Full-System Simulation of Mobile CPU/GPU Platforms

Kuba Kaszyk*, Harry Wagstaff*, Tom Spink*, Björn Franke*, Mike O'Boyle*, Bruno Bodin† and Henrik Uhrenholt‡

*School of Informatics, University of Edinburgh, Edinburgh EH8 9AB, Email: see https://www.ed.ac.uk/informatics/people

† Yale-NUS College, School of Computing, National University of Singapore. Email: bruno.bodin@yale-nus.edu.sg

‡Arm Sweden, Lund, Sweden Email: Henrik.Uhrenholt@arm.com

***Abstract*—Graphics Processing Units (GPUs) critically rely on a complex system software stack comprising kernel- and user-space drivers and Just-in-time (JIT) compilers. Yet, existing GPU simulators typically abstract away details of the software stack and GPU instruction set. Partly, this is because GPU vendors rarely release sufficient information about their latest GPU products. However, this is also due to the lack of an integrated CPU/GPU simulation framework, which is complete and powerful enough to drive the complex GPU software environment. This has led to a situation where research on GPU architectures and compilers is largely based on outdated or greatly simplified architectures and software stacks, undermining the validity of the generated results. In this paper we develop a full-system system simulation environment for a mobile platform, which enables users to run a complete and unmodified software stack for a state-of-the-art mobile Arm CPU and Mali-G71 GPU powered device. We validate our simulator against a hardware implementation and Arm's stand-alone GPU simulator, achieving 100% architectural accuracy across all available toolchains. We demonstrate the capability of our GPU simulation framework by optimizing an advanced Computer Vision application using simulated statistics unavailable with other simulation approaches or physical GPU implementations. We demonstrate that performance optimizations for desktop GPUs trigger bottlenecks on mobile GPUs, and show the importance of efficient memory use.**

***Index Terms*—Computer simulation**

I. Introduction

GPU simulation is central to driving GPU research and development. It is used for early design space exploration and architecture tuning [1]–[3], evaluation of GPU compilation techniques [4], application development and optimization [5], and in virtual platforms for system software development [6]. While Central Processing Unit (CPU) simulation techniques have reached maturity, GPU simulation often suffers from the following problems: (a) instruction sets are not accurately modeled, but approximated by an *artificial, low-level intermediate representation* [7], [8], (b) GPU simulators do not model existing commercial GPUs, but only *simplified GPU architectures* [9], (c) instead of using vendor provided driver stacks and compilers, GPU simulators often rely on *simplified system software*, which may behave entirely differently to original tools [10], [11], and (d) *GPUs are treated as standalone devices*, not modeling any CPU-GPU transactions [12]. This has led researchers using GPU simulation to rely on tools providing questionable accuracy [13].

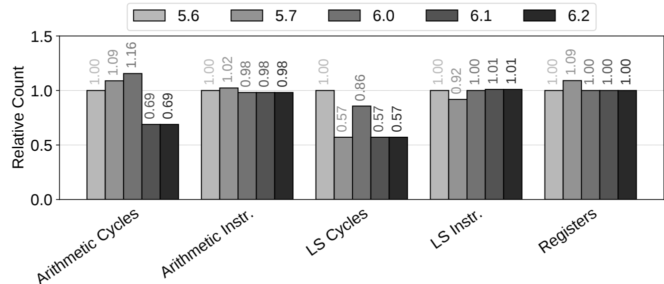


Fig. 1: MatrixMul: Different versions of Arm's OpenCL compiler result in substantially different code for the G-71.

For example, many existing GPU simulators including gem5-GPU [14], GpuTejas [15], Multi2Sim [10], GPGPU-Sim [16], and Multi2Sim-Kepler [11] claim cycle-accuracy. However, despite their claimed cycle-accuracy all of these simulators show significant differences in the reported cycle count (or other reported performance metrics) compared to actual hardware. In extreme cases, these errors can be in excess of over 100%. Other GPU simulators have either not been evaluated against hardware reference platforms or do not attempt to model any available GPU. Available instruction-accurate simulators, i.e. those without a cycle-level timing model, also show significant errors. For example, Barra [17] exhibits up to 81.6% difference in instruction counts compared to those reported by measurements on a real GPU.

Further error is introduced by the use of outdated or non-standard GPU tool chains required by several simulators. We compiled a set of OpenCL kernels with different versions of the vendor supplied compiler[1] (v5.6, 5.7. 6.0, 6.1, 6.2) for the Arm Mali-G71. Fig. 1 shows that we observed major differences between compiler versions, e.g. GPU arithmetic cycles in the selected kernel differ by 47% (6.0 to 6.1). It is more than likely that simplified or non-vendor supplied tool chains used by other GPU simulators introduce even greater error, as also highlighted in [13].

In this paper we claim that without a truly accurate GPU simulation model and a full-system environment capable of running an unmodified GPU software stack and applications

[1]https://developer.arm.com/products/software-development-tools/graphics-development-tools/mali-offline-compiler

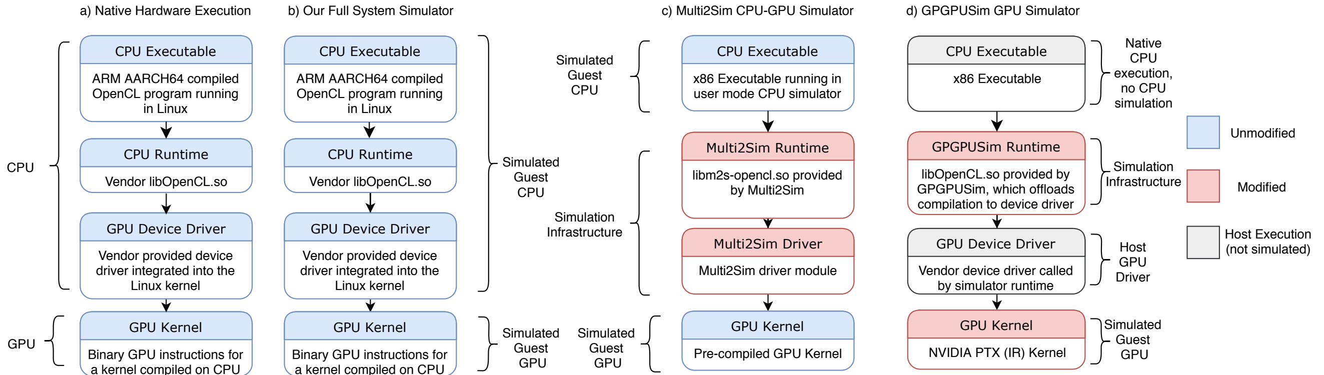


Fig. 2: Comparison of the GPU kernel execution model and software stack for (a) a native execution environment, (b) our full-system simulator, (c) Multi2Sim, and (d) GPGPU-Sim. For MultiSim and GPGPU-Sim we have highlighted non-standard software components that are different from the vendor-supplied driver stack and thus represent a source of inaccuracy.

it is not possible to gather reliable performance metrics to underpin GPU architecture research.

A. *Full-System GPU Simulation*

In this paper, we propose a fundamentally different approach to GPU simulation, avoiding the aforementioned issues. The goal of this work is to accurately simulate a state-of-the-art mobile GPU in a full-system context, enabling the use of *unmodified* vendor-supplied drivers and JIT compilers, operating in an *unmodified* target operating system, and executing *unmodified* applications. This requires our GPU simulator to be architecturally identical to a physical GPU, and model all components of the application and system software stack.

