μC-States: Fine-grained GPU Datapath Power Management

Onur Kayıran[1] Adwait Jog[2] Ashutosh Pattnaik[3] Rachata Ausavarungnirun[4] Xulong Tang[3]
Mahmut T. Kandemir[3] Gabriel H. Loh[1] Onur Mutlu[5,4] Chita R. Das[3]

[1]Advanced Micro Devices, Inc. [2]College of William and Mary
[3]Pennsylvania State University [4]Carnegie Mellon University [5]ETH Zürich

ABSTRACT

To improve the performance of Graphics Processing Units (GPUs) beyond simply increasing core count, architects are recently adopting a *scale-up* approach: the peak throughput and individual capabilities of the GPU cores are increasing rapidly. This *big-core* trend in GPUs leads to various challenges, including higher static power consumption and lower and imbalanced utilization of the datapath components of a *big core*. As we show in this paper, two key problems ensue: (1) the lower and imbalanced datapath utilization can waste power as an application does not always utilize all portions of the big core datapath, and (2) the use of big cores can lead to application performance degradation in some cases due to the higher memory system contention caused by the more memory requests generated by each big core.

This paper introduces a new analysis of datapath component utilization in big-core GPUs based on queuing theory principles. Building on this analysis, we introduce a fine-grained dynamic power- and clock-gating mechanism for the entire datapath, called μC-States, which aims to minimize power consumption by turning off or tuning-down datapath components that are not bottlenecks for the performance of the running application. Our experimental evaluation demonstrates that μC-States significantly reduces both static and dynamic power consumption in a big-core GPU, while also significantly improving the performance of applications affected by high memory system contention. We also show that our analysis of datapath component utilization can guide scheduling and design decisions in a GPU architecture that contains heterogeneous cores.

1. INTRODUCTION

Graphics Processing Units (GPUs) have been used extensively to efficiently execute modern data-parallel applications. To improve GPU performance, architects have traditionally increased the number of GPU cores while keeping each core simple. More recently, in addition to increasing the processing core count, architects are adopting a *scale-up* approach to GPUs: the peak throughput and individual capabilities of each core are increasing rapidly (i.e., each core is getting bigger). For example, the AMD GCN [5] and TeraScale [32] architectures, and the NVIDIA Kepler architecture [69] have enhanced their core micro-architectures with more fetch/decode units, wavefront/warp schedulers, and functional units, compared to their predecessors. This *big core trend* in GPU design increases each GPU core's peak throughput and capabilities in successive generations, and can potentially enable significant performance boosts for many classes of applications.

Unfortunately, the *big core design approach* also leads to challenges in power consumption and performance. First, a big core dissipates more static power than a small one due to the larger and wider components in the datapath. Even if application performance increases with big cores, the additional performance benefit might not compensate for the increase in power consumption, which can reduce the overall energy efficiency of the system. Second, some applications do *not* gain performance from the enhanced features of big cores, which leads to extra power consumption without any benefit. Third, some applications whose performance is limited by the memory system[1] can lose performance with big cores, due to the higher memory contention created by big cores (as we demonstrate in this paper), which results in both lower performance and higher power consumption.

To illustrate the effect of the big core designs on performance, Figure 1 shows the performance (in instructions per cycle) of six representative applications running on a GPU with 16 big cores, normalized to each application's performance on a GPU with 16 small cores.[2] The system with big cores dissipates 44% more static power than the system with small cores. We observe that using the big-core system (1) significantly improves the performance of two applications (`SLA`, `MM`), (2) does not affect the performance of two others (`SCAN` and `SSSP`), and (3) degrades the performance of yet two others (`BLK`, `SCP`). The applications that lose performance with big cores stress the memory system significantly, and also generate a high number of memory requests per wavefront. In our big-core system, these characteristics lead to (1) more requests to be generated per core, (2) more requests from different wavefronts to contend with each other for memory service, leading to more wavefront stalls (because each wavefront can continue execution only when *all* of its memory requests are serviced and memory contention increases the likelihood of at least one of each wavefront's requests taking longer), and (3) more cache line reservation (i.e., allocation) failures due to small L1 caches in GPUs and longer queuing latencies at the caches due to cache and memory contention. Overall, these experiments with representative applications show that a GPU architecture with big cores is *not* always the best design choice.

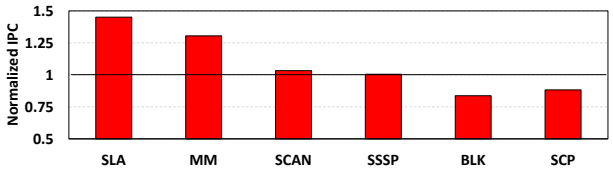


Figure 1: Performance of a GPU consisting of 16 big cores, normalized to a GPU with 16 small cores.

[1]We use the term *memory system* for the combination of caches, interconnect, and DRAM.

[2]Section 2 provides details of our small-core and big-core systems.

PACT '16, September 11–15, 2016, Haifa, Israel.

DOI: http://dx.doi.org/10.1145/2967938.2967941

As we have shown, both the small-core and the big-core GPU designs have pros and cons. **Our goal** is to design a GPU architecture that enables the big-core GPU designs to be practical and efficient in terms of both performance and power consumption. We achieve this by keeping the power consumption under control while reaping the performance benefits of big cores whenever possible, and preventing the big-core design from degrading application performance when memory becomes the performance bottleneck.

To this end, we first conduct a new, queuing theory-based analysis of the pipeline of a big-core GPU. From our detailed study, we observe that (1) *most* of the pipeline components are heavily under-utilized, (2) the utilization levels of different pipeline components vary significantly *across different applications*, and *within the same application*, and (3) the utilization level of a particular component varies significantly *across different phases of the execution*. Based on this component-level analysis, we identify the pipeline components that are critical for performance for different applications. We introduce a dynamic power- and clock-gating mechanism for the entire datapath, called μC-States, which aims to minimize power consumption by turning off or tuning-down datapath components that are not bottlenecks for the performance of the running application. We name our mechanism building on *C-States* [34], which is a *core-level* power-management technique employed in current general-purpose microprocessors under the direction of the system software. In contrast to this CPU mechanism, our proposal, μC-States, is (1) fine-grained, because it manages power consumption at the *datapath component level*, and (2) invisible to software.

The basic idea of μC-States is simple: power-gate or clock-gate datapath components (1) that are underutilized or (2) that, if operated at their full bandwidth, would lead to performance loss by causing higher memory contention (as described above). μC-States employs an algorithm that starts power/clock-gating from the pipeline back-end (i.e., execution units) and gradually ripples the power/clock-gating to the front-end (e.g, fetch/decode stages) in three different phases, in order to ensure minimal performance loss. μC-States employs clock-gating for certain pipeline components that would lose data or execution state when power-gated (see Section 2.3 and Section 4), which ensures that no state that affects correctness is lost.

Our comprehensive evaluations of μC-States show that it reduces average static chip power by 16% and dynamic chip power by 7%, compared to a baseline big-core design across 34 GPU applications. μC-States also improves the performance of applications that would otherwise lose performance with big cores by 9%, without significantly degrading the performance of any other applications. Our results demonstrate that μC-States is not specific to any wavefront scheduler and can work effectively with different schedulers. We also show that its benefits are not sensitive to power-gating related timing and energy overheads.

This paper also illustrates how our analysis of datapath component utilization of applications can be used to exploit (and perhaps design) a *heterogeneous GPU architecture* that contains both small and big GPU cores. We provide a case study that, based on our application bottleneck analysis, categorizes applications according to their affinity to small or big cores and maps each application appropriately to the core type it executes most efficiently on, in a multi-programmed environment where applications execute concurrently on the GPU system. Our study shows that such a heterogeneous GPU architecture reduces power and area cost over a big-core design, without hurting performance, and improves performance over a small-core design, due to the effective mapping of the core type to application needs.

To our knowledge, this is the first work that provides (1) a detailed and rigorous characterization of application sensitivity to datapath components in a GPU that contains big cores, and (2) a comprehensive fine-grained power- and clock-gating mechanism across the entire GPU datapath that builds upon observations from this characterization. We make the following **contributions**:

- We conduct a new, queuing-theory-based analysis of the GPU pipeline datapath components of a big-core GPU architecture across 34 applications. This study is the first to demonstrate (1) the utilization levels of all major GPU datapath components and (2) varying utilization of the components across and within applications. We believe this study and its methodology can help architects to explore the big-core and heterogeneous-core design spaces for GPUs.
- We propose μC-States, a comprehensive fine-grained dynamic power- and clock-gating technique for GPU datapath components in a big-core GPU design. We show μC-States works effectively for 14 different datapath components across 34 workloads, reducing both static and dynamic power consumption and improving performance.
- We demonstrate the advantages of a heterogeneous GPU design consisting of both big and small GPU cores on the same chip, for efficient concurrent execution of multiple applications. Our analysis shows that application-to-core mapping guided by our bottleneck datapath component analysis provides significant power and chip area savings over a big-core design, without hurting system performance.

2. BACKGROUND

2.1 GPU Architecture and Core Pipeline

A GPU compute pipeline consists of multiple components: fetch/decode units (IFID), wavefront schedulers (SCH), pipeline registers, operand collectors (OC), streaming processors (SP), special function units (SFU), and load store (LDST) units. The SP unit is composed of an integer and a floating point unit. Our small core configuration is similar to an Nvidia GTX480 GPU [9], which is based on the Fermi architecture [68]. Our big core configuration, shown in Figure 2, has twice the number of SP, SFU, LDST, and SCH units as our small core-based system. This configuration is similar to an NVIDIA GTX 660, which is based on the Kepler architecture [69], with slightly higher DRAM bandwidth. In our big core, there are 4 wavefront schedulers, and 2 SP, SFU, and LDST groups, where each group consists of 32 SP, 4 SFU, and 16 LDST units, respectively. The big core also has twice the number of IFID and OC units, and the pipeline register size as the small core. Table 3 shows the details of our big core configuration. We show in Section 6.4 that our proposal works effectively with a larger register file and a larger number of Cooperative Thread Arrays (CTAs) per-core.

