Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs

Mohammad Abdel-Majeed* abdelmaj@usc.edu
Daniel Wong* wongdani@usc.edu
Murali Annavaram annavara@usc.edu
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089

ABSTRACT

With the widespread adoption of GPGPUs in varied application domains, new opportunities open up to improve GPGPU energy efficiency. Due to inherent application-level inefficiencies, GPGPU execution units experience significant idle time. In this work we propose to power gate idle execution units to eliminate leakage power, which is becoming a significant concern with technology scaling. We show that GPGPU execution units are idle for short windows of time and conventional microprocessor power gating techniques cannot fully exploit these idle windows efficiently due to power gating overhead. Current warp schedulers greedily intersperse integer and floating point instructions, which limit power gating opportunities for any given execution unit type. In order to improve power gating opportunities in GPGPU execution units, we propose a Gating Aware Two-level warp scheduler (GATES) that issues clusters of instructions of the same type before switching to another instruction type. We also propose a new power gating scheme, called Blackout, that forces a power gated execution unit to sleep for at least the break-even time necessary to overcome the power gating overhead before returning to the active state. The combination of GATES and Blackout, which we call Warped Gates, can save 31.6% and 46.5% of integer and floating point unit static energy. The proposed solutions suffer less than 1% performance and area overhead.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*

*Abdel-Majeed and Wong made equal contributions to this work.

MICRO'46 December 7-11, 2013, Davis, CA, USA

ACM 978-1-4503-2638-4/13/12 ...$15.00.

General Terms

Design, Performance

Keywords

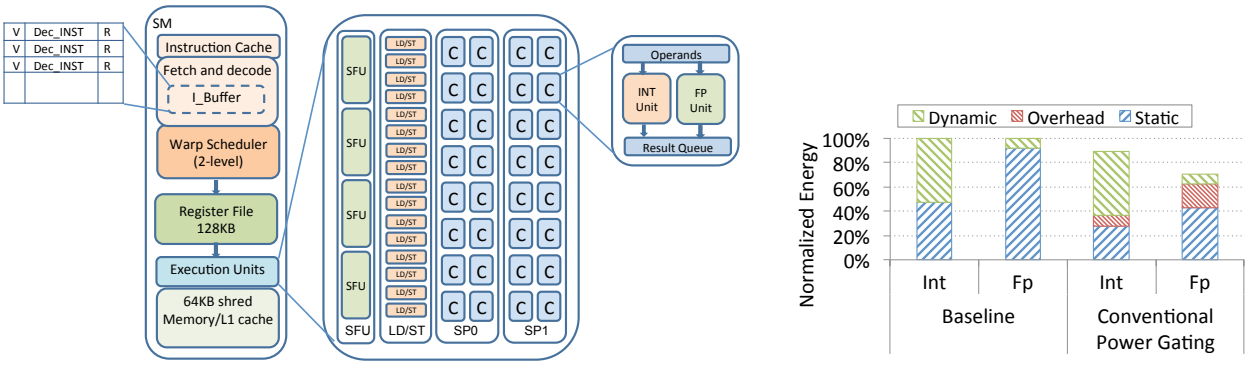
GPGPUs, Warp Scheduling, Power Gating

1. INTRODUCTION

Graphics processing units (GPUs) are massively parallel processors that are designed to run workloads with thousands of concurrent threads. By using a SIMT (single instruction multiple threads) execution model, GPUs can execute the same instruction but with hundreds of different data operands concurrently. The simplified control logic coupled with massive parallelism can achieve hundreds of GFLOPs of peak throughput at low power. Due to their high throughput and excellent performance/watt, GPUs are being re-architected to run applications beyond traditional multimedia, such as modeling of physical phenomena and large scale data analytics. When GPU designs are stretched to become general purpose GPUs (GPGPUs), they suffer reduced performance/watt due to several reasons. In [5, 12, 14] the authors showed that there is a wide variation in resource utilization when GPGPUs run applications with diverse parallelism demands. This variation leads to resource underutilization in GPGPUs which reduces power efficiency. GPGPU power efficiency is a growing concern as reflected by the recent research activity in this area [5, 12, 16, 22].

In [16], the authors showed component-level power breakdown for the NVidia GTX480 GPGPU. Their results showed that execution units consume 20.1% of the total platform power, followed by memory (17.8%) and register file (13.4%). Using the GPUWattch tool [16] we measured the static and and dynamic power for integer and floating point units for NVidia GTX480 GPGPU while running a range of GPGPU workloads (experimental details provided in Section 7). In Figure 1b the first two bars show the distribution of static and dynamic power. These results show that static energy accounts for about 50% of the total energy consumed in integer execution units, and more than 90% in floating point units. As technology scaling continues, handling component leakage power will become increasingly important.

