공학학사학위논문

Modeling the GPU Instruction Scheduling Performance using Microbenchmarks

(마이크로벤치마크를 통한 GPU 명령어 스케줄링 성능 모델링)

2023 년 2 월

서울대학교 공과대학

전기·정보공학부

김 형 주

Abstract

Modeling the performance of GPU is necessary to quickly grasp its mechanism, and instruction scheduling plays a key role in determining the performance of kernels. However, little work has been done to provide the analytical model of the GPU execution pipeline. This paper proposes the analytical model of the GPU instruction scheduling mechanism using microbenchmarks to inspect the architectural details of simulators and real devices. The microbenchmark implementation GPUDiag successfully extracted the scheduling properties, and the model predicted the kernel latency with a high correlation of $r = 0.99$. Moreover, GPUDiag found two discrepancies between gem5 and AMD RDNA architecture.

Keywords: GPU, Performance Modeling, Microbenchmark, Instruction Scheduling

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Accurate modeling of the GPU instruction execution mechanism is necessary to understand its performance variation and complex structure. Application developers rely on the programming model provided by the manufacturer to ensure the correctness and optimize the performance of their code. Researchers devise novel architectural ideas on GPUs by modeling specific components and investigating their impact on performance. They evaluate their ideas using GPU simulators [1, 2, 3, 4], which are the comprehensive models of the GPU instruction execution pipeline.

Nonetheless, the hidden details of GPU implementation make it difficult to model its properties. GPU manufacturers disclose the least amount of low-level information, and the public can only know the programming interface and a few tips on performance tuning. As a result, researchers cannot reflect the actual mechanism of GPU components in their models and simulators. Moreover, their architecture is becoming more complex to achieve better computation throughput. This complexity makes the researchers and engineers more difficult to fully understand and exploit the latest low-level structures.

To catch up with the architectural change and grasp deeper mechanisms, researchers have relied on various microbenchmarks. Mei *et. al.* [5] used the pointer-chase algorithm [6] to inspect the properties of GPU cache and memory hierarchy. This pointer-chase microbenchmark was

used by Jia *et. al.* [7, 8] to identify the comprehensive details of NVidia Volta and Turing architectures. In addition, Accel-Sim [2] integrated microbenchmarks to automatically reverse-engineer the GPU, tune its model parameters and improve the accuracy of GPGPU-Sim [1].

However, previous works could not provide an effective method to understand and estimate the performance impact of GPU components. Previous microbenchmarks mainly targeted investigating the memory hierarchy of GPUs [5, 7, 8], whereas other GPU-specific instruction scheduling mechanisms were ignored. Simulators modeled the mechanism as a form of codes [2, 4], yet it is difficult to understand the performance variation of architectural modification without time-consuming simulations. Also, they do not support microbenchmarks, which makes it impossible to compare their accuracy with real devices.

To this end, this paper proposes an analytical model of GPU instruction scheduling mechanism and microbenchmarks that can measure its parameters. The complex structures of GPGPU-Sim [1, 2] and gem5-APU [3, 4] are simplified into three hierarchical components: limit checker, block scheduler, and warp scheduler. Then, the instruction scheduling performance of each component is modeled as a set of analytical equations. Using the formulas, microbenchmarks are designed to calculate the model parameters back from measurable values.

Furthermore, the model and microbenchmarks are used to suggest possible improvements to the latest GPU simulators. This work presents GPUDiag, [1] an automatic GPU reverse-engineering software that implements the designed microbenchmarks. Because the analytical model represents the mechanism of simulators, the discrepancy of evaluation results of GPUDiag on simulators and real devices indicates the inaccuracy of the former. Consequently, this work found that gem5 [4] fails to simulate the warp and block scheduling policy of AMD RDNA [9] architecture.

Section 2 introduces the background on the programmer's view, modeling, and microbenchmark works of GPUs. Then, section 3 demonstrates the analytical model and microbenchmarks. They are implemented and evaluated in section 4.

[1] https://github.com/kimhj1473/gpudiag

Chapter 2

Background and Related Works

This chapter introduces the background and previous works on GPU modeling. Section 2.1 introduces the architecture and programming model of GPUs, and the prior works about modeling and simulating GPU architecture are listed in section 2.2. Then, section 2.3 presents the related works on uncovering the low-level structure of GPUs using microbenchmarks.

2.1 GPU Architecture and Programming Model

GPU accelerates massively parallel programs with SIMD architecture [10]. User-specified number of threads constitute a program, called the kernel, with hierarchical groups. The set of all threads in a kernel is called a grid, and the maximum unit of synchronizable threads is called a block [10, 11]. Hence, the size of a kernel is the block dimension B multiplied by the number of blocks in a grid G. They are both configurable by the programmer.

Each block consists of several warps. Warp is the group of w threads that are allocated to the same SIMD functional unit and always executed at the same time. Each warp has its own program counter, while each thread needs separate data registers and ALUs [10, 11, 9].

GPUs fetch the data from the global memory and store them on the multi-level cache hierarchy. Their register file acts as the top-level cache to temporarily store the data just before directing it to the functional units. The data can also be stored in the shared memory, which

has access latency comparable to and capacity much larger than the register file [10, 3].

Developers can program GPU devices using runtime libraries provided by manufacturers. Nvidia provides CUDA [10] and AMD supports HIP [12]. Using these environments, programmers can easily design kernels with high-level language and manage their execution [10, 12].

2.2 GPU Models and Simulators

Compared to the CPU, GPU is a multicore processor with plenty of unique characteristics. Many works have attempted to model them to better understand the performance variations. The ideal throughput of multicore processors is determined by the arithmetic intensity of application as a roofline model [13]. Later, it is applied to GPU-specific applications to estimate their exact peak throughput [14]. Considering the programming model provided by CUDA [10], the GPU scheduling performance was roughly modeled into simple equations [15]. Zorua [16] discovered the relation between the GPU performance cliffs and hardware resources.