Figure 3 shows the high-level diagram of a GPU core pipeline. The IFID block fetches/decodes instructions, and handles all types of instructions, including integer and floating (SP), special function (SFU), and memory (LDST) instructions. SCH schedules the instructions from the IFID stage and sends them to the appropriate pipeline. Depending on its type, an instruction is inserted into its corresponding pipeline register (IDOC_{SP}, IDOC_{SFU}, or $\text{IDOC}_{\text{LDST}}$). Each pipeline register is connected to a specialized operand collector (OC_{SP}, OC_{SFU}, or OC_{LDST}) that keeps track of the register accesses by the instructions. After the operand

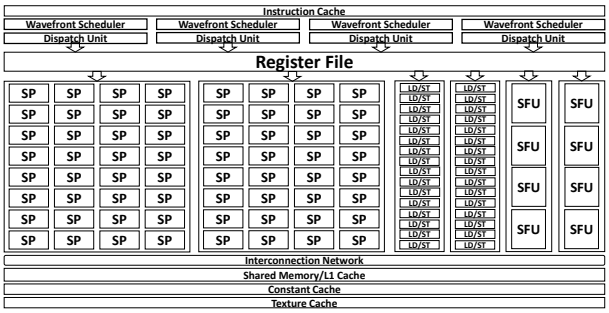


Figure 2: GPU core microarchitecture.

collectors, the control and data signals of the instruction are inserted into the corresponding pipeline register ($OCEX_{SP}$, $OCEX_{SFU}$, or $OCEX_{LDST}$) before they get sent into the execution unit (EX_{SP}, EX_{SFU}, or EX_{LDST}).

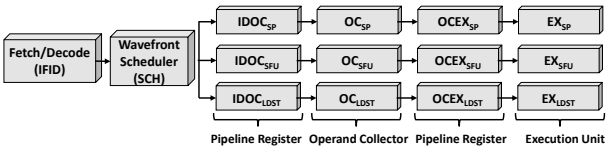


Figure 3: A high-level view of the GPU core pipeline.

2.2 Methods for Analyzing Core Bottlenecks

To identify performance bottlenecks and to gauge opportunities for turning off individual GPU datapath components, we borrow ideas from queuing theory. We make use of two key concepts. The *Utilization Law* states that the component with the highest utilization is the performance bottleneck [36]. To determine whether the performance bottleneck for memory instructions is the memory pipeline (LDST unit) or the rest of the memory system, we estimate memory latencies using *Little's Law* [56].

First, we investigate the utilization of each pipeline component. The utilization of the IFID block is calculated as the ratio of the cycles the IFID block is busy over total cycles, averaged across the width of the block. For example, in a single cycle, if IFID is 2-wide (i.e., IFID can fetch and decode two instructions per cycle), and fetches and decodes one instruction, its utilization is 0.5. The SCH utilization is given as the ratio of the cycles SCH is busy over total cycles, averaged across the SCH width. Similarly, we calculate the utilization of each pipeline register ($IDOC_{SP}$, $IDOC_{SFU}$, $IDOC_{LDST}$, $OCEX_{SP}$, $OCEX_{SFU}$, $OCEX_{LDST}$), operand collector (OC_{SP}, OC_{SFU}, OC_{LDST}), and execution unit, as the ratio of busy time of each over total cycles.

Utilization of a component is defined as the product of its throughput (X) and its average service time (S); $U = XS$. Throughput is defined as the number of completions per unit time. Service time is defined as the time for a completion. Based on the utilization levels of the pipeline components in Figure 3, we can detect the bottleneck (i.e., the component with the highest utilization). This analysis can be used for power/clock-gating of relatively less utilized (i.e., non-bottleneck) components without degrading performance.

We use the above utilization-based analysis for SP and SFU pipelines. The execution units for these two stages have deterministic service times, and from our simulations we observe little queueing delay for these units. By measuring the throughput of the SP and SFU unit each and multiplying it with the known service time of each, the utilization value calculated with Utilization Law closely matches the actual utilization value we measure for each of these two units.

However, this calculation is not so accurate in the memory pipeline, where the service time is variable and depends on memory system performance. We observe that the queuing delay in the memory pipeline is neither low nor predictable. The inability of the service time metric to capture queuing delay leads to an incomplete picture of the total delays in the unit. Therefore, for the memory pipeline, we use the *Response Time* metric, which is defined as the time between when a memory instruction is sent from the pipeline register $OCEX_{LDST}$ to the EX_{LDST} unit and when the memory instruction is committed. Response time captures both the service time as well as the queuing delay of the unit. Due to the difficulty in measuring response time in hardware, we estimate its value using *Little's Law*. *Little's Law* states that the average number of jobs (J) in the queue of a system is equal to the average arrival rate of the jobs to the system (λ) multiplied by the average response time for a job in the system (W): $J = \lambda W$.[3]

Utilization Law and Little's Law allow us to determine the performance bottlenecks of GPUs using a theoretical framework, which we then use to develop our power/clock-gating mechanism. This theory-based background provides robustness to our proposal, making it less sensitive to effects of the wavefront scheduler (Section 6.4) and dynamic changes in application behavior.

2.3 Power- and Clock-Gating

We power/clock-gate components at the granularity of a group (as defined in Section 2.1). We do not gate all of the groups of the same type. For example, when power-gating SP, we power-gate only *one* of the two groups.[4] All of the integer and floating point units in the power-gated SP group are turned off. Similarly, we clock-gate only half of the LDST units. We call a component as ***tuned-down*** if one of the groups (i.e., half of the units) in that component is power/clock-gated. Tuning-down a component halves its peak throughput.

While power-gating (PG) reduces static power, clock-gating (CG) reduces dynamic power. The potential power savings with PG is more than that with CG, and this difference will be even more significant as static power becomes more important with technology scaling [78]. However, PG is more difficult to employ because it has higher timing overheads, requires more idle time to be employed, and needs to be employed for long enough (i.e., at least for *break-even time*) to compensate for its energy overhead. Moreover, PG results in loss of data for the gated component. Thus, we use PG to tune-down components that do *not* store data (instruction decoder, wavefront scheduler, integer and floating-point units, and special functional units), and CG to tune-down components that store data (instruction buffer, pipeline registers, operand collectors, and load/store queue). For example, tuning-down for IFID means clock-gating half of the IFID. A tuned-down IFID can fetch and decode at most 2 instructions per cycle, which is half the peak number of instructions per cycle in our baseline. Tuning-down for EX_{SP} means power-gating half of the SP. While the baseline system can execute 64 integer instructions per cycle, this number is 32 for a tuned-down EX_{SP}.

Overheads. A component should be power-gated for at least a time period, called *break-even time*, in order to compensate for the incurred energy overhead of entering and

[3] We use the memory response time calculated based on Little's Law to determine whether or not the memory system is congested.

[4] We do not explore having more groups, because having more than two power domains per core might not be practical to implement.

exiting the PG mode . The time that is needed to turn the component back on is known as the *wake-up delay*. We assume a break-even time of 19 cycles and wake-up delay of 9 cycles [33]. Section 6.4 provides a sensitivity study of these parameters. CG is employed by disabling the clock signal to a pipeline component. It effectively reduces the component's activity factor, has no break-even overhead, and the clock-gated component can wake up in the next clock cycle by enabling the clock signal.

2.4 Existing Component Gating Mechanisms

Traditional CPUs. Prior works that employ pipeline-component-level gating mechanisms focused on CG [8, 55, 58]. However, such mechanisms are employed at a very fine granularity, and thus are not suitable for PG due to the wake-up and break-even overheads. As static power becomes a major contributor to the total chip power with CMOS technology scaling [78], PG will be increasingly important for power management. Furthermore, some of these works [8,55] do not consider gating the issue stage. However, as we will show in Section 3.2, reducing the peak throughput of the issue stage via a gating mechanism does not only reduce power consumption, but can also improve application performance in a GPU.

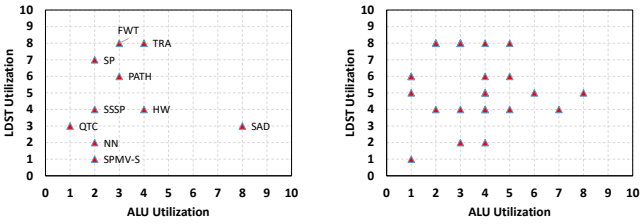
GPUs. Warped Gates [2] is the only work that proposes a pipeline-level PG mechanism for GPUs. However, it leads to significant performance losses in some applications because it can power-gate performance-critical components. It is also employed at a fine granularity, and is dependent on the wavefront scheduling algorithm to create idle cycles for the components that will be power-gated. We will show (in Section 3.2) that a rigorous analysis of component utilization in the whole GPU pipeline can guide us in determining the performance-critical components. This enables us to tune-down components without significant performance loss even if they have non-idle cycles. GpuWattch [54] employs CG for SIMD lanes in the presence of branch divergence, and does not consider any of the possible component-level power-saving opportunities that occur due to underutilization (as we analyze in Section 3.2).

3. MOTIVATION AND ANALYSIS

Through detailed workload analysis, we first show that GPU pipeline resources are heavily under-utilized. We then conduct a utilization analysis of the entire GPU pipeline to identify performance-limiting components, which motivates our bottleneck-aware power/clock-gating mechanism.

3.1 The Problem: Under-utilized GPU Core

To demonstrate the under-utilization of the GPU pipeline, we profiled our 34 GPU applications (Table 1) on an NVIDIA K20 card using the CUDA profiler. Because K20 has higher peak throughput and DRAM bandwidth than our baseline (shown in Table 3), we also profiled 27 different kernels from CUDA SDK 6.5 on an NVIDIA GTX 660 card, which is similar to our baseline. Figure 4 shows the ALU and LDST utilization of each application/kernel on a scale from 0 to 10, where 10 denotes 100% utilization of the component, on the two systems. We show only 10 representative applications with their names in Figure 4a for comparison with our simulation results in Table 1. Overall, the real system results in Figure 4 demonstrate that ALUs are heavily under-utilized in most of the applications, and several applications make heavy use of the LDST pipelines. Thus, the pipeline under-utilization problem we address in this paper is a problem clearly visible in existing GPU hardware executing real GPU workloads (and it is not an artifact of simulation or that of an over-provisioned baseline).



(a) NVIDIA K20 (b) NVIDIA GTX 660

Figure 4: ALU and LDST pipeline utilization in real GPUs.

The GPU profiler does not provide utilization statistics for pipeline components except ALU and LDST. To get a better understanding of the under-utilization problem, we tabulate the utilization of 14 different components in Table 1 across 34 benchmarks, obtained using GPGPU-Sim [9]. The y-axis lists the benchmarks, and the x-axis shows the utilization (between 0% and 100%) of each component. We break down IFID and SCH utilization into the type of instruction (SP, SFU or MEM) that is decoded and issued.