We focus on functional CPU/GPU simulation, i.e. without detailed timing information. While this method sacrifices cycle-accuracy, it enables us to improve simulation performance to a level where it is feasible to run complex CPU/GPU workloads. Such a functional simulator is also a prerequisite to detailed timing simulation and can still provide useful execution statistics, such as instruction counts, memory traces, and CPU-GPU transaction details. Simultaneously our system guarantees optimal GPU feature support, and ensures that our virtual platform executes identical code to that on physical hardware. Our fast simulation approach also supports interactive workloads, and new Application Programming Interfaces (APIs)s (e.g. Vulkan) without additional engineering.

Notable use cases for our full-system CPU/GPU simulation technology are (1) early GPU design space exploration, where a GPU currently under design can be evaluated and (2) virtual platforms for both system- and user-level software development, both without producing a physical version. These use cases benefit particularly from the accuracy and performance that our integrated CPU/GPU simulation approach offers.

B. *State-of-the-Art*

In order to further motivate our full-system approach to CPU/GPU simulation, we initially review the most popular GPU simulators: GPGPU-SIM [16] and MULTI2SIM [10], [11]. In Fig. 2 we compare the GPU kernel execution and software stacks for a native execution environment, our full-system simulation, MULTI2SIM, and GPGPU-SIM.

In native hardware (Fig. 2(a)), a CPU executable is run in Linux on an Arm CPU. This executable includes an embedded OPENCL kernel, and interacts with a vendor provided runtime library, e.g. `libOpenCL.so`, to JIT compile the OPENCL kernel to GPU instructions. This runtime interacts with a GPU device driver–a vendor-specific kernel module for low-level CPU-GPU interaction–which manages the setup of GPU jobs. Finally, the GPU executes binary instructions from memory.

Our full-system simulation model (Fig. 2(b)) implements both the CPU and GPU completely and accurately. We run the original unmodified executable and the original CPU runtime environment for the GPU and GPU driver. Our GPU simulation component completely emulates Arm's Bifrost architecture, executing the same binary as the physical GPU implementation in Fig. 2(a). Our GPU simulator interacts with the simulated CPU and driver executing on the CPU in the same way as its physical counterpart, making the simulation identical to a physical GPU for the entire software stack.

Compare this to MULTI2SIM in Figure 2(c). MULTI2SIM's OPENCL stack differs substantially from the native stack. OPENCL function invocations are handled by a non-standard runtime, which is intercepted by the CPU simulator, and redirected to the GPU simulator to launch the kernel execution.

Tools like Multi2Sim require heavy maintenance as toolchains advance. We have seen that code compiled by newer versions of AMDs OPENCL compiler, which the Multi2Sim toolchain Multi2C relies on, often contains features unsupported in Multi2Sim. The user then must rely on an outdated (and now unavailable) version of the OPENCL compiler.

GPGPU-SIM (Fig. 2(d)) provides a model for Parallel Thread Execution (PTX) or SASS execution, where PTX is a scalar low-level, data-parallel virtual Instruction Set Architecture (ISA) defined by NVIDIA, and SASS is the native shader assembly for NVIDIA GPUs. While PTX is an intermediate representation, SASS is closer to the actual GPU instruction

set. However, GPGPU-SIM requires its own runtime libraries and device drivers, which (a) differ substantially from the vendor supplied libraries, (b) are not feature complete, and (c) introduce significant accuracy problems.

C. Contributions

In this paper we develop a full-system system simulation environment for a mobile platform, enabling users to run a complete and unmodified software stack for a state-of-the-art mobile Arm CPU and Mali-G71 GPU powered device. We validate our simulator against a hardware implementation as well as Arm's stand-alone GPU simulator, achieving 100% architectural accuracy across all available toolchains. We demonstrate the capability of our GPU simulation framework by optimizing an advanced Computer Vision application using simulated statistics unavailable with other simulation approaches or physical GPU implementations. We then make a direct comparison against desktop GPUs, and show that memory usage is hugely significant to mobile GPU performance.

II. BACKGROUND: ARM BIFROST GPU

Here, we provide a brief overview of the Arm Bifrost GPU architecture. This is a state-of-the-art mobile GPU design powering many high-end smartphone Systems-on-chips (SoCs). In our simulator we implement the Arm Mali-G71 GPU, found in e.g. the Exynos 8895 SoC that powers the Samsung Galaxy S8. Fig. 3a gives an overview of the Bifrost architecture.[2]

The architecture features up to 32 unified Shader Core (SC)s, and a single logical L2 GPU cache that is split into several fully coherent physical cache segments. Full system coherency support and shared main memory tightly couples the GPU and CPU memory systems. For this, Bifrost features a built-in Memory Management Unit (MMU) supporting AArch64 and LPAE address modes. A central Job Manager (JM) interacts with the driver stack and orchestrates GPU jobs.

A. Shader Cores

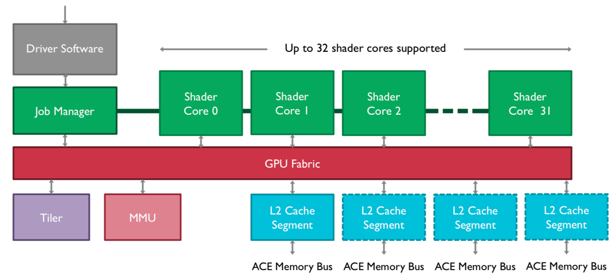
Shader Core (SC)s (see Fig. 3b), are blocks consisting of Execution Engine (EE)s–three in the Mali-G71–and a number of data processing units, linked by a messaging fabric.

The EEs are responsible for executing the programmable shader instructions, each including an arithmetic processing pipeline as well as all of the required thread state.

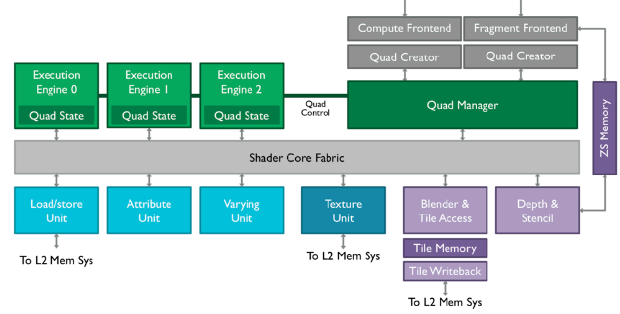
The arithmetic units implement a vectorization scheme to improve functional unit utilization. Threads are grouped into bundles of four (a "quad"), which fill the width of a 128-bit data processing unit. From the viewpoint of a single thread, the architecture behaves as a stream of scalar 32-bit operations.

Instructions are bundled into clauses of up to 8 tuples (16 instructions), as shown in Fig. 4a. Within a clause, instructions can access temporary registers, reducing pressure on the global register file (see Fig. 4b). Further details can be found in [18].

[2]Figures 3 and 4a reproduced with kind permission of Arm Ltd.