However, none of the mentioned works sufficiently investigated the performance impact of the entire instruction execution process. The roofline models [13, 14] ignore the architectural details and only focus on the operation throughput. The model based on CUDA [15] cannot explain the performance parameters with the hardware properties, and Zorua [16] does not provide any analytical explanation.

To overcome the shortcomings of analytical models that fail to handle practical applications, approaches based on simulations are also proposed. From the given instruction stream, its static analysis and analytical model are used to predict the kernel latency [17, 18]. GPGPU-Sim [1] and gem5-APU [3] implement the comprehensive instruction execution mechanism into a simulator to analyze the performance impact of each component. Although these methods are effective to investigate real workloads, using simulations to evaluate the performance impact is a time-consuming task and does not provide insight.

2.3 GPU Microbenchmarks

Microbenchmarks are widely used to extract the target feature from complex computer systems. Previous works that inspected the GPU using microbenchmarks are mainly focused on its memory hierarchy. The stride memory access latency pattern can be used to deduce the properties of cache hierarchy [19]. The fine-grained method improved its accuracy by storing the latency of each access into the GPU-specific shared memory [5]. The method is also used to dissect the cache structure of the latest Nvidia GPUs [7, 8].

Besides the memory properties, microbenchmarks are used to measure the other characteristics of GPU. Many works used microbenchmarks to measure the throughput and latency of different types of instructions [19, 20, 17, 14, 2]. The warp scheduling policy [19] and kernel scheduling policy [21] are also measured using the microbenchmarks.

However, these works fail to extract architectural properties other than the memory hierarchy. Because the throughput measurements are not based on the architectural model, microbenchmark results are not used to further estimate the hardware details. Moreover, many novel technologies are applied to commercial GPUs since the measurements are produced, but the microbenchmarks cannot effectively measure the new features.

Some works aggregated the set of microbenchmarks to systematically extract the architectural features from real devices. DiagSim [22] built a complete dependency tree of CPU microbenchmarks and applied them to easily verify the integrity of the simulators. Accel-Sim [2] developed an automatic tool that runs GPU microbenchmarks and tunes the configuration of the GPU simulator to improve its accuracy. Nonetheless, no work has investigated the comprehensive mechanism and performance impact of GPU instruction scheduling using microbenchmarks.

Chapter 3

Modeling and Microbenchmarks

This chapter demonstrates the design of the analytical model and microbenchmarks. In section 3.1, the instruction scheduling components of GPU simulators [1, 2, 3, 4] are simplified as a parameterized model, and its impact on kernel performance is represented as equations. The equations are used to design the microbenchmarks in section 3.2. They extract the model parameters, which reflect the architectural properties, by measuring their latency.

3.1 Analytical Model

The instruction scheduling and execution mechanism implemented in GPGPU-Sim [1, 2] and gem5-APU [3, 4] is modeled into three hierarchical components, which are illustrated in Figure 3.1. The model consists of three different parts: the kernel limit checker, the block scheduler with state buffers, and the warp scheduler with functional units. When the kernel is launched into the hardware, its size and resource requirements are examined by the limit checker. If the requirement exceeds a certain limit, the launch will fail with an error. Next, the block scheduler allocates the blocks in the kernel to the state buffers. Because the amount of resources is limited, the number of blocks that can be allocated simultaneously is limited to N_{slot}. Lastly, the warp scheduler dispatches the warps in the allocated blocks to the appropriate functional unit.

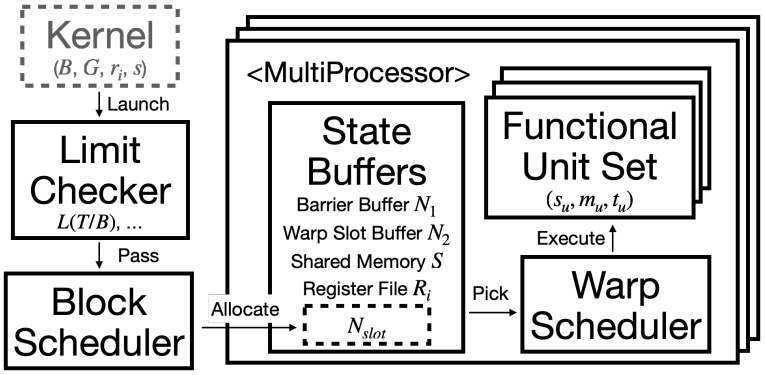


Figure 3.1: Diagram of the GPU Analytical model. The kernel resource usage is checked by the limit checker and scheduled to the state buffers. The allocated warps are further scheduled to the functional units. Each component is described by a series of parameters.

3.1.1 Kernel Resource Usage and Limit Checker

The kernel resource usage is expressed as a set of parameters. The number of blocks per grid is G, and the number of threads per block is B. Hence, the number of warps per block is $b = \lceil B/w \rceil$. Each block uses s bytes of shared memory and each warp uses r_i type-i registers. Different types of registers are allowed to represent the scalar and vector registers in AMD GPUs.