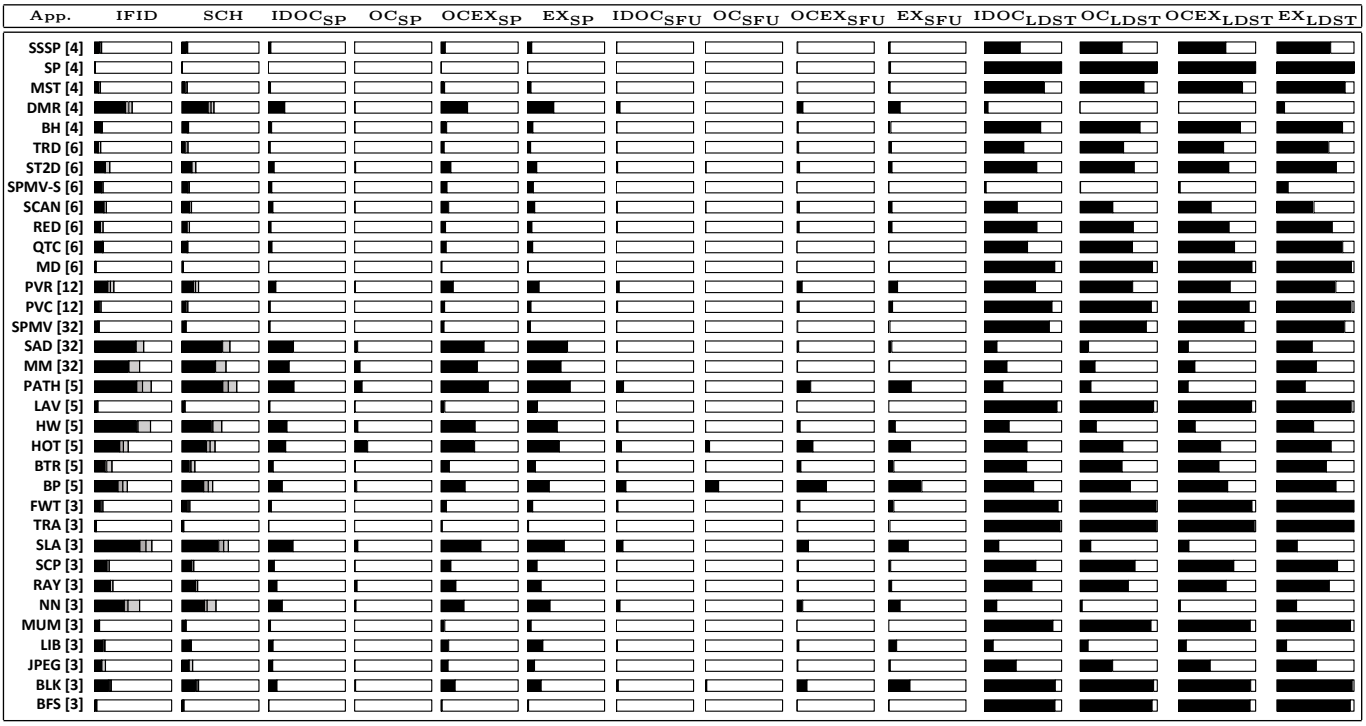
Five key observations are in order. First, we observe that the real experiment and the simulation results follow the same trends: there is significant ALU under-utilization, and the LDST pipeline is stressed in several applications. We observe the same trends for other benchmarks that are not shown in Figure 4a. Second, each application stresses *different* components *differently*. For example, compute-intensive applications (e.g., `SAD`, `MM`, `HW`) utilize IFID, SCH, and EX_{SP} significantly, and do not utilize EX_{LDST} to the same extent. Conversely, memory-intensive applications (e.g., `QTC`, `BFS`, `BLK`) do not utilize IFID, SCH, and EX_{SP}, but heavily utilize EX_{LDST}. Third, in the applications that have high EX_{SP} or EX_{SFU} utilization, the OCEX register, SCH and IFID units also have similar utilization as the execute stage, but the IDOC register and the operand collectors usually have lower utilization. This leads us to conclude that the bottleneck for SP or SFU instructions in such applications is one of IFID, SCH, OCEX, or EX units. Fourth, in the applications that have high EX_{LDST} utilization, units other than $IDOC_{LDST}$, OC_{LDST}, $OCEX_{LDST}$, and EX_{LDST} have very low utilization. Memory instructions in such applications are bottlenecked by either the LDST unit, or the memory system. Due to the heavy utilization of these units, the SCH units stall for long periods either waiting for data to come back from memory, or not finding independent instructions to execute, leading to low IFID and SCH utilization. Fifth, we observe very low utilization in SFU-related components (0.8% in OC_{SFU}) as well as OC_{SP} (1.8%) units across all applications. We also note that, across the 34 applications, the average utilization of memory instruction-related components is around 50%-60%, and average IFID, SCH, and EX_{SP} utilization is 24%, 21%, and 17%, respectively.

3.2 Application Performance Sensitivity to Pipeline Resources

3.2.1 Representative Applications

We now evaluate the impact of tuning-down GPU core components on application performance. The intuition is that tuning-down components that are not highly utilized, or less utilized compared to other components would not degrade performance, but would lead to power savings. We evaluate 8 different core configurations listed in Table 2, where each entry is a different type of tuned-down core. For

Table 1: Component-wise utilization breakdown for 34 CUDA applications. A more shaded bar indicates higher utilization. IFID and SCH utilization is further broken down into SP, SFU, and LDST instruction utilizations, from left to right with different shades of gray.



example, in C_{HALF}, all 14 components are tuned down (i.e., operate at half their peak baseline throughput). In C_{IFID}, all components except IFID are tuned-down. Similarly, in C_{SP}, all components except EX_{SP} are tuned-down. In C_{COMP}, all components except components related to the compute path (including IFID and SCH) are tuned-down. In C_{MEM}, all components except components related to the memory path (including IFID and SCH) are tuned-down. Note that tuning-down a component means power-gating/clock-gating only half of that component.

Table 2: Different Tuned Down Configurations

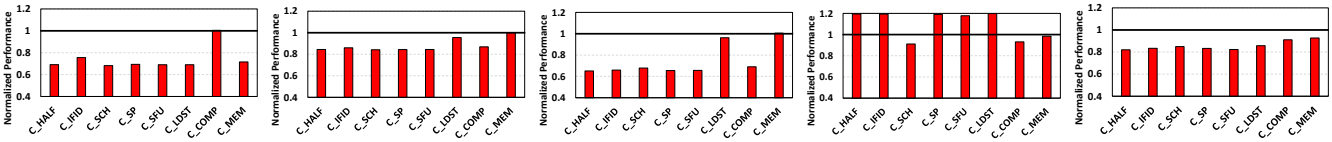
Config.	Tuned-down components	Config.	Tuned-down components
C_{HALF}	ALL	C_{COMP}	ALL except IFID, SCH, and Compute Related Components
C_{IFID}	ALL except IFID		
C_{SCH}	ALL except SCH		
C_{SP}	ALL except EX_{SP}	C_{MEM}	ALL except IFID, SCH, and Memory Related Components
C_{SFU}	ALL except EX_{SFU}		
C_{LDST}	ALL except EX_{LDST}		

Figure 5 shows the effect of these 8 tuned-down configurations on the performance of 5 representative applications. `HW`, a compute-intensive application, demonstrates a case where IFID, SCH, and EX_{SP} are all critical for performance. `SPMV-S`, `QTC`, and `BLK` show three different cases in which we analyze memory related performance bottlenecks. Finally, we study `NN`, which is sensitive to most of the components in the pipeline.

We observe in `HW` that C_{COMP} has almost no performance impact. This is expected, as all components related to memory ($IDOC_{LDST}$, OC_{LDST}, $OCEX_{LDST}$, EX_{LDST}) have low utilization (Table 1) and tuning them down would not lead to performance loss. We also observe that C_{HALF} degrades performance to 69% of that of the baseline. C_{IFID} (76%) performs significantly better than C_{SCH} (68%), because the utilization of IFID is higher than that of SCH and the other components that are used by SP and SFU instructions (Table 1). Thus, for SP and SFU instructions, IFID is the bottleneck component. Therefore, tuning down IFID causes more performance loss than tuning-down the other components. In C_{IFID}, the bottleneck moves to the execution units, and increasing these bottleneck resources in C_{COMP} leads to higher performance compared to C_{IFID}. `HW` *exemplifies a compute-intensive application. Because many compute components such as IFID, SCH, and the execution units have similar utilization values, all of these resources are critical for performance in such an application.*

The next three applications, `SPMV-S`, `QTC`, and `BLK` are sensitive to memory pipeline resources. As shown in Table 1, all of the components in `SPMV-S` are under-utilized. This means that the bottleneck is not the memory system. However, EX_{LDST} is the unit with the highest utilization, thus EX_{LDST} is the bottleneck component for this application. Therefore, C_{LDST} performs close to the baseline. Also, similar to `HW`, since IFID utilization is slightly higher than SCH utilization, C_{IFID} performs slightly better than C_{SCH}. Thus, C_{MEM}, which combines the benefits of C_{IFID} and C_{LDST}, performs as well as the baseline. `SPMV-S` *exemplifies a case where performance is highly dependent on* EX_{LDST} *even when the LDST pipeline utilization is low.*

Both `QTC` and `BLK` demonstrate high memory pipeline utilization. However, they show contrasting sensitivities to different configurations. `QTC` is similar to `SPMV-S`, because the utilization of EX_{LDST} is significantly higher than that of all the other components (making EX_{LDST} the bottleneck component), and contributes significantly to performance. However, in `BLK`, the bottleneck is the memory system,[1] not EX_{LDST}, because we observe very high DRAM bandwidth utilization and very long memory latencies. In this application, a higher issue width degrades performance significantly because it leads to more contention in the already-oversubscribed memory system. Thus, configurations that tune-down components in a way that reduces memory contention (C_{HALF}, C_{IFID}, C_{SP}, C_{SFU}, C_{LDST}) significantly improve performance over the baseline.



(a) HW (b) SPMV-S (c) QTC (d) BLK (e) NN

Figure 5: Effect of tuned-down configurations on performance. All results are normalized to the baseline (Section 2).