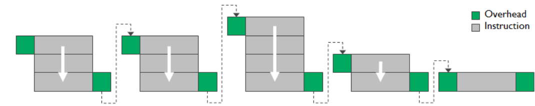


(a) Bifrost GPU design

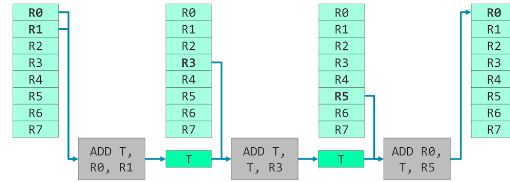


(b) Mali-G71 shader core design

Fig. 3: Bifrost GPU architecture and Shader Cores [18].



(a) Arm Bifrost clause scheduling and execution model [18].



(b) Temp. registers within a clause reduce global register file accesses.

Fig. 4: The Bifrost GPU execution model.

B. Job Manager

The Job Manager (JM) receives jobs from the GPU device driver, and schedules them for execution on the GPU. The jobs contain information specific to the shader being executed, including job dependences, dimensions, and pointers to data the shader binary, which is then used to map jobs onto SCs.

C. Supported APIs

The Arm Mali-G71 GPU offers full support for current and next generation APIs, enabling advanced 3D graphics acceleration and GPU compute functionality. This includes support for the KHRONOS OPENGL ES 3.2, 3.1/2.0/1.1, VULKAN 1.0 and OPENCL1.1/1.2/2.0 Full Profile APIs. Additionally, support is provided for the Android Extension Pack, Android Renderscript, and Microsoft Windows DIRECTX 11.

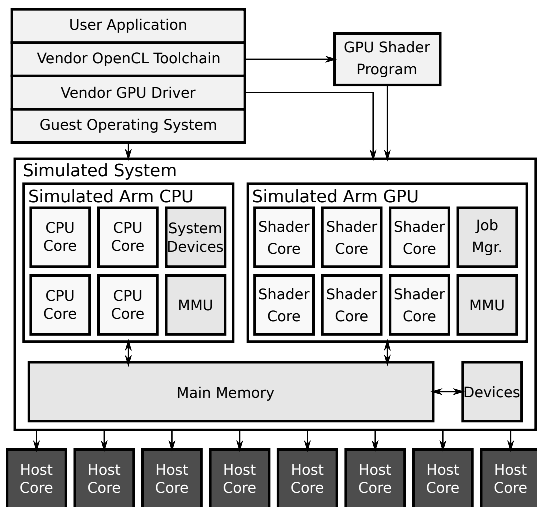


Fig. 5: Guest applications execute on simulated CPU-GPU platform, running native Arm Linux with unmodified GPU device drivers. Guest cores map onto host threads.

III. Our Simulation Approach

Our simulation environment, as shown in Fig. 5, provides a full-system view of a CPU/GPU platform. Such an approach also requires additional components to be emulated including an MMU, interrupt controller, timer devices, storage and network devices. In order to benefit from existing device drivers, we model the Arm Versatile Express and Juno platforms, each augmented with an Arm Mali-G71 GPU.

Both the simulated CPU and GPU are modeled using high-level architecture descriptions [19], and generated using a retargetable simulation framework [20], which also supports other architectures. They each run in separate threads on the host CPU, providing concurrent and asynchronous operation.

A. CPU Simulation

We simulate the CPU through full-system Dynamic Binary Translation (DBT) (similar to QEMU [21]), which boots a Linux Arm kernel and user space from a file system mounted by the simulated storage device. For complete and accurate modeling, we simulate essential platform devices, ensuring that our simulator can support a full software stack without simulation-specific adaptation of any software component.

B. GPU Simulation

We generate an interpretive GPU simulation module for the programmable GPU SCs from the Mali architecture description, and non-core components are directly implemented.

1) CPU-GPU Interface: The GPU interfaces with the CPU via memory mapped registers, hardware interrupts, and memory, through which the simulated GPU exposes its JM to the CPU. For GPU compute jobs, the OpenCL driver sets up shader programs in the shared CPU-GPU memory space, and then triggers an interrupt on the GPU by writing to a control register, indicating that a job is ready for execution. These interrupts are visible to the JM, which begins execution.

2) Shader Core Simulation: The generated simulator code comprises the instruction decoder and main EEs of the GPU. The interpretive execution model is split into two phases: (1) decode, and (2) execution. During phase one, the shader program and its associated metadata are decoded for later use. In phase two, a dispatcher iterates over the job dimensions and creates simulated *GPU* threads. These threads are grouped into "warps", where all threads execute in lockstep. Warps are in turn grouped into thread-groups, i.e. OpenCL workgroups.

3) Performance Optimizations: The simulation is broken up into two stages - decode, and execution. During the decode stage, the GPU extensively caches guest code, which is then accessed during the execution phase. This model ensures that the entire shader program is decoded exactly once.

In hardware, each SC executes one thread-group at a time. In our simulator, however, the number of SCs and host threads is individually configurable. For example, instead of mapping 8 SCs onto 8 host threads, we can map the executing thread-groups onto 32 host threads, creating *virtual cores*.

This necessitates additional measures for managing local storage. The GPU driver allocates local storage for 8 thread-groups corresponding to the 8 detected SCs. To support more thread-groups executing in parallel, the simulator allocates additional local memory for each host thread, outwith the guest system. Local guest memory accesses are intercepted and mapped to host memory, guaranteeing functional correctness.

4) Job Manager Simulation: In our GPU simulator the JM operates in its own host simulation thread. It fully implements the functionality of its hardware counterpart such as parsing job descriptors and orchestrating the operation of the SCs.

5) Memory Management Unit Simulation: Our simulator incorporates a complete software implementation of the GPU's MMU. The driver provides the MMU with page table pointers, and the MMU reports errors (permissions violations, faults) to the driver through memory mapped registers and interrupts.

IV. Instrumentation

Through instrumentation the simulator provides useful statistics, without the overhead of a cycle-accurate simulator:

A. Program Execution

We gather *instruction counts and breakdowns, data accesses, and clause information* - statistics directly relating to the executing instructions. From these we can directly see the codesize, ratios of memory instructions to arithmetic, types of memory accesses - all vital to understanding performance implications of the executed code. Each clause is instrumented with detailed metrics at decode time, and during execution, we record clause frequency. If executing with multiple host threads, this is gathered by each parallel unit. Metrics are totalled at job completion, requiring no further synchronization.

B. System

The GPU operates as an accelerator, therefore it is vital to understand its interaction with the rest of the system. The number of pages accessed by the GPU shows the interaction

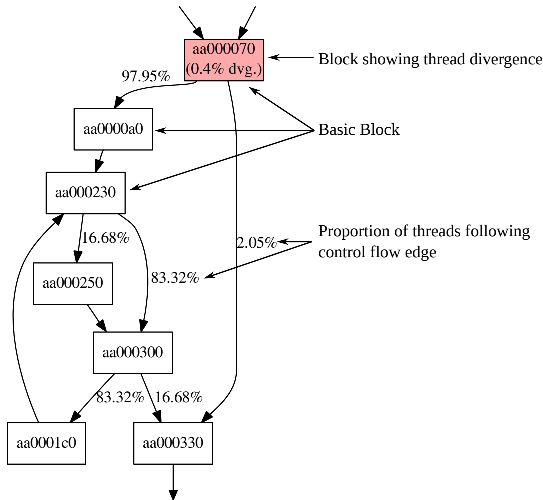


Fig. 6: *BFS*: Our simulator generates a control flow graph pinpointing the divergence on actual GPU instructions.

with the memory system and MMU, which are expensive in terms of performance. Interrupts and system register accesses describe the communication with the CPU –also a bottleneck.