The limit checker examines the validity of the resource request from the kernel by comparing the request with the architectural constraints. If we denote $L(T/B)$ the limit of threads per block, $L(T/G)$ the limit of threads per grid, $L(S/B)$ the limit of shared memory per block, $L(R_i/T)$ the limit of registers per thread and $L(R_i/B)$ the limit of registers per block, the limit checker is modeled with the equation 3.1. [1] [2]

$$\begin{cases} B \le L(T/B),\ BG \le L(T/G) \\ \lceil s, s_{min} \rceil \le L(S/B) \\ \forall i,\ r_i \le L(R_i/T),\ b\lceil r_i, r_{i,min} \rceil \le L(R_i/B) \end{cases} \tag{3.1}$$

3.1.2 Block Scheduler and State Buffers

Kernel Latency from the Block Scheduler When the kernel size meets the constraint provided by the limit checker, the block scheduler distributes the blocks in the kernel to n_{mp}

[1] $\lceil a, b \rceil$ and $\lfloor a, b \rfloor$ means a rounded up or down in the unit of b, thus $\lceil a/b \rceil b$ and $\lfloor a/b \rfloor b$ respectively.
[2] These conditions are selected from the "Technical Specifications per Compute Capability" table in [10].

multiprocessors, each with N_{slot} block slots. As shown in Figure 3.1, a GPU has multiple identical multiprocessors that can execute the allocated blocks independently. Each multiprocessor executes total $g = \lceil G/n_{mp} \rceil$ blocks, where N_{slot} blocks can be allocated simultaneously. As a result, a multiprocessor schedules N_{slot} blocks $\lfloor g/N_{slot} \rfloor$ times, and then schedules the remaining $g \bmod N_{slot}$ blocks. If we assume $fu(c)$ the relative time of executing c warps, the former takes $fu(bN_{slot})$ time while the remaining latter requires $fu(b(g \bmod N_{slot}))$. In sum, the relative kernel latency is expressed as equation 3.2.

$$T(g, b) = \lfloor \frac{g}{N_{slot}} \rfloor fu(bN_{slot}) + fu(b(g \bmod N_{slot})) \tag{3.2}$$

N_{slot} from the State Buffers The number of concurrently schedulable blocks N_{slot} is determined by the number of available hardware resources in the state buffers. According to Figure 3.1, four types of buffers affect N_{slot}: barrier buffer, warp slot buffer, shared memory, and register files. The barrier buffer has N_1 storages for the intra-block synchronization status, and the warp slot buffer keeps the execution context of N_2 warps. Also, the shared memory stores S bytes, and the type-i register file has R_i registers. As a result, the resource requirement of N_{slot} concurrent blocks must stay below the size of the buffers, and it determines the N_{slot} as equation 3.3. [3] Note that s_{min} and $r_{i,min}$ refer to the allocation granularity of shared memory and register file, respectively.

$$N_{slot}(b, s, r_i) = \min \left(N_1, \lfloor \frac{N_2}{b} \rfloor, \lfloor \frac{S}{\lceil s, s_{min} \rceil} \rfloor, \lfloor \frac{R_i}{\lceil r_i, r_{i,min} \rceil b} \rfloor \right) \tag{3.3}$$

3.1.3 Warp Scheduler and Functional Units

Role of the Warp Scheduler After the block scheduler allocates blocks to the state buffers, the warp scheduler dispatches the warps in the blocks to appropriate functional units, such as ALUs and load-store units. Hence, the relative latency of executing c allocated warps $fu(c)$ depends on the properties of functional units participating in the given kernel. This paper assumes that the warp scheduler schedules the instructions to optimize for functional unit utilization. Simple round robbin policy applies well, as it equally distributes the warps to entire functional

[3] This equation summarizes the behavior described in "CUDA Occupancy Calculator" [10] and gem5 [3, 4].

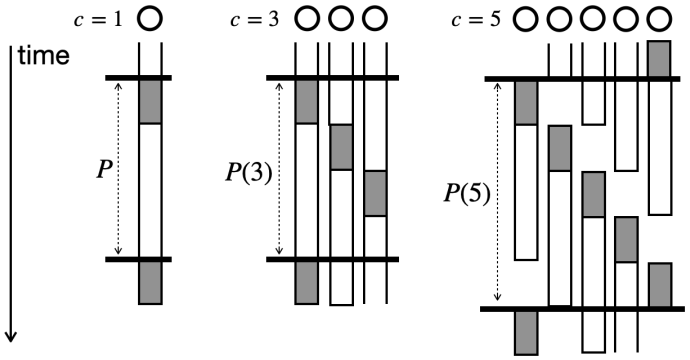


Figure 3.2: The idea of the functional unit formula. Two functional units participate in the kernel, $t_1 = 1$, $t_2 = 3$, $m_1 = 1$, $m_2 = \infty$ and they are dependent to each other. At $c = 3$, functional units are pipelined and P does not increase. At $c = 5$, P increased to $t_1 c/m_1$ and the other unit operation does not affect the period as taking the vacant time slots is sufficient for it.

units. Nonetheless, the real GPU can adopt other scheduling policies to optimize the cache hit ratio for memory-intensive applications [23]. In this case, the measured $fu(c)$ can be larger than the values estimated in the rest of this subsection.

The warp scheduler is also in charge of directing each type of instruction to the corresponding functional units, so the instruction stream should also be considered. To focus on the repeated nature of practical workloads, this paper models the instruction stream of the kernel as repeating $\{insts\}$ instructions $N_{period} \gg 1$ times. Each repetition period requires latency $P(c)$ if the kernel consists of 1 block with c warps. Then, the $P_1 = P(1)$ is the property of $\{insts\}$ as neither hardware replication nor thread parallelism affects its value. Using this kernel specification, $fu(c)$ is precisely defined using the kernel execution time $T(G, b)$ as equation 3.4.

$$fu(c) := \lim_{N_{period} \to \infty} \frac{T_{\{insts\}}(G = 1, b = c)}{T_{\{insts\}}(G = 1, b = 1)} \tag{3.4}$$

$fu(c)$ from the Functional Units The execution time $fu(c)$ is decided from the latency and replication of functional units. Each functional unit $u \in FU$ is modeled with three parameters. t_u is the latency, or more precisely the inverse throughput of the unit, considering its pipelined implementation. $m_u s_u$ is the number of replicated units in a multiprocessor. They are partitioned into s_u groups, each of which has a group of dedicated warps. The warps are not scheduled for the units that reside in the other group.

Assume there is a bottleneck unit in the set of functional units required for executing $\{insts\}$.

Table 3.1: List of microbenchmarks. Seven tests and their inputs, outputs, and corresponding model components are described. The variables follow the notations introduced in section 3.1.