In applications bottlenecked by memory bandwidth (e.g., BLK), we observe high memory access latencies. This is in contrast to QTC, where latency is relatively low, and the performance bottleneck is EX_{LDST}. In applications bottlenecked by the memory system,[1] we see a high number of stalls *after* the memory instructions leave EX_{LDST} (this can be due to contention anywhere in the memory system). From a theoretical point of view, the average memory latency is similar to the response time of EX_{LDST}, described in Section 2.2. We observe that both the average memory latency and the average response time of EX_{LDST} in QTC is approximately half of those in BLK. In summary, QTC *demonstrates a case where the memory pipeline (EX_{LDST}) is the bottleneck, and more memory pipeline resources improve memory-level parallelism, leading to better performance. On the other hand,* BLK *(similar to* SCP, TRA, FWT, SPMV, RAY, *and* RED*) demonstrates a case where the memory system (i.e., caches, network, DRAM) is the bottleneck, and increasing the instruction throughput aggravates the problem of memory contention, leading to lower performance.*

In NN, almost all components affect performance. Thus, tuning-down most of the components reduces performance, and the baseline outperforms all other configurations.

3.2.2 Memory-Bottlenecked Applications

In applications such as BLK (Figure 5d), SCP, TRA, FWT, SPMV, RAY, and RED, the performance bottleneck is the contended memory system. These applications have very high memory bandwidth utilization, and injecting more requests to the system by doubling the instruction throughput (e.g., C_{SCH} compared to C_{HALF}) leads to significant performance degradation. To understand the reason for this, we investigate the stalls incurred at L1 caches. The stalls happen due to three main reasons. First, due to the high number of memory requests, Miss Status Handling Registers are filled, and further incoming requests stall. Second, miss queue buffers are filled, and this prevents new packets from being injected into the network. Third, if the L1 data cache uses an Allocate-on-Miss policy, misses cannot reserve cache lines before they can be serviced. If the policy is Allocate-on-Fill, our simulations show that the problem manifests itself at L2 and DRAM, and the memory system bottleneck still causes performance issues. Figure 6 shows the performance and the *L1 stall cycles per L1 miss* of C_{SCH}, normalized to those of C_{HALF}, for representative applications. We observe that the increase in L1 stall cycles per miss correlates well with the performance loss due to doubling the instruction throughput in C_{SCH} compared to C_{HALF}. The applications with higher memory bandwidth consumption suffer higher performance losses compared to the others. We find these main trends to be similar in other applications as well.

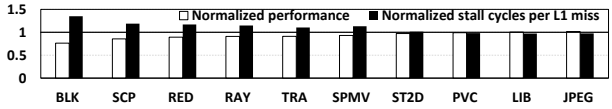
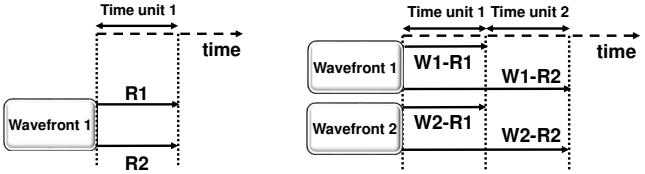


Figure 6: Performance and the number of L1 stall cycles per L1 miss with C_{SCH}, normalized to C_{HALF}.

In order to provide insight into the increase in stalls due to doubled issue width, Figure 7 provides an illustration by showing how the execution latency of a memory instruction in a wavefront (Figure 7a) can more than double if another wavefront's (Wavefront 2) memory instruction contends with this wavefront's (Wavefront 1) instruction when the issue width is doubled (Figure 7b).

In Figure 7a, the scheduler issues an instruction from only a *single* wavefront due to the small issue width. This wavefront issues two memory requests (R1 and R2), and it stalls until *all* of these requests are serviced and their data returns to the core. Assume that the memory bandwidth is close to saturation and allows only two requests to be serviced per unit time and that the wavefront's two requests return to the core after a single time unit. Once they arrive, the wavefront can continue its execution with the next instruction. Thus, *the latency of the wavefront's instruction is only a single time unit* since both of the instruction's memory requests arrive after a single time unit. Assuming all instructions in the wavefront are similar, performance is sustained at one instruction per time unit.



(a) Single issue width (b) Doubled issue width

Figure 7: Effect of increasing issue width on performance.

In Figure 7b, the issue width is doubled, and the scheduler can issue one instruction each from *two* wavefronts at the same time. Similar to (a), assume that due to limited memory bandwidth, only two requests can be serviced per time unit. Also assume that, due to contention in the memory system, the memory system services R1 from each wavefront concurrently and these two (W1-R1 and W2-R1) return in one time unit. Thus, neither of the instructions in different wavefronts is complete at the end of the first time unit because R2 is not even sent to the memory system. Once R1 from each wavefront is complete, R2 from each wavefront is sent to the memory system. Only at the end of the second time unit, R2 from each wavefront is also complete, and hence each wavefront can now move to issue its next instruction. Thus, *the latency of each instruction is two time units* since both of the instructions' memory requests arrive after two time units. Assuming all instructions in a wavefront are similar, performance is sustained at one instruction per *two* time units. Thus, the performance of each wavefront halves compared to the case where issue width is half (Figure 7a). The actual performance of the system in (b) is actually even worse than depicted: due to the larger number of outstanding memory requests in the system, contention and queueing delays and buffering conflicts in various parts of the memory system (caches, interconnect, and DRAM banks, buses, row buffers) increase [7, 17–19, 21, 42, 47–49, 51, 62, 64–66, 82, 84], and, hence, the latency of servicing each request becomes larger than a single time unit. Thus, *either wavefront in (b) can complete one instruction long after two time units,*

leading to overall system performance degradation compared to when the issue width is half.[5]

We observe this scenario especially in applications with high memory divergence [7, 14, 59] and those where some requests are much more critical than others [7, 42] for application performance. Such applications are more prone to longer latencies for an instruction because one or few memory requests can easily dominate the performance of an instruction if they get delayed due to increased memory contention. This problem becomes more severe in cases where a wavefront instruction issues a high number of requests. For example, Figure 6 shows that PVC is not very sensitive to issue width although it has high bandwidth consumption. This is because, a wavefront in PVC generates 2.36 memory requests, on average, whereas this number is much higher in more sensitive applications (e.g., 4.60 in BLK).

Several works [7,14,59] identified that, if multiple requests from the same wavefront are serviced at different levels in the memory hierarchy (e.g., one request is an L1 hit, while the other is served by the DRAM), long latency operations such as DRAM accesses become critical for performance, and cause significant wavefront stalls. The problem we observe is evident *even when all the requests are served at the same level in the memory hierarchy*. The increase in outstanding requests puts pressure on the memory system. Specifically, as L1 caches are small in GPUs, applications with high miss rates cause a majority of the cache lines to be reserved (waiting for misses to return), preventing other misses to be served. This causes cache stalls to increase. A miss that cannot be served increases the probability of its corresponding wavefront to stall longer, and becomes critical for performance. In a big core configuration, because there are more requests in the system, the number of critical requests increases, causing performance degradation. Overall, we find that *memory-divergent applications that are bottlenecked by memory bandwidth benefit from reduced issue width.*[6]

Summary. We conclude that applications utilize various datapath components differently and are bottlenecked by different parts of the datapath or the memory system. Hence, a PG/CG mechanism that is aware of the performance bottlenecks of each application is promising to achieve both power reduction and performance improvement in a GPU system with big cores.

4. μC-States: A DYNAMIC POWER- AND CLOCK-GATING MECHANISM

We describe μC-States, our approach for tuning-down components that are *not* critical for performance. In contrast to the core-level CPU power-management technique, C-States [34], μC-States works at the datapath component level and is invisible to software. The objective is to 1) reduce the static and dynamic power of the GPU pipeline by employing PG and CG in appropriate pipeline resources, and 2) maintain and, when possible, improve GPU performance by employing PG/CG in a bottleneck-aware fashion, based on our analysis in Section 3.2.

Power Benefits. We employ μC-States to reduce both static and dynamic power consumption. As opposed to the prior pipeline-level PG mechanism that creates idle times to power-gate components [2] via a smart wavefront scheduling mechanism, our mechanism relies on our analysis to determine the components that are *not critical* for application performance. Gating such components, even if they are not idle, does not have a significant effect on performance. Since we do not need to create idle times for gating, our mechanism (1) can be employed at a *coarser time granularity*, (2) has benefits that are unlikely to be dependent on the energy overheads of entering/exiting the power-gating state, (3) is independent of the underlying wavefront scheduler.

Performance Benefits. Our strategy aims to improve the performance of applications that are bottlenecked by the memory system. Section 3.2 already showed that such applications benefit from reduced instruction throughput. The performance benefits of our technique mainly come from clock-gating the wavefront scheduler and reducing the issue width. This is fundamentally different from prior works that focus on reducing GPU cache thrashing [7, 41, 43, 44, 67, 76] (see Section 6.2), and pipeline-level CG techniques for traditional CPUs [8,55] that do not focus on limiting issue width. Note that we power-gate or clock-gate only one group in a component. Because the peak throughput of the tuned-down component is halved, the progress of instructions inside the datapath is not entirely blocked.

Maintaining Execution State. μC-States employs power-gating for components that do *not* store data and clock-gating for components that do (see Section 2.3). When μC-States decides to power-gate a component, it stops pushing more work into the component. Once the component finishes all its outstanding work, the component is power-gated. Thus, no data or execution state is lost.

μC-States Gating Algorithm. In order to dynamically tune the pipeline components, each core requires a *Gating Control Unit*. As shown in Figure 8, the controller periodically decides whether or not a group of components should be tuned-down. Because tuning-down a component might impact the utilization of that component and other components, tuning *all* components at the same time is sub-optimal. μC-States gates components in three *phases*. As the performance of the system is determined by the throughput of the functional units and the LDST units, and because the throughput of these components do not *directly* affect the instruction count incoming to any other component, the algorithm starts with gating these components. In the first phase (after an interval of 2048 cycles), the controller tunes only the EX_{SP}, EX_{SFU} and EX_{LDST} units. In the second phase (2048 cycles after the decision for the first phase), the remaining components belonging to SP, SFU, and LDST pipelines (i.e., IDOC registers, OCEX registers, OC units, and register file banks) are tuned. In the third phase, the components that affect all types of instructions (i.e., IFID and SCH units) are tuned. Then, the controller moves back to the first phase. Each type of resource in our architecture comprises of two groups. We tune only one group during run-time (Section 2.3). Thus, at a given time, there is always at least one group active from each type of resource to ensure forward progress. Algorithm 1 provides the pseudo-code of μC-States.

The First Phase. The first phase of our algorithm tunes EX_{SP}, EX_{SFU}, and EX_{LDST}. It makes two main decisions. The first decision is related to EX_{SP} and EX_{SFU}. As dis-

[5] A better memory system design that tries to service each wavefront's requests in parallel (e.g., using *ranking* principles similar to those in [7,13,18,22,48,49,63,66,71,73,83,84,91]), can provide better performance than existing GPU memory systems, but they cannot eliminate the memory contention problem. Thus, we leave the exploration of such memory systems to future work.