C. Control Flow

Control flow execution in the GPU monitors thread divergence, which occurs when threads within a warp take different paths after a conditional branch. This is a serious performance problem, as if a thread diverges, other threads in the warp must wait for the diverging thread to reconverge, without scheduling other work. We monitor this by tracking the PC on clause boundaries, and building a Control Flow Graph (CFG). This CFG, shows which thread executes which path, and identifies diverging threads at their divergence point, as shown in Fig. 6.

Simulated Platform	Arm-v7A/v8A CPU Arm Mali Bifrost GPU - G71, 8 Cores
	Arch Linux (Kernel 4.8.8) Arm Mali Bifrost DDK r3p0/r9p0
Multi2Sim Eval. Platform	x86 CPU, Southern Islands GPU
Evaluation Platform	HiKey960 - Arm-v8A CPU Arm Mali Bifrost GPU - G71, 8 Cores
	Android-O/Debian Linux Arm Mali Bifrost DDK r3p0/r9p0
Host Platform 1 (main experiments)	Intel(R) Core(TM) CPU i7-4710MQ 4 cores with HT, 2.50GHz
Host Platform 2 (Parallel scaling, Fig. 10)	Intel(R) Xeon(R) CPU L7555 32 cores with HT, 1.87GHz

TABLE I: System configurations for performance evaluation.

V. Evaluation

First, we present the validation strategy for our simulator against Arm hardware and a proprietary simulator, achieving 100% architectural accuracy across all available toolchains. We then compare our simulator's performance and effectiveness against Multi2Sim 5.0. Unless explicitly stated, all comparisons against Multi2Sim use Multi2Sim's functional simulation mode. Finally, we demonstrate the versatility of our simulator through a series of use cases. Our evaluation focuses on the widely accepted OpenCL compute API, which allows for direct comparison with other GPU simulators.

Suite	**Benchmark**	**Input Type & Size**
Rodinia 3.1	Back Propagation	65536 nodes
Parboil	Breadth First Search	1257001 nodes
AMD APP 2.5	Binary Search	16777216 elements
AMD APP 2.5	Binomial Option	512 samples
AMD APP 2.5	Bitonic Sort	2048 elements
Parboil	Cutoff-limited Coulombic Potential (cutcp)	67 atoms
AMD APP 2.5	DCT	10000x1000 matrix
AMD APP 2.5	DwtHaar1D	8388608 signal
AMD APP 2.5	Floyd Warshall	256 nodes
AMD APP 2.5	Matrix Transpose	3008x3008 matrix
Rodinia 3.1	Nearest Neighbor	5 records 30 latitude 90 longitude
AMD APP 2.5	Recursive Gaussian	1536x1536 image
AMD APP 2.5	Reduction	9999360 elements
AMD APP 2.5	Scan Large Arrays	1048576 elements
Parboil	SGEMM	128x96, 96x160 matrices
AMD APP 2.5	SobelFilter	1536x1536 image
Parboil	Sparse Matrix Vector Mult.	1138x1138x2596 matrix 2596 elements
Parboil	Stencil	128x128x32 matrix 100 iterations
AMD APP 2.5	URNG	1536x1536 image
clBLAS	SGEMM	1024x1024 matrix

TABLE II: Benchmarks and data set sizes.

Details of our host and guest platforms are provided in Table I. As different benchmarks scale in different ways, the default host configuration used 8 threads for GPU simulation. We show additional results for selected benchmarks.

We chose kernels from a variety of benchmark suites. First, we include AMD APP SDK 2.5 as pre-compiled GPU binaries packaged with Multi2Sim enable direct comparison. AMD driver 2.5, which Multi2Sim, and its compiler Multi2C, rely on, is no longer available, and code compiled using newer versions often contains features unsupported by Multi2Sim. We also report results for Parboil [22] and Rodinia [23] benchmark suites, which provide larger, more complex workloads. The benchmarks and inputs are presented in Table II.

Next, we consider a robotic vision application, SLAMBench, demonstrating a concrete use-case for our approach. We show that our simulated metrics relate directly to hardware runtimes. Finally, we demonstrate how optimizations for desktop GPUs trigger bottlenecks on embedded GPUs.

A. Validation and Accuracy

Correctness of our full-system simulation approach has been established by comparison against the commercially available HiKey 960 with a Mali-G71 MP8 GPU. We also validated the GPU part of our simulator against a detailed proprietary simulator for the target GPU architecture. Our comparisons

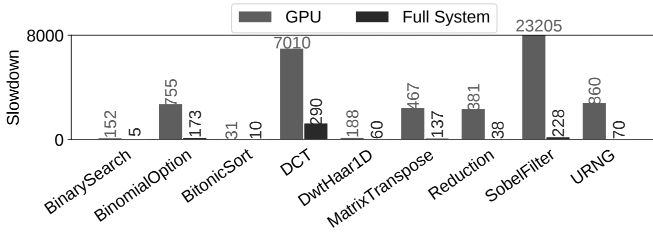


Fig. 7: Simulation slowdown relative to the HIKEY960 for GPU only, and for the entire benchmark (CPU+GPU).

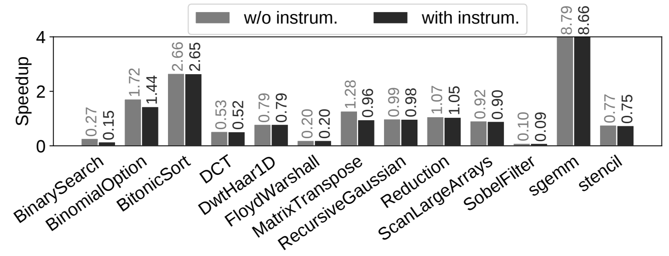


Fig. 8: Our simulator's speed with and without instrumentation, relative to Multi2Sim functional simulation (=1.0).

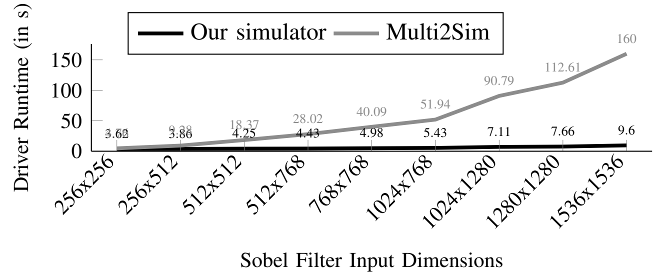


Fig. 9: The software stack executing on our DBT CPU simulator scales exceptionally well relative to Multi2Sim.

have shown complete accuracy for the evaluated benchmarks, for all evaluated metrics. This is possible only because our simulation is driven by the exact binary that is executed in hardware, thanks to the full support of a native software stack.

1) Comparison to Hardware: Validation has focused on: (a) Correctness of OPENCL kernel execution on the GPU, evaluated through extensive testing, (b) correctness of performance metrics, including instruction counts, instruction breakdowns, clause sizes, data access breakdowns, and divergence, for which we compare results from our instrumented simulator to hardware performance counters on the HIKEY 960.