Test Name	Inputs	Outputs	Component
kernel limits	w	$\forall L(X/Y)$	Limit checker
functional units	w, $L(T/B)$	P, $fu(c)$, thruput_{bot} for each $\{insts\}$	Functional units
num mp	w, $fu_{br}(c)$	n_{mp}	Functional units
warpsched policy	w, $L(T/B)$	warp scheduling policy	Functional units
warpslot buffer	w, $L(T/B)$, n_{mp}	N_1, N_2, $N_{slot}(b)$	State buffers
shmem buffer	$L(S/B)$, n_{mp}, $N_{slot}(1)$	S, s_{min}	State buffers
regfile buffer	w, $L(T/B)$, $L(R_i/T)$, $L(R_i/B)$, n_{mp}, $N_{slot}(b)$	R_i, r_{min}	State buffers

According to Figure 3.2, $P(c)$ stays constant with small c due to the dependencies between the participating units. As c grows, $P(c)$ also increases because the instruction throughput $c/P(c)$ cannot exceed the bottleneck throughput m_{bot}/t_{bot}. These two factors act as lower bounds and make equation 3.5. Note that non-bottleneck units do not affect the time, as they can be scheduled in the empty time slots guaranteed by the bottleneck condition.

$$P(c) = \max\left(P_1, \frac{t_{bot}c}{m_{bot}}\right), \text{ where } bot = \text{argmin}_u\left(\frac{m_u}{t_u c}\right) \tag{3.5}$$

We can now derive $fu(c)$ using the above equation. Because we focus on the $N_{period} \to \infty$ case as in equation 3.4, the edge effects in Figure 3.2 disappears and $fu(c)$ becomes $P(c)/P_1$. In addition, as each unit in s_u groups can operate only on the dedicated warps, the maximum number of warps allocated to each group becomes $\lceil c/s_u \rceil$. Thus, equation 3.5 transforms into equation 3.6.

$$fu(c) = \max_u\left\{1, \frac{t_u}{m_u P_1}\lceil\frac{c}{s_u}\rceil\right\} \tag{3.6}$$

3.2 Microbenchmarks

The parameters of the analytical model of instruction scheduling demonstrated in section 3.1 are measured with the microbenchmarks listed in Table 3.1. Inputs are provided by the user or taken from the previous outputs, where each test produces the measured parameter values as its outputs. Each test is related to one of the three model components described in section 3.1. By executing them in order, the model parameter values can be retrieved.

```
__kernel__ measure_fu(G, B, insts, Nperiod) {
    __syncthreads(); // if G > 1, must sync with global memory
    start = clock();
    #pragma unroll Nperiod
    for (i=0; i<Nperiod; i++) asm(insts);
    __syncthreads(); // if G > 1, time each block and find max
    return (clock() - start);
}
```

Figure 3.3: Pseudocode of the functional units test kernel. It measures the time to execute the instruction repeated by N times. Proper synchronization between threads should be done.

3.2.1 Kernel Limits Test

The limit checker parameters are measured by launching kernels with various resource requirements and checking the status. Failed launch implies that the kernel resource usage exceeds the limit. Using the kernel that its B, G, s, and r_i are controllable, $L(T/B)$, $L(T/G)$, $L(S/B)$, and $L(R_i/T)$ are measured by finding the failure point with binary searching. $L(R/B)$ is measured by finding the maximum r_i value for each possible B and calculate $L(R_i/B) = \max_B(r_i(B)B/w)$.

3.2.2 Functional Units Test

The functional units are investigated by launching the instruction repeating kernel described in section 3.1.3 and Figure 3.3. The given $\{insts\}$ are repeated N_{period} times and all threads should be properly synchronized before and after the instructions. The threads from different blocks should synchronize themselves through the global memory. In addition, the code in figure 3.3 should be executed twice to warm up the instruction cache.

The three tests related to functional units shown in Table 3.1 each uses the variant of `measure_fu()`. First, funcional units test is done by launching the kernel with $(G, b) = (1, c)$. $P(c)$ is the time value at c divided by N_{period} if N_{period} is sufficiently large. Hence, $P_1 = P(c)$ and $fu(c) = P(c)/P_1$. Because $G = 1$, no additional synchronization is necessary. To investigate the hardware deeper, the maximum throughput of the bottleneck unit per multiprocessor is

calculated from the $fu(c)$ data using the equation 3.7, which is derived from equation 3.6.

$$\text{throughput}_{bot} = \frac{m_{bot} s_{bot}}{t_{bot}} = \frac{\max_c (c/fu(c))}{P_1} \tag{3.7}$$

Next, the n_{mp} is measured by finding G that doubles the execution time. From the equation 3.2, the change of the $T(g, b)$ from $g = 1$ to 2 is described as equation 3.8.

$$\frac{T(g=2,b)}{T(g=1,b)} = \begin{cases} \frac{fu(2b)}{fu(b)} & (N_{slot}(b) \geq 2) \\ 2 & (N_{slot}(b) = 1) \end{cases} \tag{3.8}$$

If $fu(2b) \approx 2fu(b)$, the kernel execution time becomes twice when g changes from 1 to 2 regardless of N_{slot}. According to the definition of g, this point is where G becomes $n_{mp}+1$ from n_{mp}. As a result, using $c_{2\times}$ that satisfies $fu(2c_{2\times}) \approx 2fu(c_{2\times})$, n_{mp} can be found by executing the kernel with $b = c_{2\times}$, increasing G from 1 and find the point where the time gets doubled. The $c_{2\times}$ can be found from the measured $fu(c)$ data.

Note that the num mp test requires $G > 1$, so the kernel in figure 3.3 should be synchronized using the global memory semaphore. Because it is not the hardware-supported barrier method, the temporal inaccuracy may exist. However, the error is amortized for sufficiently large N_{period}.