[6] Note that the problem we have described is different from the one observed by prior work [43,67,76], where limiting the number of wavefronts/CTAs reduces cache line reuse distances and improves locality. In this paper, we reduce the *issue width*, which does *not* have a significant effect on cache line reuse. In fact, reducing the issue width can actually benefit applications that do *not* benefit from caching, whereas the aforementioned prior works obtain performance benefits mainly via improved cache performance.

Algorithm 1 μC-States

Phase 1
for i = SP, SFU **do**
  if $U[EX_i] < th_{\mathrm{U}}$ **then**
    if EX_i not tuned down **then**
      $tune - down[EX_i]$.
  else
    $tune - up[EX_i]$.
if $ResponseTime[EX_{LDST}] > th_{\mathrm{RT}}$ **then**
  if LDST not tuned down **then**
    $tune - down[EX_{LDST}]$.
else
  $tune - up[EX_{LDST}]$.

Phase 2
for i = OCEX, OC, IDOC **do**
  for j = SP, SFU, LDST **do**
    if $U[i_j] < U[EX_j]$ **then**
      if i_j not tuned down **then**
        $tune - down[i_j]$.
      else
        $tune - up[i_j]$.

*U[i] = Utilization of component i
*$tune - up[i]$ = Turn ON all groups of component i

Phase 3
if $(U[SCH] < (th_{\mathrm{U}} \times 2))$ &&
  $(U[SCH_i] < U[EX_i])$ $\forall$ i = SP, SFU, LDST **then**
  if SCH not tuned down **then**
    $tune - down[SCH]$.
  if $U[IFID] < U[SCH]$ **then**
    if IFID not tuned down **then**
      $tune - down[IFID]$.
  else
    $tune - up[IFID]$.
else
  $tune - up[SCH]$; $tune - up[IFID]$.

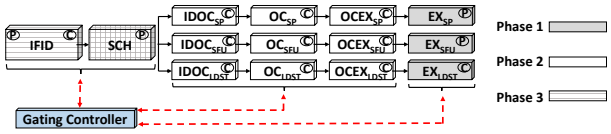


Figure 8: Phases of μC-States. Components marked with P and C are power-gated, and clock-gated respectively. In IFID, the instruction decoder is power-gated, and the instruction-buffer is clock-gated.

cussed in Section 3.1, the performance of the applications that have high EX_{SP} or EX_{SFU} utilization are sensitive to those resources. Thus, the controller tracks the utilization of EX_{SP} and EX_{SFU} during an interval. At the end of the interval, it tunes-down the components with low utilization levels. In order to track component utilization, the controller tracks the busy cycles of each component during the interval. EX_{SP} busy cycles in an interval is between 0 and the number of powered-on SPs, multiplied by the interval length. Dividing this number by the total number of SPs, and then by the interval length yields EX_{SP} utilization. If this value is lower than a threshold (th_{U}), the component is tuned-down. If it is above the threshold, all groups are turned on. Note that, if a component is tuned-down, its utilization will double assuming its throughput remains unchanged. Thus, a threshold greater than 0.5 is likely to cause performance loss.

The second decision is related to EX_{LDST}. Applications do not benefit from more LDST units if the performance bottleneck of the application is the memory system rather than the memory pipeline (Section 3.2). Applications that are bottlenecked by the memory system observe high memory access latency. Thus, tracking memory access latency can be one way to find which applications are bottlenecked by the memory system. However, since tracking the average memory access latency in hardware is costly due to the need of tagging each request with a time-stamp and measuring the latency of each returned request, we approximate it by calculating the response time of EX_{LDST} using Little's Law (see Section 2.2). The controller calculates the average response time of EX_{LDST}, and tunes-down the unit if the response time is higher than the Response Time threshold (th_{RT}). Response time is calculated by keeping track of the number of wavefronts that have outstanding memory accesses (J) and the number of wavefronts that have entered EX_{LDST} during the last interval (λ). Note that the first phase does not require EX_{LDST} tracking utilization, but requires calculating EX_{LDST} response time.

The Second Phase. This phase tunes IDOC and OCEX pipeline registers, OC units, and register file banks. It makes use of the fact that the bottleneck component is the component with the highest utilization. Based on this, the controller calculates the ratio of the utilization of each component to the utilization of its corresponding execute stage unit (e.g., $IDOC_{SP}$ is compared against EX_{SP}; or $OCEX_{LDST}$ is compared against EX_{LDST}). If this ratio is lower than 1 (i.e., the component is not the bottleneck), the component is tuned-down. The ratio must be less than 0.5 to ensure that the bottleneck does not move from the execute stage to the tuned-down component. However, we find that the pipeline registers, OC units, and the register file banks are rarely bottleneck components, therefore, we employ an aggressive approach for tuning-down these components. In case these components become the bottleneck after this phase, the decision is reverted back in the next execution of this phase. This phase requires the utilization values for all IDOC, OC, OCEX, and EX stage components.

The Third Phase. This phase tunes IFID and SCH. Tuning these components is different from tuning others, because IFID and SCH serve *all* types of instructions, whereas other components are dedicated for a specific type of instruction. In this phase, we break the utilization of SCH into three: SP, SFU, and MEM type instructions. The controller compares SCH utilization for each type of instruction to the utilization of the corresponding execute stage unit. If SCH utilization is higher for at least one of the instruction types, SCH is considered to be the bottleneck, and is not tuned-down. Also, if SCH utilization is greater than a threshold ($2\times$ th_{U}), it is not tuned-down.[7] Otherwise, SCH is tuned-down. If the overall IFID utilization is lower than SCH utilization, IFID is *not* the bottleneck component, and it is also tuned-down whenever SCH is tuned-down.

Component Interactions. Tuning-down a component affects the other components that share the same instruction path. For example, if EX_{SP} is tuned-down, its utilization increases, which creates opportunities for tuning-down other non-bottleneck components. Following this, if SCH is also tuned-down, its utilization increases. If this moves the bottleneck to SCH, SCH will not be tuned-down anymore, as the limited SCH throughput would reduce system throughput. Thus, μC-States tries to achieve a system with balanced component utilization levels across its iterations.

5. EXPERIMENTAL METHODOLOGY

Simulated System. We simulate the baseline architecture described in Table 3 using a modified version of GPGPU-Sim v3.2.2 [9]. The modifications allow GPGPU-Sim to run multiple applications concurrently and support heterogeneous core configurations.

Application Suites for GPUs. We evaluated 34 GPU applications from various suites such as the CUDA SDK [9], Rodinia [15], Parboil [81], Mars [30], SHOC [16], and LonestarGPU [11], listed in Table 1.

[7] We pick this threshold, because we observe that SCH is utilized mainly due to SP and SFU components, not LDST (see Table 1). The threshold for keeping all SCH groups on should be greater than th_{U}, because SCH serves both classes of instructions, each using th_{U} in their respective Phase-1 tuning decisions. Thus, we use $2\times$ th_{U} to tune SCH.

Table 3: Baseline GPGPU configuration.

Core Config.	16 Shader Cores, SIMT Width = 32 × 4
Resources/Core	Max. 1536 Threads, 32 threads/wavefront, 48 wavefronts, 36864 Registers
Caches/Core	16KB 4-way L1 Data Cache, 12KB 24-way Texture, 8KB 2-way Constant Cache, 2KB 4-way I-cache, 48KB Shared Memory, 128B Line Size
LLC Cache	768 KB/Memory Partition, 128B Line, 16-way
Wavefront Sch.	Greedy-then-oldest [76]
Features	Memory Coalescing, Inter-wavefront Merging
Interconnect	16×6 Crossbar, 32B Channel Width
Memory Model	6 Shared GDDR5 MCs (×2 sub-partitions), 924 MHz, FR-FCFS [75,92], 16 DRAM-banks/MC

Performance Metrics. We report Instructions Per Cycle (IPC). For heterogeneous GPU simulations, we report Weighted speedup (WS) [23, 24, 80], defined as $WS = \sum_{i=1}^{N} SD_i$, where SD_i is the slowdown of the i^{th} application given by $SD_i = \frac{IPC_i}{IPC_i^{alone}}$, where IPC_i^{alone} is IPC of the i^{th} application running alone on the entire GPU.

Power Metrics. We use GPUWattch [54] to obtain the dynamic power consumption. To estimate static power, we configure GPUWattch [54] to model our baseline, and use its area estimates to obtain the area of individual core components. We assume the static power of each core component is directly proportional to its area [79]. We conservatively assume that non-core components, such as the memory subsystem and DRAM, contribute to 40% of static power (using a smaller ratio would increase our power benefits). We assume no static power for a component group when it is power-gated. We report dynamic and static power separately from each other.

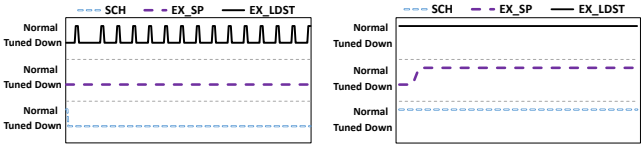
Mechanism Thresholds. Our scheme uses 2 thresholds: th_U and th_{RT}. We determine th_U to be 0.15, based on the data given in Table 1 and the sensitivity of each application to EX_{SP} and EX_{SFU}. Phase-3 of our scheme uses 2× th_U for tuning SCH. We observe that our scheme works efficiently when th_U is between 0.1 and 0.2. Phase-1 uses 100 cycles as th_{RT}, and our scheme is effective when this value is between 80 and 150. All thresholds are empirically determined.

Hardware Overhead. To implement μC-States, each core requires a storage overhead of 32B. For the computation of the response time and the utilization of the pipeline components, each core requires a 14-bit division unit, a 7-bit adder, and a 14-bit comparator.

