barra, a parallel functional GPGPU simulator. (2009).

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. Ieee, 248–255.

[16] Shi Dong, Xiang Gong, Yifan Sun, Trinayan Baruah, and David Kaeli. 2018. Characterizing the microarchitectural implications of a convolutional neural network (cnn) execution on gpus. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 96–106.

[17] Richard M Fujimoto. 1990. Parallel discrete event simulation. *Commun. ACM* 33, 10 (1990), 30–53.

[18] Nitin A Gawande, Jeff A Daily, Charles Siegel, Nathan R Tallent, and Abhinav Vishnu. 2018. Scaling deep learning workloads: NVIDIA DGX-1/Pascal and intel knights landing. *Future Generation Computer Systems* (2018).

[19] 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 *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 608–619.

[20] Wen-mei Hwu. 2015. *Heterogeneous System Architecture: A new compute platform infrastructure*. Morgan Kaufmann.

[21] 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.

[22] Hai Jiang, Yi Chen, Zhi Qiao, Tien-Hsiung Weng, and Kuan-Ching Li. 2015. Scaling up MapReduce-based big data processing on multi-GPU systems. *Cluster Computing* 18, 1 (2015), 369–383.

[23] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W Keckler, Mahmut T Kandemir, and Chita R Das. 2015. Anatomy of gpu memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 223–234.

[24] David Kanter. 2015. Graphics processing requirements for enabling immersive vr. *AMD White Paper* (2015).

[25] Gwangsun Kim, Minseok Lee, Jiyun Jeong, and John Kim. 2014. Multi-GPU system design with memory networks. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 484–495.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[27] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K Reinhardt, and Lizy K John. 2017. GPU triggered networking for intra-kernel communications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 22.

[28] Michael LeBeane, Brandon Potter, Abhisek Pan, Alexandru Dutu, Vinay Agarwala, Wonchan Lee, Deepak Majeti, Bibek Ghimire, Eric Van Tassell, Samuel Wasmundt, et al. 2016. Extended task queuing: Active messages for heterogeneous systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 80.

[29] Sangpil Lee and Won Woo Ro. 2013. Parallel GPU architecture simulation framework exploiting work allocation unit parallelism. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 107–117.

[30] Sangpil Lee and Won Woo Ro. 2016. Parallel gpu architecture simulation framework exploiting architectural-level parallelism with timing error prediction. *IEEE Trans. Comput.* 4 (2016), 1253–1265.

[31] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 49–63. https://doi.org/10.1145/3297858.3304044

[32] Geetika Malhotra, Seep Goel, and Smruti R Sarangi. 2014. Gputejas: A parallel simulator for gpu architectures. In *High Performance Computing (HiPC), 2014 21st International Conference on*. IEEE, 1–10.

[33] Mitesh R Meswani, Gabriel H Loh, Sergey Blagodurov, David Roberts, John Slice, and Mike Ignatowski. 2014. Toward efficient programmer-managed two-level memory hierarchies in exascale computers. In *2014 Hardware-Software Co-Design for High Performance Computing*. IEEE, 9–16.

[34] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. 2017. Beyond the socket: NUMA-aware GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 123–135.

[35] Saiful Mojumder, Marcia Louis, Yifan Sun, Amir Kavyan Ziabari, José L. Abellán, John Kim, David Kaeli, and Ajay Joshi. 2018. Profiling DNN Workloads on a Volta-based DGX-1 System. In *Proceedings of the International Symposium on Workload Characterization (IISWC'18)*. IEEE.

[36] NVIDIA. 2010. CUDA Programming guide.

[37] NVIDIA. 2018. Developing a Linux Kernel Module using GPUDirect RDMA. (2018).

[38] NVIDIA. 2018. NVIDIA DGX-2. (2018). https://www.nvidia.com/en-us/data-center/dgx-2/

[39] NVIDIA. 2018. NVIDIA TITAN RTX. https://www.nvidia.com/en-us/titan/titan-rtx/

[40] Open Source Initiative. [n. d.]. The MIT Licence.

[41] Rajat Raina, Anand Madhavan, and Andrew Y Ng. 2009. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*. ACM, 873–880.

[42] Ahmed Sanaullah, Saiful A Mojumder, Kathleen M Lewis, and Martin C Herbordt. 2016. GPU-accelerated charge mapping.. In *HPEC*. 1–7.

[43] Tom Sercu, Christian Puhrsch, Brian Kingsbury, and Yann LeCun. 2016. Very deep multilingual convolutional neural networks for LVCSR. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 4955–4959.

[44] Daniel J Sorin, Mark D Hill, and David A Wood. 2011. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture* 6, 3 (2011), 1–212.

[45] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66–73.

[46] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–10.

[47] Yifan Sun, Saoni Mukherjee, Trinayan Baruah, Shi Dong, Julian Gutierrez, Prannoy Mohan, and David 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, 209–218.

[48] The Go Authors. 2009. Effective Go. (2009).

[49] Josep Torrellas. 1999. Cache-Only Memory Architecture. In *IEEE Computer Magazine*.

[50] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: a simulation framework for CPU-GPU computing. In *Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*. IEEE, 335–344.

[51] Jan Vesely, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H Loh, Mark Oskin, and Steven K Reinhardt. 2018. Generic system calls for GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 843–856.

[52] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B Gibbons, and Onur Mutlu. 2018. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs. ISCA.

[53] Jun Wang, Eric Papenhausen, Bing Wang, Sungsoo Ha, Alla Zelenyuk, and Klaus Mueller. 2017. Progressive clustering of big data with GPU acceleration and visualization. In *Scientific Data Summit (NYSDS), 2017 New York*. IEEE, 1–9.

[54] Siyue Wang, Xiao Wang, Shaokai Ye, Pu Zhao, and Xue Lin. 2018. Defending DNN Adversarial Attacks with Pruning and Logits Augmentation. In *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 1144–1148.

[55] James Whitney, Chandler Gifford, and Maria Pantoja. 2018. Distributed execution of communicating sequential process-style concurrency: Golang case study. *The Journal of Supercomputing* (2018), 1–14.

[56] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. 2015. Deep image: Scaling up image recognition. *arXiv preprint arXiv:1501.02876* (2015).

[57] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Villa Oreste. 2018. Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. ACM.

[58] Amir Kavyan Ziabari, Yifan Sun, Yenai Ma, Dana Schaa, José L Abellán, Rafael Ubal, John Kim, Ajay Joshi, and David Kaeli. 2016. UMH: A hardware-based unified memory hierarchy for systems with multiple discrete GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4 (2016), 35.