Lastly, the warp scheduling policy can be inferred from the data of scheduled warp ids at each time step. This can be measured by replacing `clock()` lines of Figure 3.3 with `asm(log_warp_id)` and launch it with $(G, B) = (1, L(T/B))$. The measured time values will be temporarily stored on different registers and copied to the memory later.

3.2.3 State Buffers Test

The properties of state buffers are examined by measuring N_{slot} of kernels with varying resource usages and using equation 3.3. Figure 3.4 describes the kernel that can measure N_{slot}. The global memory semaphore `sync` makes deadlock on $g > N_{slot}$, because blocks more than N_{slot} cannot be scheduled to the same multiprocessor at the same time as described in subsection 3.1.2. Hence, N_{slot} is measured by finding the maximum value of g that makes the kernel terminates for given (B, s, r_i).

The three state buffer tests in Table 3.1 use the N_{slot} measurements to calculate their outputs.

```
__kernel__ measure_nslot(G, B, s, ri, sync) {
    // *sync is assumed to be initialized as 0
    if (thread_idx == 0) atomicAdd(sync, 1);
    while(*sync < G)
        if (clock() > timeout) return TIMEOUT;
    return SUCCEED;
    use_resources(s, ri); // never executed
}
```

Figure 3.4: Pseudocode of the state buffers test kernel. It measures N_{slot} by synchronizing the blocks through the global memory and checks if all blocks can be executed concurrently. The resource usage can be controlled by the never executed code segment.

Using additional assumptions on b, s, and r_i, equation 3.3 is converted into below equations.

$$N_{slot,wsb}(b) = \min\left(N_1, \lfloor \frac{N_2}{b} \rfloor\right) \qquad (s = 0, r_i b \ll R_i) \tag{3.9}$$

$$N_{slot,shm}(s) = \min\left(N_{slot,wsb}(1), \lfloor \frac{S}{\lceil s, s_{min} \rceil} \rfloor\right) \qquad (b = 1, r_i b \ll R_i) \tag{3.10}$$

$$N_{slot,rf,i}(b, r_i) = \min\left(N_{slot,wsb}(b), \lfloor \frac{R_i}{\lceil r_i, r_{i,min} \rceil b} \rfloor\right) \qquad (s = 0) \tag{3.11}$$

From the equations, the warp slot test calculates N_1 and N_2 with equation 3.12, the shmem buffer test measures S, and the regfile buffer test estimates R_i with equation 3.13.

$$N_1 = N_{slot,wsb}(1),\ N_2 = \max_b(bN_{slot,wsb}(b)) \tag{3.12}$$

$$S = \max_s(sN_{slot,shm}(s)),\ R_i = \max_{b,r_i}(r_i bN_{slot,rf,i}(b, r_i)) \tag{3.13}$$

Measuring s_{min} and $r_{i,min}$ is possible by finding the minimum value that makes equation 3.10 and 3.11 consistent with the measured data. S and R_i are calculated with equation 3.13 using max instead of x_{min} values, for $x \in \{s, r_i\}$. Using these values, the measured LHS values of equation 3.10 and 3.11 are compared with the re-calculated RHS values, assuming certain x_{min}. The correct x_{min} is the first value that makes both sides consistent.

Chapter 4

Implementation and Evaluation

In this chapter, the analytical model and microbenchmarks designed in chapter 3 are implemented into GPUDiag and their efficacy is evaluated. Section 4.1 explains the system design of GPUDiag, and section 4.2 describes the setup of evaluation. The result is illustrated and discussed in section 4.3.

4.1 Microbenchmark Implementation

Figure 4.1 describes the structure of GPUDiag. The microbenchmark algorithms are implemented on top of the GPU runtime. While the CPU-side host codes are directly implemented for each test, the GPU-side kernel codes are synthesized by the kernel generator for flexible test execution. For the portability of the system, a thin layer of API wrapper is introduced between the microbenchmarks and the runtime. Because GPUDiag uses GPU runtime APIs and binary utilities, the microbenchmarks are able to be executed on both real devices and simulators.

Each microbenchmark in Table 3.1 is implemented by inheriting the test template. The test template provides the necessary interfaces to generate, compile, execute codes, measure data, and summarize the results. The shell commands required to manipulate GPU binaries are also abstracted so that the template can utilize appropriate commands for different microbenchmarks and environments.

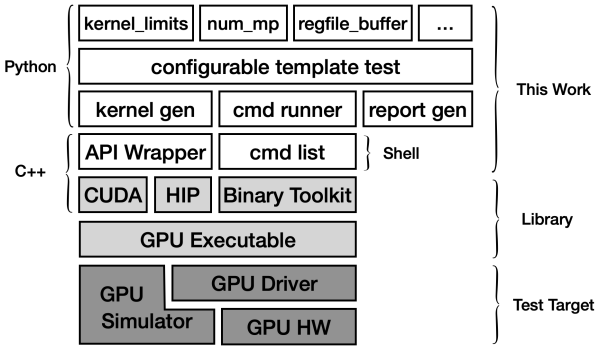


Figure 4.1: Software stack of GPUDiag. The white boxes are implemented on top of the runtime and binary utilities provided by the manufacturer. Template test generates kernel source codes, runs the command, and generates the report.

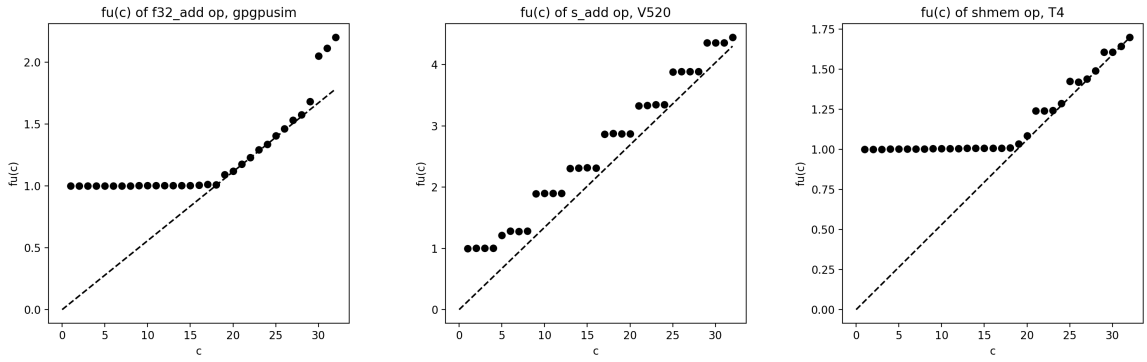
Table 4.1: GPU targets used for evaluation. Two simulators and two devices from Nvidia and AMD are used to evaluate the design with various environments.