6. EXPERIMENTAL RESULTS

6.1 Dynamic Adaptation

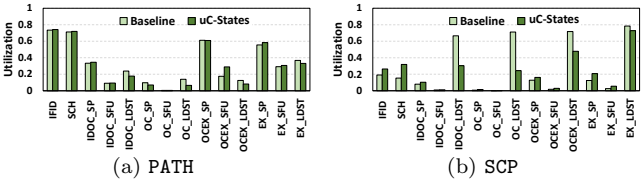
We analyze the dynamic behavior of μC-States. We show how it tunes down components during execution in two different applications with different dynamic behavior, `RED` and `PATH`, in Figures 9a and 9b, respectively.[8] Note that SCH and EX_{SP} are power-gated, whereas EX_{LDST} is clock-gated. `RED` is one of the applications that experiences significant tuning-down/up at run-time. Since this application has low EX_{SP} utilization (Table 1), μC-States tunes down the SP units after the first interval, and keeps it tuned down during the whole execution. Following this, the scheduler becomes under-utilized compared to the execution units, and is also tuned-down throughout the execution. Although SCH and EX_{SP} show a stable behavior over time, one group of EX_{LDST} units is frequently turned on and off. This happens due to the sudden changes in response time observed by EX_{LDST}. The most frequent re-tuning of EX_{LDST} is done at an interval of 6144 (2048 × 3) cycles. Overall, we observe that a group of EX_{LDST} units is tuned down 80% of the time. Our analysis shows that `RED` performs as good as the baseline with C_{HALF} (Table 2), and tuning down the components most of the time does not hurt performance. In contrast to `RED`, `PATH` shows stable behavior throughout execution. In `PATH`, most of the components have high utilization and EX_{LDST} observes low response time. Therefore, SCH, EX_{SP}, and EX_{LDST} are *not* tuned-down throughout execution, except for a short period of time where EX_{SP} is tuned down. Thus, our proposal tunes each component only a few times during execution, and does so at a coarse granularity. This provides robustness, and reduces the impact of PG-related timing overheads on performance and potential power savings.



(a) `RED` (b) `PATH`

Figure 9: Dynamic behavior of applications showing the tuning down of three components: SCH, EX_{SP}, EX_{LDST}. "Normal" denotes the case where all groups are turned on.

To demonstrate the effect of μC-States on the utilization of each datapath component, Figure 10 shows the average utilization of each component with our baseline and with μC-States, for two applications with different behavior. We make three observations. First, the utilization levels of the components in compute-intensive `PATH` do not change significantly with μC-States, because, as shown in Figure 9b, μC-States rarely tunes-down the pipeline components. Second, in `SCP`, IFID and SCH utilization increases. This is because: (1) since we tune-down IFID and SCH during most of the execution, component utilization increases as the same amount of work is done with fewer resources; and (2) because our definition of SCH utilization is directly related to wavefront IPC,[9] and because μC-States improves performance in `SCP` (14%), we see an increase in SCH utilization. Third, in `SCP`, the reduced scheduler throughput reduces queuing in the memory pipeline, leading to lower memory pipeline utilization. μC-States also reduces the average response time of the EX_{LDST} unit by 9%, across 34 applications.



(a) `PATH` (b) `SCP`

Figure 10: Component utilization with baseline and μC-States.

In Figure 11, we show the average fraction of time that IFID, SCH, EX_{SP}, EX_{SFU}, and EX_{LDST} units are on. Note that this value changes between 0.5 and 1, because when we tune-down a component, we power-gate or clock-gate only half of its units. A component with a value of 0.5 means that the component is tuned-down throughout the entire execution. The figure demonstrates that IFID is usually critical for performance, and it is tuned-down moderately. SCH is less critical for performance, and it is tuned-down in most of the applications except the ones that can achieve high throughput and are compute-intensive. The execute-stage

[8] For clarity, we analyze only SCH, EX_{SP}, and EX_{LDST}, because they are the most critical components for performance. We plot a representative period starting from the kernel launch.

[9] Wavefront IPC is defined as the number of wavefront instructions issued per cycle. It does not consider control divergence.

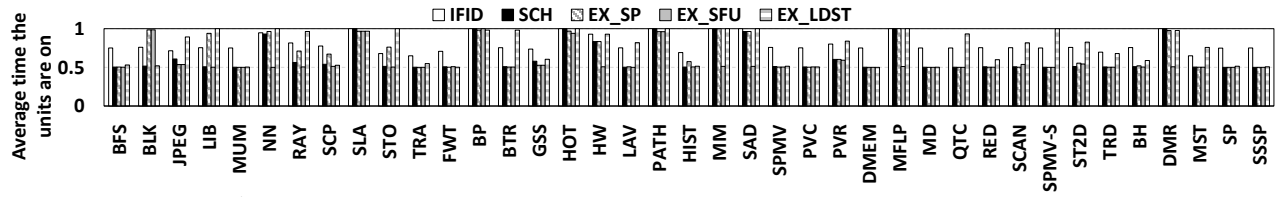


Figure 11: Average fraction of time during which IFID, SCH, EX_{SP}, EX_{SFU}, and EX_{LDST} units are on.

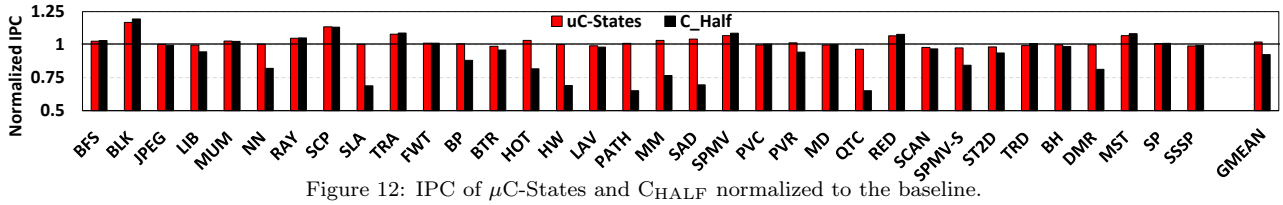


Figure 12: IPC of μC-States and C_{HALF} normalized to the baseline.

units are tuned-down according to the application's compute requirements. EX_{LDST} is often tuned-down in applications that observe high memory access latency. Overall, μC-States is effective at determining the components that are not critical for performance and tuning them down.

6.2 Application Performance

Figure 12 shows the impact of μC-States on performance (IPC), normalized to that of the baseline. We also compare our mechanism to C_{HALF}. We make four major observations. First, overall, μC-States performs 2% and 10% better than the baseline and C_{HALF}, respectively, on average. Second, μC-States does not degrade any application's performance significantly, compared to the baseline. The worst performing application is `QTC`, with only 3% performance loss. In comparison, Abdel-Majeed *et al.* [2] demonstrate in their work that their proposal causes up to 10% performance loss. Third, μC-States improves the performance of applications that benefit from C_{HALF}. In applications such as `BLK`, `SCP`, `RAY`, `TRA`, `SPMV`, `RED` and `MST`, μC-States performs better than the baseline, and as good as C_{HALF}, due to the positive effect of halving the scheduler width (as discussed in Section 3.2 for Figure 5d). These seven applications experience 9% average performance improvement with μC-States over the baseline. Fourth, μC-States performs as good as the baseline in applications that significantly benefit from more pipeline resources (i.e., the applications that perform poorly with C_{HALF}). These applications either require more compute resources (`SLA`, `HW`, `PATH`, `MM`, and `SAD`), more memory pipeline resources (`QTC` and `SPMV-S`), or both (`NN` and `DMR`). We conclude that μC-States enables a big-core GPU to become higher performance than both a small-core GPU and an unmanaged big-core GPU.

Comparison to CCWS. Our approach for improving performance is significantly different from previously proposed wavefront throttling techniques such as CCWS [76]. CCWS *reduces the number of wavefronts* that can issue requests if the first level data cache is detected to be thrashed due to high TLP, and is thus *useful for cache-sensitive applications*. In contrast, our approach does *not* consider cache thrashing. Instead, it *reduces the issue width* if we detect that the *memory system* is the bottleneck resource in the system, and is thus *useful for memory-bottlenecked applications that suffer from memory divergence*. We evaluated CCWS using two cache-insensitive applications that benefit from our technique (`BLK` and `TRA`), and observed that the application performance degrades by 4% and 2%, respectively, compared to the baseline. Our technique and CCWS aim to solve different problems, and thus can be combined to provide further benefits.

6.3 Power Consumption

Static Power. Figure 13 shows the impact of μC-States on static power consumption, normalized to that of the baseline. The figure also shows the static power of C_{HALF} as a straight line. We show the component-wise breakdown in Figure 14. We observe that μC-States reduces **static power** to the levels of that of C_{HALF} in applications that do not need all the pipeline resources for higher performance. In such applications, the majority of the benefits come from tuning down the EX_{SP} units. The only exception is `BLK`, in which EX_{SP} can be tuned down without affecting performance, but its utilization is greater than the respective threshold to be tuned down. Since performance of `BLK` improves due to the reduced issue width, we also observe significant SCH static power savings. Overall, 81% of μC-States' static power savings come from EX_{SP}, and 9% from IFID and SCH. μC-States provides 16% average static power savings.

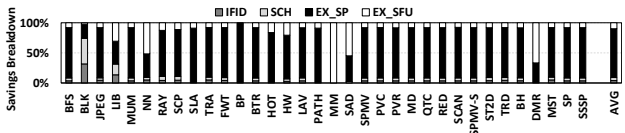


Figure 14: Distribution of static power savings with μC-States, for different pipeline components.

Dynamic Power. μC-States also reduces the average **dynamic power** consumption by 7% (not graphed). The dynamic power savings mainly come from clock-gating the pipeline registers, EX_{LDST}, operand collectors, and the register file banks. The savings from the instruction buffers is negligible. `MM`, an application that rarely utilizes EX_{LDST}, is the application that experiences the highest dynamic power savings (14%). Assuming that static/dynamic power con-

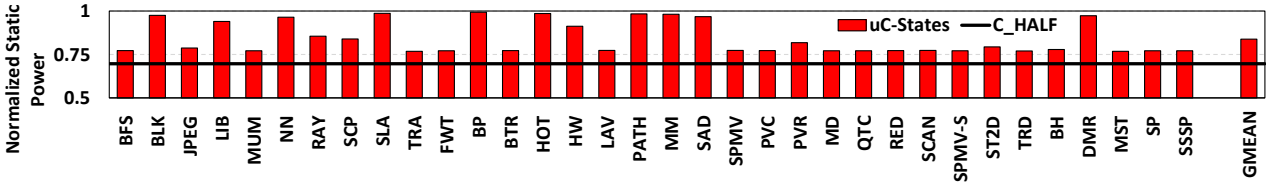


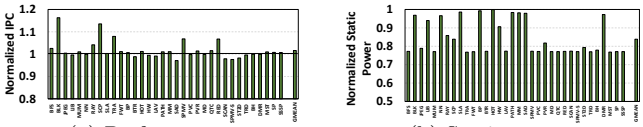
Figure 13: Static power of μC-States and C_{HALF} normalized to the baseline.

tributes to 40%/60% of the total chip power in a similar architecture [57], μC-States reduces average chip power by around 11%.