2) Comparison to Reference Simulator: We have also validated our simulator against a proprietary, detailed standalone GPU simulator. We executed selected kernels on both simulators using an instruction tracing mode, where individual instructions and their effects are observable. Additionally, we employed fuzzing techniques for rigorous instruction testing, covering an extensive range of inputs.

B. Simulation Performance

Next we evaluate three key simulation performance metrics.

1) GPU OpenCL Simulation Speed: Fig. 8, presents execution performance of our GPU simulator relative to Multi2Sim, where most benchmarks exhibit similar performance levels. Exceptions are *BinarySearch* and *SobelFilter*, where our simulator is up to 10x slower than Multi2Sim, and *sgemm*, where our simulator is 8.8x faster. While this disparity is due to implementation differences between the simulators and simulated architectures, the results demonstrate that accurate full-system simulation of a GPU platform is feasible and yields competitive performance. Fig. 7 shows simulation slowdown over native execution. The average slowdown is 4561x.

Full instrumentation of the GPU simulation generally adds <5% overhead, due to the approach described in Section IV. This means that we provide useful statistics, with performance similar to Multi2Sim's, which by default only reports instruction breakdown and job dimensions. In cycle-accurate mode, Multi2Sim reports additional statistics, including active execution units, compute unit occupancy, and stream core utilization, however, in our tests it failed to complete the majority of workloads, due to large inputs. On smaller workloads, we observe slowdowns of up to 10x over functional simulation.

2) CPU OpenCL Driver Simulation Speed: Full-system GPU simulation, executing the full software stack on the CPU, adds substantial stress to CPU simulation. Fig. 9 shows software stack runtimes for *SobelFilter* with different input sizes. While Multi2Sim spends >150s on CPU-side execution for the largest tested input, our JIT-based CPU simulator executes the entire stack in <10s, resulting in better performance, while maintaining complete accuracy. Overall, Fig. 7 shows that slowdown for the entire system over native hardware is low, averaging only 223x slowdown.

3) Simulation Performance Optimizations: In Fig. 10 we evaluate the performance optimization introduced in Section III-B3, mapping GPU SCs onto multiple host threads. In the worst case, *BinarySearch* is iterative, with short kernels executing with heavy CPU interaction, limiting improvement. For *SobelFilter*, the best case, large thread-group sizes executed for a single kernel enable efficient parallel execution, resulting in steady speedup as host threads are added.

C. Application Results

We first focus on architectural features of Bifrost which would be useful in early design space exploration.

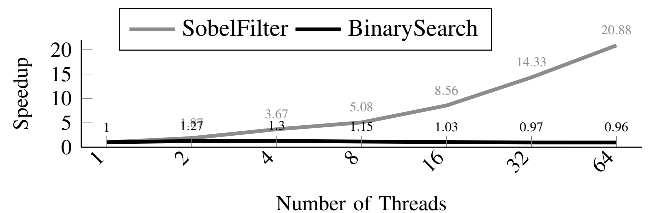


Fig. 10: Increasing the number of host simulation threads yields vast performance improvements for certain benchmarks.

1) Identifying Empty Instruction Slots: Fig. 11 shows instruction mixes for OPENCL benchmarks. For example, *SobelFilter* is a compute-intensive filter with very few empty slots and memory accesses and almost no control flow. In contrast, the number of empty slots in *Reduction* and *ScanLargeArrays* indicates low GPU utilization. On average, 50% of instructions are arithmetic operations, while local memory and control flow each contribute around 10%. Performance can be substantially improved by reducing the number of empty instruction slots introduced by the OPENCL toolchain.

2) Moving Data Closer the Core: Different types of data storage have various access latencies, which when poorly utilized can lead to colossal drops in performance. Ideally, data should be kept as close to the GPU's execution cores as possible. Our simulator shows exact data placement throughout the hierarchy, and can be used to guide optimization.

Data breakdowns are shown in Fig. 12. *SobelFilter* exhibits few main memory accesses, while the figures for *backprop* suggest that it could benefit from enhancements to the OPENCL compiler, more registers, or a better algorithm. Fast accesses to temporary values, constants and ROM dominate. More reads from than writes to global registers suggest effective reuse of register data. Global memory accesses account for <10% of accesses, except for a single case, *backprop*.

3) Evaluating the Bifrost Clause Model: Clauses contain up to 8 instruction words (16 instructions), which execute unconditionally. Longer clauses are preferable - they reduce global register file accesses through temporary register use and limit the scope for control flow and thread divergence.

Fig. 13 shows the distribution of clause sizes for all benchmarks. Several, including *BinomialOption* and *FloydWarshall* exhibit a majority of clauses of size 1 or 2, and occasionally size 8. Others peak at mid-size clauses, e.g. *BitonicSort*, or are bimodally distributed, e.g. *RecursiveGaussian*. Compare this to the instruction mix in Fig. 11, where e.g. *RecursiveGaussian* features a larger fraction of arithmetic instructions and few empty slots, whereas *Reduction* is reversed. Overall, kernels with larger clauses feature fewer empty slots, while short clauses and empty slots show some correlation.

Potentially, some kernels perform little work between control flow operations, or the compiler is unable make use of available slots. Benchmarks with shorter clauses also display a large proportion of memory accesses, suggesting that memory bottlenecks limit the potential of the clause model. The model might suit graphics workloads, as they benefit from additional data processing units and exhibit regular behaviour, however re-visiting the model for compute might be worthwhile.

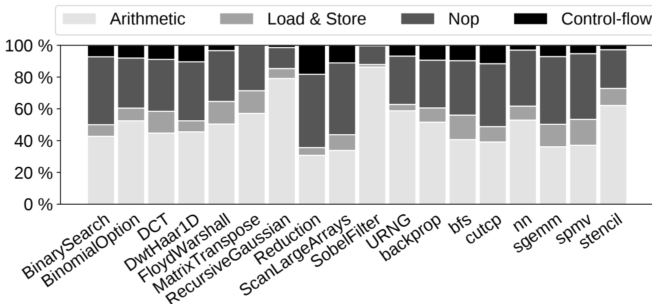


Fig. 11: A breakdown of instruction mixes and empty slots can help us identify bottlenecks in GPU code.

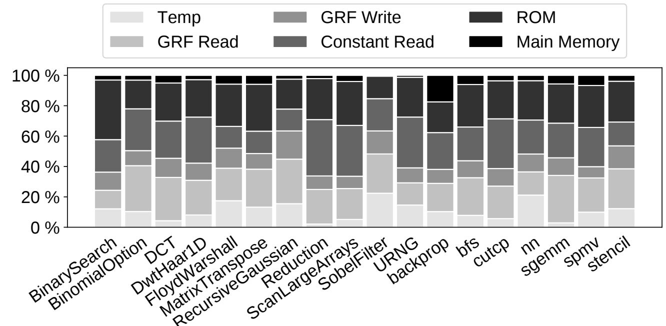


Fig. 12: Data access breakdowns show a complete view of each architecturally visible level of memory hierarchy.