Target	**GPU μArch**	**Simulator**	**Runtime**
GPGPU-Sim [2]	Nvidia Turing	yes	CUDA
gem5-APU [4]	AMD GCN3	yes	HIP
Tesla T4	Nvidia Turing	no	CUDA
Radeon V520	AMD RDNA1	no	HIP

GPUDiag executes the series of microbenchmarks automatically, considering their dependencies described as inputs and outputs of Table 3.1. The microbenchmark execution is configurable by the separate config script, and the measurements and their visualization are organized into report files.

4.2 Evaluation Environment

GPUDiag is evaluated on several environments listed in Table 4.1. They include simulators and real devices for Nvidia and AMD. Two simulators, gpgpusim[2] and gem5-APU[3], are used to verify the accuracy of the analytical model and microbenchmarks designed in chapter 3. Then, GPUDiag is evaluated on the real devices to compare the model with the off-the-shelf



(a) F32 add on GPGPU-Sim [2] (b) Scalar add on V520 (c) Shared memory on T4

Figure 4.2: Typical functional unit test results. Measured $fu(c)$ values are plotted for each instruction. Their shapes agree with the equation 3.6.

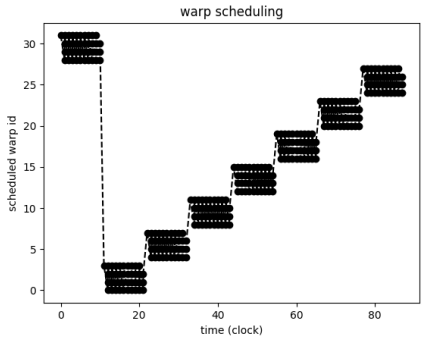
GPUs. Nvidia Tesla T4 is from the AWS g4dn.xlarge instance, while AMD Radeon V520 is from the AWS g4ad.xlarge instance. The measured parameter values are compared with the official specification, and the shapes of data plots are compared with the prediction of the model equations.

4.3 Evaluation Results

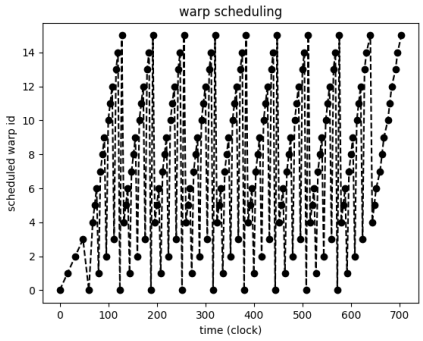
This section explains the evaluation results of GPUDiag. Subsection 4.3.1 describes the results of functional unit tests in subsection 3.2.2 and compare them with the model in subsection 3.1.3. Then, subsection 4.3.2 lists the results of state buffer tests in subsection 3.2.3 and match them with the model in subsection 3.1.2. Both subsections introduce the discovered discrepancies between simulators and hardware at the end of each subsection. Finally, the aggregated model accuracy is evaluated by inspecting the end-to-end latency measurements and equation 3.2 in subsection 4.3.3.

4.3.1 Warp Scheduler Model Evaluation

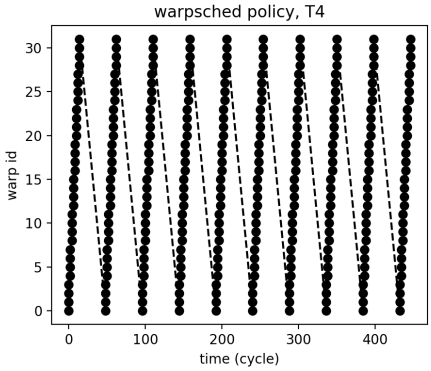
Functional Units Test Results The main goal of functional units test in Table 3.1 is to measure $fu(c)$ and compare with equation 3.6. Some typical shapes of data point graphs are



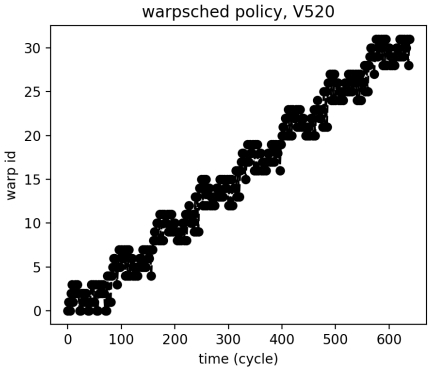
(a) GTO [23] policy on GPGPU-Sim [2]



(b) OF policy on gem5-APU [3]



(c) Result on T4



(d) Result on V520

Figure 4.3: Typical warpsched policy test results. IDs of warps scheduled at each time slot are plotted. The difference in their shapes reflects that they have different policies.

shown in Figure 4.2. According to Figure 4.2a, $fu(c)$ is 1 for small cs and increases linearly as c increases, which coincides with equation 3.6. Figure 4.2b exhibits the step function shaped plot, which follows the equation if $s_u = 4$. The last figure can be explained if the effects of two different functional units overlap. The equation 3.6 assumes that only the maximum value of $t_u\lceil c/s_u \rceil m_u$ among the units related to the instruction is exposed as $fu(c)$. As a result, if two different functional units have similar throughput and different s_u, two graphs can overlap. Figure 4.2c shows two functional units with $s_u = 1$ and $s_u = 4$ are competing for the bottleneck condition.