Comparison to GPUWattch. We compare μC-States to the per-lane CG strategy proposed in GPUWattch [54]. GPUWattch clock gates the individual SIMD lanes that are not used in the presence of branch divergence. Thus, it reduces power consumption only when *some of the SIMD lanes of an active component* are idle. In contrast, μC-States clock-gates a component group when it predicts that not using that group negligibly impacts application performance. We find that these two approaches are complementary. GPUWattch alone saves 5% dynamic power, and when combined with μC-States (with 7% dynamic power savings alone), the total dynamic power savings become 10%. μC-States provides static power savings on top of this, which the GPUWattch strategy cannot provide.

6.4 Sensitivity Studies

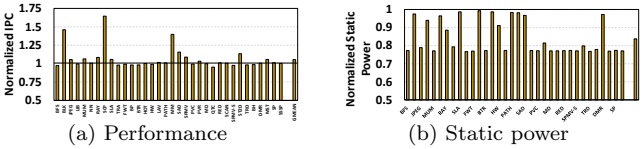
Sensitivity to register file size and the number of CTAs/core. In our baseline, we used a register file size and CTAs/core limit similar to NVIDIA Fermi [68], because prior work shows that the register file is underutilized [50, 88] and high CTA load can cause performance degradation [43]. We conduct a sensitivity study to show that μC-States works with a larger register file (64KB) and a higher CTAs/core limit (16), as in the NVIDIA Kepler architecture. Figures 15a and 15b show the performance and static power of μC-States normalized to the baseline with more register file and CTA resources, respectively. Similar to our previous results with the baseline configuration, we observe 2% performance improvement and 16% static power savings. Most of the applications that have low component utilization levels do not experience higher utilization by executing more CTAs. These applications are usually the ones that experience power savings when components are tuned-down. Because the component utilization levels are low even with more CTAs, there is still ample scope for tuning-down cores. Thus, the power benefits with μC-States do not reduce with more CTAs. On the other hand, the applications that have high component utilization levels do not provide significant power savings in our baseline, and thus the power savings they obtain with μC-States do not change significantly. We also observe that μC-States maintains the performance of these applications. We conclude that μC-States works effectively when employed in a system that has a larger register file and a higher CTAs/core limit.



(a) Performance (b) Static power

Figure 15: Performance and static power of μC-States using a larger register file and CTAs/core limit, normalized to the baseline with a larger register file and CTAs/core limit.

Sensitivity to wavefront scheduling mechanisms. We conduct a study to show that μC-States can be employed with different wavefront schedulers [26, 40, 41, 53, 67, 76, 77]. We change our baseline GTO wavefront scheduler [76] to a round-robin (RR) scheduler [26]. Figure 16a and Figure 16b show the performance and static power of μC-States normalized to the baseline with RR, respectively. Overall, we observe that the performance benefits of μC-States increase to 5%, and static power savings remain at 16% with the RR scheduler.[10] Thus, our mechanism can work with any scheduler, but its power and performance benefits may vary with different schedulers. This is in contrast to recent work (e.g., Abdel-Majeed *et al.* [2]) where the power saving mechanisms are scheduler specific. Because the major goal of a warp scheduler is to improve performance, and not to make the utilization of pipeline components similar, we expect PG/CG opportunities to remain similar with different schedulers. Thus, our mechanism can identify any not-so-performance-critical components for power savings regardless of the choice of the wavefront scheduler.



(a) Performance (b) Static power

Figure 16: Performance and static power of μC-States with the RR scheduler normalized to the baseline with RR scheduler.

Sensitivity to timing overhead parameters. We evaluated μC-States with different break-even and wake-up time parameters (Section 2.3) to show that our mechanism is not sensitive to these overheads. Taking a more conservative approach compared to Warped Gates [2], which uses break-even times of 9, 14, and 19 cycles, and wake-up times of 3, 6, and 9 cycles, we experimented with higher break-even times of 19, 29, and 39 cycles, and higher wake-up times of 9, 15, and 21 cycles. Since μC-States works at a much coarser granularity, and since it does not re-tune components frequently, we did not observe any significant impact on its power and performance benefits.

Sensitivity to the time interval between phases. In μC-States, the interval between two phases is 2048 cycles. Setting this parameter to 4K, 8K, and 16K cycles reduces performance benefits to 1.7%, 1.6%, and 1.1%, and static power benefits to 14%, 13.5%, and 12%, respectively. This reduction is because the responsiveness of μC-States to dynamic changes in the application behavior reduces with a longer interval.

7. HETEROGENEOUS-CORE GPU

Although we used our detailed GPU datapath component characterization to propose a new PG/CG technique, it can also be used for other purposes. One example is scheduling and design decisions in a GPU that contains heterogeneous (both small and big) cores. We provide a case study that shows having heterogeneous GPU cores is beneficial from power, performance and area aspects in a multiple application environment, when combined with a simple application mapping mechanism driven by our bottleneck analysis.

We simulate a heterogeneous GPU system consisting of 8 big cores used in our baseline, and 8 C_{HALF} cores. We compare this system to two different homogeneous configurations: our baseline (16 *big cores*) and C_{HALF} (16 *small cores*). We use nine workloads formed by mixing three applications that prefer C_{HALF} (`BLK`, `SCP`, and `FWT`) with three applications that prefer baseline cores (`MM`, `SAD`, and `NN`). The applications that prefer C_{HALF} are mapped to small cores. The other applications are mapped to the large cores. Figure 17 shows the performance of our heterogeneous-core GPU and C_{HALF} normalized to our baseline. We observe that 1) C_{HALF} causes 12% average performance loss over

[10] RR has been shown to perform poorly compared to GTO [76]. μC-States provides higher performance benefits for applications that prefer lower issue width with RR, compared to GTO.

the baseline, mainly because of the performance loss experienced by the applications that prefer more pipeline resources than C_{HALF} provides, and 2) the heterogeneous-core GPU performs as good as the baseline with fewer resources. Thus, having a heterogeneous-core GPU for executing multiple applications can provide the same performance as the baseline with 15% lower static power and lower area, when applications are intelligently mapped to the heterogeneous cores. We defer a detailed study of design issues to a future work.

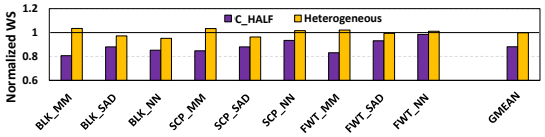


Figure 17: Performance of an example heterogeneous GPU and C_{HALF}, normalized to the baseline.

8. RELATED WORK

To our knowledge, this is the first work that performs a study on the sensitivity of the entire GPU datapath on performance, proposes a comprehensive power gating and clock gating mechanism for GPUs, and shows a proof-of-concept case study for *heterogeneous cores* within GPUs.

Power Saving Techniques. Abdel-Majeed *et al.* [2] propose a wavefront scheduler that clusters the same type of instructions and issues them to the execution units closeby in time before switching to other instruction type(s). This increases the idle time of specific execution units, and, thus, the PG opportunities. Our mechanism is different in four major ways: 1) instead of only execution units, we target the *whole* pipeline, and also the interactions between different pipeline components, based on well-established ideas from queuing theory to identify performance bottlenecks in GPU cores, 2) our scheme does *not* rely on long idle periods and is independent of the wavefront scheduler choice, 3) our strategy is *not* sensitive to timing and energy overhead parameters because it works at a coarser granularity, and 4) our mechanism does *not* cause significant performance loss for any of our applications, and improves performance of applications where the memory system is the bottleneck.

There are white papers on power saving mechanisms by AMD [4] and NVIDIA [70]. Our approach of using CG for some components instead of PG, to maintain execution state, is similar to AMD's approach. In AMD's design [4], some blocks are always-on (AON), and some are on-and-off (ONO), whereas we employ CG for the blocks that we do not power-gate. ONO blocks are 3D-graphics blocks, and they can be power-gated during runtime, whereas we focus on compute-related datapath components. NVIDIA Tegra 4 [70] employs CG and DVFS techniques for compute and graphics units when executing graphics and compute applications, respectively. Our mechanism works at a finer time and component granularity, building on our detailed analyses of fine-grained component utilization in Section 3.2.

Prior works in the context of CPUs clock-gate pipeline components based on resource usage [8, 55, 58], use reduced-size issue buffers for higher energy efficiency [25], dynamically tune resources for higher energy efficiency [3, 10, 74], and use a combination of fetch gating and issue queue adaptation [12]. None of these works improve performance. Hong *et al.* [31] propose core PG mechanisms to allow only a set of GPU cores to be active. Lee *et al.* [52] statically analyze the best GPU configuration under a fixed power budget. Ausavarungnirun *et al.* [6] propose a memory controller design that improves energy efficiency for a heterogeneous CPU-GPU architecture. Other prior works [1, 27, 28, 37, 54, 88–90] propose power efficiency techniques for memory, caches, register files and pipeline components in the context of GPUs. None of these works develop a comprehensive power and clock gating mechanism like this paper does for the entire GPU pipeline.

Configurable and Heterogeneous Architectures. Many works design more configurable and heterogeneous architectures for better power consumption and/or performance. Park *et al.* [72] adaptively customize the SIMD lanes in a multi-threaded architecture to improve energy-efficiency. Several works [35, 45, 46] propose mechanisms to morph larger cores into multiple smaller cores, or combine smaller cores to make larger ones. Van Craeynest *et al.* [87] propose a techniques to estimate an application's performance while executing the application on a larger core. Several works from Suleman *et al.* [85, 86], Joao *et al.* [38, 39] and Du Bois *et al.* [20] propose techniques that accelerate the bottleneck portions of multi-threaded applications on large cores of a heterogeneous multi-core processor. Guevara *et al.* [29] explore various design points for a heterogeneous architecture consisting of Xeon and Atom processors within a certain power budget. Mishra *et al.* [60, 61] design heterogeneous interconnects to maximize performance and power efficiency. None of these works consider fine-grained power or clock gating of GPU pipeline components or having heterogeneous cores in GPUs.