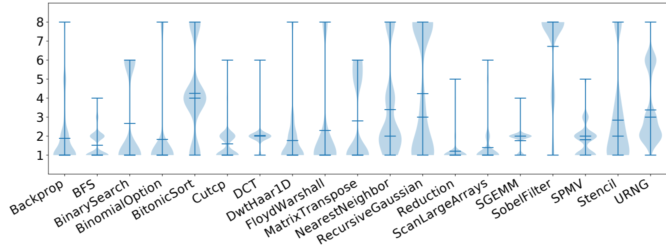


Fig. 13: A clause size distribution presents optimization targets and bottlenecks for the Bifrost clause model.

D. System Level Results

CPU-GPU communication can account for as much as 76% of execution time [23]. In our full-system environment, we are able to gather system-level statistics unavailable to other GPU simulators or hardware. Our approach provides the capability to observe CPU-GPU interactions, allowing us to monitor memory usage, interrupts, and control register accesses, presented in Table III for selected benchmarks. While *SobelFilter* exhibits little CPU-GPU interaction, *BFS* touches more pages, and involves a higher number of transactions.

Page use differs by up to three orders of magnitude across benchmarks, with *stencil* and *BFS* dominating this metric. *BFS* is particularly heavy on control interactions showing an unusually high number of control register accesses and interrupts resulting from over 1000 individual compute jobs.

E. Optimizing OpenCL Applications

1) SLAMBench: We demonstrate the capabilities of our full-system simulator by evaluating the OpenCL SLAMBENCH [24] computer vision application, which comprises several compute kernels and dataflow orchestrated by the CPU.

Benchmark	Page Acc.	Ctrl. Reg Reads	Ctrl. Reg Writes	Interr. Asserted	Comp. Jobs
BFS	51723	308098	66209	8022	1003
Binomial Option	31	136	70	4	1
SobelFilter	4609	136	70	4	1
Stencil	99603	14795	1982	105	100

TABLE III: System statistics detail the CPU-GPU interaction.

In its full configuration, SLAMBench executes 40000 kernels, impossible to simulate with existing GPU simulators out-of-the-box, due to their limitation to single kernels, tool chain incompatibilities or lack of support for CPU-GPU interactions.

Countless configuration options are available in the benchmark, each with varying performance. We execute the KFusion benchmark, with *standard*, *fast3*, and *express* configurations. Fig. 14 shows metrics for *fast3* and *express* relative to *standard*. Both show major improvement. The relative instruction count for each category is at most 8% for *Fast3* and just 2% for *Express*, while the ratio for local memory instructions is much higher - 29% for *Fast3* and 19% for *Express*, meaning increased local memory use relative to total instruction count.

Our metrics can easily guide us to a good solution, without requiring hardware. While we cannot predict the exact framerate, the simulated metrics suggest successive improvement between *standard*, *Fast3*, and *Express*. This is truly the case - *Fast3* is 3.35 times faster than *standard* and *Express* is 7.72 times faster than *standard*.

2) SGEMM: [25] shows that optimizations applied to the same code targeting different architectures result in greatly different performance relative to hand-tuned code. This is exacerbated in mobile GPUs, whose architectures are completely different to desktop GPUs [26]. We evaluate this claim through six SGEMM kernels ([27], [28]), a core component of linear algebra and machine learning applications, which are increasingly moving to mobile devices. Starting with (1), the kernels in Fig.15 are iteratively optimized for NVidia GPUs. There is no correlation between speedups on Mali and NVidia, indicating vastly different architectures.

Again, simulation statistics directly relate to native runtimes, and we see that the optimal solution on Mali (4) executes far fewer instructions than the slowest version (6). Between (5) and (6), arithmetic and control flow instruction counts are similar, however (5) is almost twice as fast as (6). Interestingly,

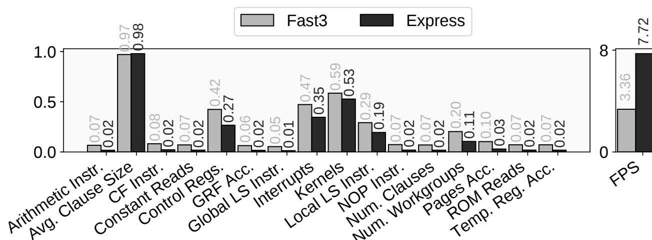


Fig. 14: Simulated SLAMBench statistics directly relate to HW performance, aiding the search for optimal configurations.

(6) meant to increase register usage, but the increase is just 3% on Mali. Instead, (6) greatly reduces local, and increases global memory accesses relative to (5). (4) almost completely avoids global memory, shifting instead to local memory. This supports the claim in [29], that data movement in mobile platforms is a major contributor to execution time and cost.

VI. Related Work

To ease comparison to other GPU simulation approaches we provide an overview of features in Table IV, including each simulator's maximum relative error as reported in their original publications. In some cases, accuracy has not been evaluated or the simulated GPU does not model existing GPUs.

[13] presents a cycle-accurate full-system CPU/GPU simulation framework, similar to our own, however the OS kernel driver is emulated, i.e. all driver calls are intercepted by the emulated driver. Similarly to Multi2Sim, any changes to the driver stack need to be implemented directly in gem5, whereas our framework supports any new drivers out of the box. Additionally, in [13] the runtime system executes natively, and only the GPU executes in the simulator. This makes it impossible to simulate systems in which the target CPU architecture differs from the host, for example, you could not simulate an Arm CPU + GPU on an x86 host. While their functional simulation is completely accurate, the cycle-accurate simulator exhibits an average error of 42%. Instruction-accurate simulation, such as our own, is a prerequisite for cycle-accurate simulation, and provides the basis for building a cycle-accurate full-system simulator, targeting any architecture. Instead, we focus solely on fast, functional simulation, enabling execution of realistic, high-intensity workloads such as SLAMBench, allowing for realistic modelling of the full system and software stack. In 2007, Fung et al. developed a cycle-level simulator (GpGPU-Sim) for an NVidia-like GPU built around the SimpleScalar back-end [7]. In this approach both the target ISA and toolchain are crude approximations. In [35] a Mali GPU is modeled and OpenCL kernels are simulated using GpGPU-Sim, yet its completeness and accuracy are insufficient for most use cases. Collange et al. have developed Barra, an architectural simulator for the native NVidia G8x and G9x instruction sets [17]. With a reimplementation of the low level Cuda runtime API, they produced a simulator capable of running Cuda applications. While the target ISA is matched, the software stack is vastly different from the vendor supplied Cuda environment. GPUOcelot [8] supports NVidia's Cuda API and implements a full function simulator providing an NVidia virtual machine referred to as Ptx – a machine model and low level virtual ISA that is claimed to be representative of ISAs for data parallel execution. The simulator can execute compiled kernels from the Cuda compiler, but the underlying machine architecture is an abstraction of the target machine and executes a form of intermediate representation with an added-on cost model. GPUTejas [15] uses GPUOcelot to capture a GPU execution trace for further parallel simulation. Gem5-GPU [14] combines the Gem5 and GpGPU-Sim simulators. While

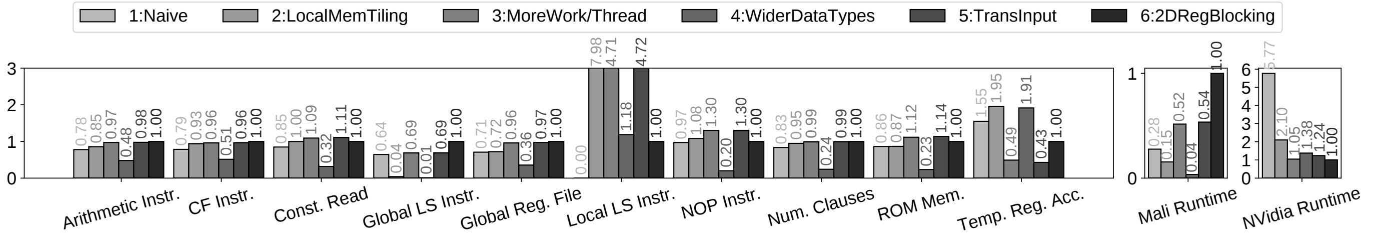


Fig. 15: Mali simulation statistics for different versions of SGEMM compared against native Mali, and NVidia K20m runtimes. All statistics are normalized to SGEMM6, the slowest kernel on Mali.