Warpsched Policy Test Results Next, the warp scheduling policy is measured using the method described in section 3.2.2. Figure 4.3 shows the warp IDs scheduled at each cycle. Oldest

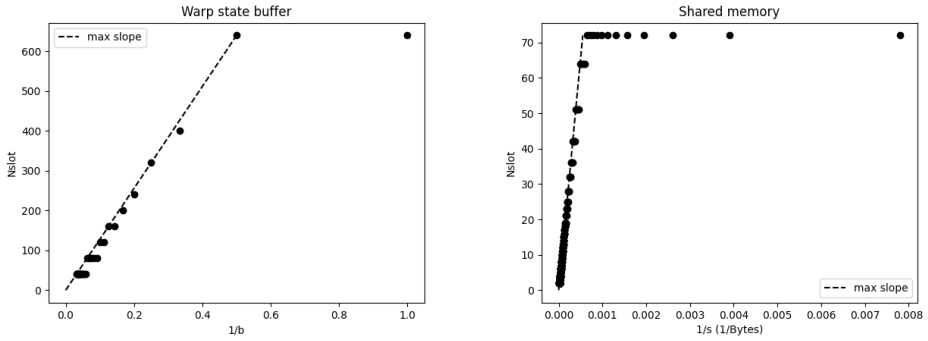
first(OF) and round robbin(RR) policy supported by gem5 [3] shows similar patterns with each other as in Figure 4.3b, while greedy than oldest(GTO) [23] policy of GPGPU-Sim [2] displays peculiar pattern of Figure 4.3a. Tesla T4 in Figure 4.3c exhibits the round-robbin policy, whereas Radeon V520 in Figure 4.3d shows the pattern of GTO policy.

Warp Scheduler Discrepancy of V520 from gem5 As discussed in the above paragraph, Radeon V520 exhibits a different warp scheduling policy from the ones implemented in gem5 [3]. Two policies supported by gem5 are round robbin and oldest first [3, 4], and neither shows the pattern depicted in Figure 4.3d, which is more similar to the GTO [23] policy in Figure 4.3a. Consequently, gem5 [4] should add support for the GTO warp scheduling policy to accurately simulate the latest RDNA architecture of AMD GPUs.

4.3.2 Block Scheduler Model Evaluation

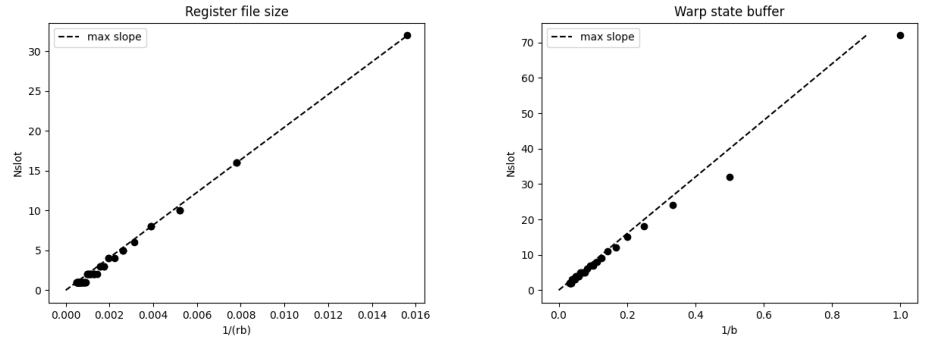
State Buffer Tests Results The state buffer microbenchmarks described in subsection 3.2.3 are executed on different environments. Figure 4.4 shows the N_{slot} values at each resource usages, plotted against the inverse of each usage to visualize the linear relation estimated by equation 3.9, 3.10, and 3.11. According to the equations, N_{slot} governed by a resource constraint is proportional to the inverse of the resource usage. This is clearly observed in Figure 4.4a, 4.4b, and 4.4c. The maximum slope visualized as dotted lines represents the state buffer size as described in equation 3.12 and 3.13. Moreover, the data points exhibit step function shapes below the dotted lines, which is explicable using the rounding and x_{min} of equation 3.3. The upper bound of figures 4.4a and 4.4b represents the other limit, such as N_1.

All other parameters are measured accurately using the method from section 3.2.3, except the warpslot buffer parameters on Radeon V520. The measured number of concurrent warps is not divided by n_{mp}. The values at large b agree with the model, but N_{slot} was smaller than expected on small bs. The reason might be an unexpected deadlock condition related to the load store units, or an unpredicted behavior of the block scheduler.



(a) Warpslots on T4

(b) Sharedmem on V520

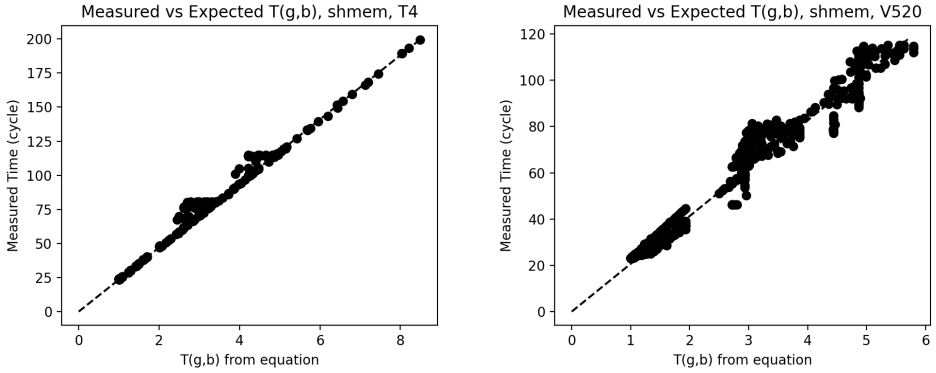


(c) Regfile on GPGPU-Sim [2]

(d) Warpslots on V520

Figure 4.4: Typical state buffer test results. N_{slot} is measured on each resource usage and plotted against the inverse of each usage. (a - c) can be explained using equation 3.3, whereas (d) graph exhibits aberrant data points.