9. CONCLUSION

We introduce μC-States, a new, comprehensive power- and clock-gating framework for GPUs that improves both power consumption and performance. μC-States minimizes power consumption by turning off, or tuning-down, datapath components that are not bottlenecks for the performance of the running application. Our mechanism is based on a new, rigorous, queuing-theory-based analysis of datapath component utilization of various applications in big-core GPUs, which demonstrates that: (1) many GPU datapath components are heavily underutilized, and this underutilization varies across components and within and across applications, and (2) having more resources in a GPU core can sometimes degrade application performance by increasing contention in the memory system. We show that our analysis could also be useful in guiding scheduling and design decisions in a heterogeneous-core GPU with both small and big cores. Our case study demonstrates that a heterogeneous-core GPU can provide lower power consumption and smaller chip area at the same performance as a conventional big-core GPU design. We believe and hope that the analysis and mechanisms provided in this paper can be useful for developing other new analyses and optimization techniques for more efficient GPU and heterogeneous architectures.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their feedback. This research is supported in part by NSF grants #1205618, #1213052, #1212962, #1302225, #1302557, #1317560, #1320478, #1320531, #1409095, #1409723, #1439021, #1439057, and #1526750. Adwait Jog acknowledges the start-up grant from the College of William and Mary. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] M. Abdel-Majeed and M. Annavaram, "Warped Register File: A Power Efficient Register File for GPGPUs," in *HPCA*, 2013.

[2] M. Abdel-Majeed *et al.*, "Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs," in *MICRO*, 2013.

[3] D. H. Albonesi *et al.*, "Dynamically Tuning Processor Resources with Adaptive Processing," *IEEE Computer*, 2003.

[4] AMD, "AMD: Power-gating in a High-Performance GPU," 2009. Available: http://www.powerforward.org/media/p/125.aspx

[5] AMD, "AMD Graphics Cores Next (GCN) Architecture," 2012. Available: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf

[6] R. Ausavarungnirun *et al.*, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.

[7] R. Ausavarungnirun *et al.*, "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance," in *PACT*, 2015.

[8] R. I. Bahar and S. Manne, "Power and Energy Reduction via Pipeline Balancing," in *ISCA*, 2001.

[9] A. Bakhoda *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[10] R. Balasubramonian *et al.*, "Memory Hierarchy Reconfiguration for Energy and Performance in General-purpose Processor Architectures," in *MICRO*, 2000.

[11] M. Burtscher *et al.*, "A Quantitative Study of Irregular Programs on GPUs," in *IISWC*, 2012.

[12] A. Buyuktosunoglu *et al.*, "Energy Efficient Co-adaptive Instruction Fetch and Issue," in *ISCA*, 2003.

[13] K. K.-W. Chang *et al.*, "HAT: Heterogeneous Adaptive Throttling for On-Chip Networks," in *SBAC-PAD*, 2012.

[14] N. Chatterjee *et al.*, "Managing DRAM Latency Divergence in Irregular GPGPU Applications," in *SC*, 2014.

[15] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.

[16] A. Danalis *et al.*, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *GPGPU*, 2010.

[17] R. Das *et al.*, "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems," in *HPCA*, 2013.

[18] R. Das *et al.*, "Application-aware Prioritization Mechanisms for On-chip Networks," in *MICRO*, 2009.

[19] R. Das *et al.*, "AERGIA: Exploiting Packet Latency Slack in On-chip Networks," in *ISCA*, 2010.

[20] K. Du Bois *et al.*, "Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior," in *ISCA*, 2013.

[21] E. Ebrahimi *et al.*, "Fairness via Source Throttling: a Configurable and High-performance Fairness Substrate for Multi-core Memory Systems," in *ASPLOS*, 2010.

[22] E. Ebrahimi *et al.*, "Parallel Application Memory Scheduling," in *MICRO*, 2011.

[23] S. Eyerman and L. Eeckhout, "System-level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.

[24] S. Eyerman and L. Eeckhout, "Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance," *IEEE Comput. Arch. Letter*, vol. 13, no. 2, pp. 93–96, Jul. 2014.

[25] D. Folegnani and A. González, "Energy-effective Issue Logic," in *ISCA*, 2001.

[26] W. Fung *et al.*, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *MICRO*, 2007.

[27] M. Gebhart *et al.*, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *ISCA*, 2011.

[28] S. Z. Gilani *et al.*, "Exploiting GPU Peak-power and Performance Tradeoffs through Reduced Effective Pipeline Latency," in *MICRO*, 2013.

[29] M. Guevara *et al.*, "Navigating Heterogeneous Processors with Market Mechanisms," in *HPCA*, 2013.

[30] B. He *et al.*, "Mars: A MapReduce Framework on Graphics Processors," in *PACT*, 2008.

[31] S. Hong and H. Kim, "An Integrated GPU Power and Performance Model," in *ISCA*, 2010.

[32] M. Houston, "Anatomy of AMD's TeraScale Graphics Engine," 2008. Available: http://s08.idav.ucdavis.edu/houston-amd-terascale.pdf

[33] Z. Hu *et al.*, "Microarchitectural Techniques for Power Gating of Execution Units," in *ISLPED*, 2004.

[34] Intel, "Energy-Efficient Platforms - Considerations for Application Software and Services," 2011. Available: http://www.intel.com/content/dam/doc/white-paper/energy-efficient-platforms-2011-white-paper.pdf

[35] E. Ipek *et al.*, "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors," in *ISCA*, 2007.

[36] R. Jain, "The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling," *New York: John Willey*, 1991.

[37] N. Jing *et al.*, "An Energy-efficient and Scalable eDRAM-based Register File Architecture for GPGPU," in *ISCA*, 2013.

[38] J. A. Joao *et al.*, "Bottleneck Identification and Scheduling in Multithreaded Applications," in *ASPLOS*, 2012.

[39] J. A. Joao *et al.*, "Utility-based Acceleration of Multithreaded Applications on Asymmetric CMPs," in *ISCA*, 2013.

[40] A. Jog *et al.*, "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*, 2013.

[41] A. Jog *et al.*, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

[42] A. Jog *et al.*, "Exploiting Core Criticality for Enhanced Performance in GPUs," in *SIGMETRICS*, 2016.

[43] O. Kayiran *et al.*, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *PACT*, 2013.

[44] O. Kayiran *et al.*, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.

[45] Khubaib *et al.*, "MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP," in *MICRO*, 2012.

[46] C. Kim *et al.*, "Composable Lightweight Processors," in *MICRO*, 2007.

[47] H. Kim *et al.*, "Bounding Memory Interference Delay in COTS-based Multi-Core Systems," in *RTAS*, 2014.

[48] Y. Kim *et al.*, "ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.

[49] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[50] N. Lakshminarayana and H. Kim, "Spare Register Aware Prefetching for Graph Algorithms on GPUs," in *HPCA*, 2014.

[51] C. J. Lee *et al.*, "Improving Memory Bank-level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.

[52] J. Lee *et al.*, "Improving Throughput of Power-constrained GPUs Using Dynamic Voltage/Frequency and Core Scaling," in *PACT*, 2011.

[53] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads," in *PACT*, 2014.

[54] J. Leng *et al.*, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ISCA*, 2013.

[55] H. Li *et al.*, "Deterministic Clock Gating for Microprocessor Power Reduction," in *HPCA*, 2003.

[56] J. D. Little, "A Proof for the Queuing Formula," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.

[57] J. Lucas *et al.*, "How a Single Chip Causes Massive Power Bills GPUSimPow: A GPGPU Power Simulator," in *ISPASS*, 2013.

[58] S. Manne *et al.*, "Pipeline Gating: Speculation Control for Energy Reduction," in *ISCA*, 1998.

[59] J. Meng *et al.*, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *ISCA*, 2010.

[60] A. K. Mishra *et al.*, "A Heterogeneous Multiple Network-on-Chip Design: An Application-Aware Approach," in *DAC*, 2013.

[61] A. K. Mishra *et al.*, "A case for heterogeneous on-chip interconnects for CMPs," in *ISCA*, 2011.

[62] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.

[63] T. Moscibroda and O. Mutlu, "Distributed Order Scheduling and Its Application to Multi-core DRAM Controllers," in *PODC*, 2008.

[64] S. P. Muralidhara *et al.*, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.

[65] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.

[66] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.

[67] V. Narasiman *et al.*, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *MICRO*, 2011.

[68] NVIDIA, "Fermi: NVIDIA's Next Generation CUDA Compute Architecture," 2011.

[69] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.

[70] NVIDIA, "Tegra 4 Family GPU Architecture," 2013. Available: http://www.nvidia.com/docs/IO/116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf

[71] G. P. Nychis *et al.*, "On-chip networks from a networking perspective: congestion and scalability in many-core interconnects," in *SIGCOMM*, 2012.

[72] Y. Park *et al.*, "Libra: Tailoring SIMD Execution Using Heterogeneous Hardware and Dynamic Configurability," in *MICRO*, 2012.

[73] M. K. Qureshi *et al.*, "A Case for MLP-Aware Cache Replacement," in *ISCA*, 2006.

[74] P. Ranganathan *et al.*, "Reconfigurable Caches and Their Application to Media Processing," in *ISCA*, 2000.

[75] S. Rixner *et al.*, "Memory Access Scheduling," in *ISCA*, 2000.

[76] T. G. Rogers *et al.*, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.

[77] T. G. Rogers *et al.*, "Divergence-Aware Warp Scheduling," in *MICRO*, 2013.

[78] K. Roy *et al.*, "Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-submicrometer CMOS Circuits," *Proceedings of the IEEE*, 2003.

[79] K. Roy *et al.*, "Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-submicrometer CMOS Circuits," *Proceedings of the IEEE*, vol. 91, no. 2, pp. 305–327, 2003.

[80] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," in *ASPLOS*, 2000.

[81] J. A. Stratton *et al.*, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois, at Urbana-Champaign, Tech. Rep. IMPACT-12-01, March 2012.

[82] L. Subramanian *et al.*, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.

[83] L. Subramanian *et al.*, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.

[84] L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013.

[85] M. A. Suleman *et al.*, "Data Marshaling for Multi-core Architectures," in *ISCA*, 2010.

[86] M. A. Suleman *et al.*, "Accelerating Critical Section Execution with Asymmetric Multi-core Architectures," in *ASPLOS*, 2009.

[87] K. Van Craeynest *et al.*, "Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE)," in *ISCA*, 2012.

[88] N. Vijaykumar *et al.*, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *ISCA*, 2015.

[89] B. Wang *et al.*, "Exploring Hybrid Memory for GPU Energy Efficiency Through Software-hardware Co-design," in *PACT*, 2013.

[90] W.-k. S. Yu *et al.*, "SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-grained Multi-threading," in *ISCA*, 2011.

[91] J. Zhao *et al.*, "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems," in *MICRO*, 2014.

[92] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order," no. U.S. Patent Number 5,630,096, Sep. 1997.