Simulator	Full System	Guest CPU	Guest GPU	GPU ISA	GPU Toolchain	Prog. Model	Perf. Model	Simulation Model	Max. Rel. Error[1]
Barra [17]	GPU only	N/A	NVIDIA Tesla	Approx. Tesla ISA	Emulated	CUDA	Instruction-Accurate	Execution-Driven	$\leq$ 81.6%
GPGPU-Sim [16]	GPU Only	N/A	NVIDIA-like	PTX GT200 SASS	Custom	CUDA	Cycle-Accurate	Execution-Driven	$\leq$ 50.0%
gem5-GPU [14]	Yes	x86	NVIDIA GTX 580	PTX GT200 SASS	Custom	CUDA	Cycle-Accurate	Execution-Driven	$\leq$ 22.0%
Multi2Sim [10]	Yes	x86/Arm/ MIPS	AMD Everg./S.Isl. NVIDIA Fermi	AMD GCN1 SASS	Custom	OpenCL CUDA	Cycle-Accurate	Execution-Driven	$\leq$ 30.0%
Multi2Sim Kepler [11]	Yes	x86/Arm/ MIPS	NVIDIA Kepler	SASS	Custom	CUDA	Cycle-Accurate	Execution-Driven	$\leq$ 200%
ATTILA [9]	GPU Only	N/A	ATTILA	ARB	Custom	OpenGL	Cycle-Accurate	Execution-Driven	N/A[2]
GPUOcelot [8]	GPU Only	N/A	NVIDIA AMD Radeon	PTX	Custom	CUDA	Instruction-Accurate	Trace-Based	Not Evaluated[3]
HSAemu [30]	Yes	Retargetable/ Arm-v7A	Generic	HSAIL	Custom	OpenCL	Cycle-Accurate	Execution-Driven	N/A[2]
GPUTejas [15]	GPU Only	N/A	NVIDIA Tesla	PTX GPUOcelot μ-ops	Custom	CUDA	Cycle-Accurate	Trace-Driven	$\leq$ 29.7%
MacSim [31]	Yes	x86	NVIDIA GeForce G80/GT200/Fermi	PTX GPUOcelot μ-ops	Custom	CUDA	Cycle-Accurate	Trace-Driven	Not Evaluated[3]
TEAPOT [32]	Yes	Generic	Generic Mobile GPU	Emulated	Custom	OpenGL	Cycle-Accurate	Trace-Driven	N/A[2]
QEMU/MARSSx86/ PTLsim [33]	Yes	x86	NVIDIA Tesla-like	Generic	Custom	OpenGL	Cycle-Accurate	Execution-Driven	Not Evaluated[3]
GemDroid [34]	Yes	x86/Arm-v7A	ATTILA [9]	ARB	Custom	OpenGL	Cycle-Accurate	Execution-Driven	N/A[2]
GCN3 Simulator [13]	Yes	x86	AMD Pro A12-8800B APU	GCN3	Vendor	ROCM	Cycle-Accurate	Execution-Driven	˜42%
Our Simulator	Yes	Retargetable/ Arm-v7A/v8A	Retargetable/ Arm Mali-G71	Retargetable/ Native Binary	Vendor	Any/ OpenCL	Instruction-Accurate	Execution-Driven	0.0%

[1] Maximum error of a performance metric reported in the original publication.
[3] Original publication does not provide an accuracy evaluation against a hardware implementation of the simulated GPU.

TABLE IV: Feature comparison of existing GPU simulators. Our simulator is the only full-system CPU/GPU mobile platform simulator capable of hosting an unmodified GPU software stack and supporting true GPU native code execution.

GEM5-GPU is a configurable full-system simulator, it suffers from similar shortcomings as GPGPU-SIM. The GPU side of GPGPU-SIM does not accurately model a real GPU, and heavily relies on a simulator-specific software stack. ATTILA [9] is an execution-driven simulator targeting the academic ATTILA unified-shader GPU. While enabling research for GPU architectures and OPENGL application tuning, ATTILA does not model a real GPU and suffers from the lack of a full driver stack. GEMDROID [34] integrates ATILLA with the GEM5 architecture simulator, however this framework still lacks a realistic driver stack and GPU architecture. TEAPOT [32] is a trace-based GPU simulator, designed for the evaluation of mobile GPUs and has a cycle accurate GPU model for evaluating performance. TEAPOT supports OPENGL ES 1.1/2.0 and runs unmodified Android applications, but relies on the open-source GALLIUM3D drivers for a generic 'softpipe' GPU. [36] maps several guest GPUs onto the host system's GPU using multiplexing in a virtual platform. However, in this approach GPU kernels are intercepted at the CUDA API level, whereas our simulator executes actual Mali binary instructions. A full-system CPU/GPU simulation framework sharing some features with our simulator has been presented in [33], however their GPU model is a simplified and generic approximation. A microarchitectural simulator for Intel's integrated GPU has recently been described in [37], which relies on binary instrumentation of kernels for trace generation. While this allows for inspection of GPU code, insertion of tracing code modifies the GPU kernels and

interferes with their execution. Similarly, SASSI [38] provides binary instrumentation for NVIDIA kernels. [39] parallelizes GPGPU-SIM, but is limited by cycle-level synchronization points, unlike our functional simulator.

VII. Summary & Conclusion

In this paper we have presented the first ever fully retargetable full-system simulator supporting an unmodified software stack for a commercially available, state-of-the-art mobile GPU. Its validated instruction-accurate performance model enables more accurate insights into the GPU's operation than with simulators claiming cycle-accuracy for crudely approximated architectures and non-standard runtime environments. Our full-system approach will ensure a long-lasting simulator, requiring little maintenance as new toolchains are released. While we draw on several known simulation techniques, we have demonstrated the feasibility of accurate full-system CPU/GPU simulation at performance levels comparable to or better than those of existing, less accurate simulators. Our simulation approach enables us to gain insights into mobile GPU workloads including system-level transactions between the CPU and GPU - inaccessible using other GPU simulation approaches. Our simulator can characterize mobile GPU applications with accuracy unavailable using existing GPU simulators and provides a most useful tool to researchers and developers alike.

A. Future Work & Software Release

Future work will include 3D graphics support, further performance optimizations, e.g. JIT-compiled execution of GPU code, and micro-architectural performance modeling and simulation based design space exploration of machine learning and computer vision enabled mobile GPUs.