Block Scheduler Discrepancy of V520 from gem5 However, Figure 4.4d exhibits different behavior from the other three plots. Although the data points roughly follow the linear relation, they become farther away from the dotted line as $1/b \to 1$. Equation 3.3 is not able to explain the behavior. Moreover, the number of simultaneously schedulable blocks measured by `measure_nslot()` in Figure 3.4 is not divisible by n_{mp} on some points, which strictly violates the assumption that the multiprocessors in a GPU are identical. Accordingly, because the model based on the mechanism of gem5 [4] fails to explain warp slots characteristic of V520, further research is required to model the block scheduling policy of AMD RDNA architecture and apply it to the simulator.



(a) $T(g, b)$ on T4

(b) $T(g, b)$ on V520

Figure 4.5: Evaluating the accuracy of the performance model. Execution time of the shared memory instruction kernel is plotted against the expected $T(g, b)$ calculated from equations 3.2, 3.3 and measured $fu(c)$ values. Both cases fit well with the equation.

4.3.3 End-to-End Performance Accuracy

Lastly, the overall accuracy of the model is evaluated by comparing the equation 3.2 with the measured kernel latency. Figure 4.5 shows the typical result from the shared memory instruction repeating kernel. The graphs plot the measured end-to-end execution time against the computed prediction from equation 3.2, equation 3.3, and the $fu(c)$ values from section 4.3.1. The result shows that the equations from section 3.1 are able to effectively estimate the actual execution time of the kernel on real GPUs. For both Tesla T4 and Radeon V520, the plot shows a strong positive correlation of $r = 0.992$ and 0.984, respectively.

Although Figure 4.5 shows the accuracy of the model, the equation fails to estimate the execution time on some conditions. Figure 4.5a has data points that their measured times are higher than expected. This is because equation 3.2 assumes that the block scheduler does its best to schedule the blocks whenever sufficient resources are available, yet the real hardware might sacrifice the optimality for other metrics. In addition, Figure 4.5b shows the measurements that have smaller latency than the prediction. It is because the $fu(c)$ values used for the prediction can also be measured inaccurately.

Chapter 5

Conclusion

Understanding the mechanism of instruction scheduling in GPU is essential in optimizing the application performance and improving the hardware architecture. This has been a difficult task since the manufacturers are reluctant to disclose the hardware details and simulators are too complex and time-consuming to track down the impact factors.

This study provided the analytical model of GPU instruction scheduling mechanism and microbenchmarks that can extract its properties from real devices. The analytical model of the GPU execution pipeline is proposed as simple mathematical formulas. Based on the model, a set of microbenchmarks is designed and implemented to extract the architectural properties from real devices and evaluate the accuracy of the model.

As a result, the implementation GPUDiag effectively revealed the characteristics of instruction scheduling components. Measurements from both the simulators and real GPUs matched with the proposed model equations, and the hardware properties coincided with the known values. The end-to-end performance comparison between the prediction model and measurements showed a high average correlation of $r = 0.99$. Moreover, GPUDiag found the discrepancy of warp and block scheduling policy between gem5 and AMD RDNA architecture, suggesting the objective that can improve the simulator accuracy.

Bibliography

[1] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE international symposium on performance analysis of systems and software*, pp. 163–174, IEEE, 2009.

[2] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 473–486, IEEE, 2020.

[3] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, *et al.*, "Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 608–619, IEEE, 2018.

[4] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[5] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2016.

[6] R. H. Saavedra and A. J. Smith, "Measuring cache and tlb performance and their effect on benchmark runtimes," *IEEE Transactions on Computers*, vol. 44, no. 10, pp. 1223–1235, 1995.

[7] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[8] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing t4 gpu via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[9] AMD, "Rdna 1.0 instruction set architecture reference guide." https://developer.amd.com/wp-content/resources/RDNA_Shader_ISA.pdf, 2020.

[10] Nvidia, "Cuda toolkit documentation v11.5.0." https://docs.nvidia.com/cuda/index.html, 2021.

[11] HSAFoundation, "Hsa programmer's reference manual v1.2." http://hsa.glossner.org/wp-content/uploads/2021/02/HSA-PRM-1.2.pdf, 2018.

[12] AMD, "Amd hip programming guide." https://github.com/RadeonOpenCompute/ROCm/blob/master/AMD_HIP_Programming_Guide.pdf, 2021.

[13] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[14] E. Konstantinidis and Y. Cotronis, "A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling," *Journal of Parallel and Distributed Computing*, vol. 107, pp. 37–56, 2017.

[15] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. Narayanan, and K. Srinathan, "A performance prediction model for the cuda gpgpu platform," in *2009 International Conference on High Performance Computing (HiPC)*, pp. 463–472, IEEE, 2009.

[16] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A holistic approach to resource virtualization in gpus," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–14, IEEE, 2016.

[17] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *2011 IEEE 17th international symposium on high performance computer architecture*, pp. 382–393, IEEE, 2011.

[18] G. Alavani, K. Varma, and S. Sarkar, "Predicting execution time of cuda kernel using static analysis," in *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pp. 948–955, IEEE, 2018.

[19] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 235–246, IEEE, 2010.

[20] R. Taylor and X. Li, "A micro-benchmark suite for amd gpus," in *2010 39th International Conference on Parallel Processing Workshops*, pp. 387–396, IEEE, 2010.

[21] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "Gpu scheduling on the nvidia tx2: Hidden details revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, pp. 104–115, IEEE, 2017.

[22] J.-E. Jo, G.-H. Lee, H. Jang, J. Lee, M. Ajdari, and J. Kim, "Diagsim: Systematically diagnosing simulators for healthy simulations," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 1, pp. 1–27, 2018.

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