Our simulator has been made publicly available [40] to facilitate further research and development of mobile GPU architectures based on accurate simulation tools.

References

[1] W. Jia, K. A. Shaw, and M. Martonosi, "Stargazer: Automated regression-based GPU design space exploration," in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*, ser. ISPASS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 2–13. [Online]. Available: http://dx.doi.org/10.1109/ISPASS.2012.6189201

[2] A. Jooya, A. Baniasadi, and N. J. Dimopoulos, "Efficient design space exploration of GPGPU architectures," in *Proceedings of the 18th International Conference on Parallel Processing Workshops*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 518–527. [Online]. Available: http://dx.doi.org/10.1007/9783-642-36949-0_60

[3] G. Ceballos, A. Sembrant, T. Carlson, and D. Black-Schaffer, "Behind the scenes: Memory analysis of graphical workloads on tile-based GPUs," in *International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS, 2018.

[4] C. Bertolli, A. Betts, N. Loriant, G. R. Mudalige, D. Radford, D. A. Ham, M. B. Giles, and P. H. J. Kelly, "Compiler optimizations for industrial unstructured mesh CFD applications on GPUs," in *Languages and Compilers for Parallel Computing*, H. Kasahara and K. Kimura, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 112–126.

[5] W. Jia, E. Garza, K. A. Shaw, and M. Martonosi, "GPU performance and power tuning using regression trees," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, pp. 13:1–13:26, May 2015. [Online]. Available: http://doi.acm.org/10.1145/2736287

[6] D. A. Burgoon, E. W. Powell, and J. A. S. Waitz, *A Mixed C/Verilog Dual-Platform Simulator*. Boston, MA: Springer US, 2002, pp. 175–186. [Online]. Available: https://doi.org/10.1007/978-1-4757-6674-5_15

[7] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 407–420. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.12

[8] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of PTX kernels," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 3–12. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306801

[9] V. M. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. E, "ATTILA: a cycle-level execution-driven simulator for modern GPU architectures," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS-2006. IEEE, March 2006, pp. 231–241.

[10] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 335–344. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370865

[11] X. Gong, R. Ubal, and D. R. Kaeli, "Multi2Sim Kepler: a detailed architectural GPU simulator," in *Proceedings of International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS. IEEE, April 2017.

[12] S. Lee and W. W. Ro, "Parallel GPU architecture simulation framework exploiting work allocation unit parallelism," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software, Austin, TX, USA, 21-23 April, 2013*, ser. ISPASS. IEEE, 2013, pp. 107–117.

[13] A. Gutierrez, B. Beckmann, A. Dutu, J. Gross, J. Kalamatianos, O. Kayiran, M. LeBeane, M. Poremba, B. Potter, S. Puthoor, M. Wyse, J. Yin, A. Jain, T. Rogers, X. Zhang, and M. Sinclair, "Lost in abstraction: Pitfalls of analyzing GPUs at the intermediate language level," in *Proceedings of The 24th IEEE International Symposium on High-Performance Computer Architecture*, ser. HPCA, February 2018.

[14] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous CPU-GPU simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, Jan.-June 2015. [Online]. Available: doi.ieeecomputersociety.org/10.1109/LCA.2014.2299539

[15] G. Malhotra, S. Goel, and S. R. Sarangi, "GpuTejas: A parallel simulator for GPU architectures," in *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*. IEEE Computer Society, 2014, pp. 1–10. [Online]. Available: https://doi.org/10.1109/HiPC.2014.7116897

[16] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*. IEEE Computer Society, 2009, pp. 163–174. [Online]. Available: https://doi.org/10.1109/ISPASS.2009.4919648

[17] S. Collange, D. Defour, and D. Parello, "Barra, a parallel functional GPGPU simulator," in *18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS, 2010.

[18] J. Davies, "The Bifrost GPU architecture and the ARM Mali-G71 GPU," in *HotChips*, August 2016.

[19] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC architecture description language and tools," *International Journal of Parallel Programming*, vol. 33, no. 5, pp. 453–484, Oct 2005.

[20] H. Wagstaff, M. Gould, B. Franke, and N. Topham, "Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 21:1–21:6. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488760

[21] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[22] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, Mar. 2012.

[23] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[24] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. J. Kelly, A. J. Davison, M. Luján, M. F. P. O'Boyle, G. Riley, N. Topham, and S. Furber, "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2015, arXiv:1410.2167.

[25] T. Remmelg, T. Lutz, M. Steuwer, and C. Dubach, "Performance portable gpu code generation for matrix multiplication," in *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. ACM, 2016, pp. 22–31.

[26] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 100–112.

[27] C. Nugteren, "myGEMM," https://github.com/cnugteren/myGEMM, GitHub Repository (accessed 2018-07-30).

[28] ——, "Clblast: A tuned opencl blas library," *arXiv preprint arXiv:1705.05249*, 2017.

[29] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 316–331. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173177

[30] J.-H. Ding, W.-C. Hsu, B. Jeng, S.-H. Hung, and Y.-C. Chung, "HSAemu - a full system emulator for HSA platforms," *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 1–10, 2014.

[31] H.Kim, J.Lee, N.B.Lakshminarayana, J.Sim, J.Lim, and T.Pho, "MacSim: A CPU-GPU heterogeneous simulation framework," Georgia Institute of Technology, Tech. Rep., 2012.

[32] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "TEAPOT: A toolset for evaluating performance, power and image quality on mobile graphics systems," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 37–46. [Online]. Available: http://doi.acm.org/10.1145/2464996.2464999

[33] P. Wang, G. Liu, J. Yeh, T. Chen, H. Huang, C. Yang, S. Liu, and J. Greensky, "Full system simulation framework for integrated CPU/GPU architecture," in *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test, VLSI-DAT 2014, Hsinchu, Taiwan, April 28-30, 2014*. IEEE, 2014, pp. 1–4. [Online]. Available: https://doi.org/10.1109/VLSI-DAT.2014.6834872

[34] N. Chidambaram Nachiappan, P. Yedlapalli, N. Soundararajan, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "GemDroid: A framework to evaluate mobile platforms," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '14. New York, NY, USA: ACM, 2014, pp. 355–366. [Online]. Available: http://doi.acm.org/10.1145/2591971.2591973

[35] J. Cui, K. Mei, D. Liu, and B. Li, "Construction of embedded Mali GPU simulator for OpenCL," *Hsi-An Chiao Tung Ta Hsueh/Journal of Xi'an Jiaotong University*, vol. 49, pp. 20–24, February 2015.

[36] Y. Jung and L. P. Carloni, "ΣVP: Host-GPU multiplexing for efficient simulation of multiple embedded GPUs on virtual platforms," in *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: ACM, 2015, pp. 106:1–106:6. [Online]. Available: http://doi.acm.org/10.1145/2744769.2744913

[37] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C.-K. Luk, "Performance characterisation and simulation of Intels integrated GPU architecture," in *International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS, 2018.

[38] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 185–197.

[39] S. Lee and W. W. Ro, "Parallel gpu architecture simulation framework exploiting work allocation unit parallelism," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 107–117.

[40] https://gensim.org/simulators/gpusim, Binary Release (accessed 2019-02-12).