Prior research has reduced static power consumption in memory and register files of GPGPUs [24, 5], but to the best of our knowledge, techniques for reducing static power of execution units within a GPGPU have not been explored. In this research, we show that due to inherent application-



(a) GTX480 SM architecture

(b) Power breakdown for execution units

Figure 1: Baseline Nvidia GTX480 details

level inefficiencies GPGPU execution units experience significant idle time. We propose to power gate idle execution units to reduce the static power. Power gating allows unused execution units to be turned off (or gated) when not in use thereby mitigating static power consumption. Power gating has been widely used in microprocessors [13], caches [11], and NOCs [9]. Power gating has also been used in GPUs, but at a coarser granularity of gating whole streaming multiprocessor (SM) [22]. However, when applying power gating at a finer granularity of execution units in a GPGPU several new challenges arise. The focus of this work is to first describe these challenges and propose solutions to address them.

The following are the contributions of this work:

Conventional power gating limitations for GPGPU execution units: We found that GPGPU execution units tend to be idle only for very short periods (majority of time less than 10 cycles), which conventional power gating techniques [13] are unable to exploit. We show that the Two-level warp scheduler [12] used in current GPGPUs greedily schedules instructions to execution units which results in short switching cycles between different types of execution units, such as floating point, integer, special function units, and load/store units. Hence, no single execution unit stays idle for sufficiently long period to amortize the cost of power gating overhead.

Gating Aware Two-level warp scheduler: To address the inefficiencies of GPGPU scheduler in extracting idle periods, we present a gating-aware Two-level warp scheduler (GATES). GATES prioritizes issuing clusters of instructions that require the same type of execution unit for longer intervals before switching to a new instruction type. Thus GATES stretches the length of idle periods for each execution unit type. GATES can be built through low overhead extensions to current Two-level warp scheduler.

Blackout power gating: While GATES extend the idle periods, there are still many idle windows that are shorter than the break-even time. To address this concern we propose a new power gating controller called Blackout. Blackout places new limitations on power gating state transitions. In particular, Blackout forces an execution unit to be gated for at least as many cycles as it takes to recoup the power gating overhead. This policy is applied even when there are instructions that are waiting to use the execution unit during the gated time interval. We show that forcing an execution unit to stay gated even when there is a ready instruction does not hurt performance in GPGPUs primarily because of the abundant heterogeneity of available instruction types.

Adaptive idle detect: Finally, we present a runtime approach to limit performance loss of Blackout for certain workloads by adaptively adjusting the amount of time a unit is idle before the unit is gated. We call this approach Adaptive idle detect. Adaptive idle detect relies on easy to obtain runtime performance metrics to determine the amount of time a unit must be idle before gating is enabled. We combine GATES and Blackout to create a coordinated power gating scheme, called *Warped Gates*, with virtually no performance loss.

The rest of the paper is organized as follows: Section 2 provides background about the GPGPU architecture and conventional power gating techniques. Section 3 highlights why current power gating techniques are inefficient when applied to GPGPU execution units and motivates the need for more efficient techniques. Sections 4, 5, and 6 discusses the proposed techniques and the required microarchitectural support. Section 7 presents the simulation methodology and results. We discuss related work in section 8 and conclude in section 9.

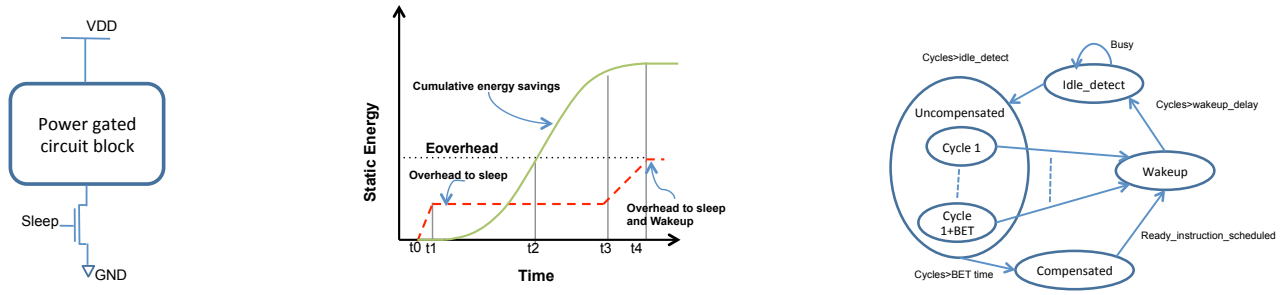
2. BACKGROUND

This section provides an overview of the baseline GPGPU architecture and the conventional power gating technique that form the foundation for the work presented in this paper.

2.1 Baseline GPGPU architecture

This work uses an Nvidia GTX480 Fermi GPGPU as the baseline architecture. GTX480 consists of a set of 15 Streaming multiprocessors (SMs). The overall architecture for the SM is shown in figure 1a. Each SM is comprised of a large register file, thread scheduler(s) and execution units. Each SM has its own 64KB shared memory and cache. Fermi supports up to 48 active warps per SM. Each warp comprises of 32 threads executing the same instruction in a lockstep manner, also called Single Instruction Multiple Thread (SIMT) execution model. Thus, there are a total of 1,536 active threads per SM. The execution flow within each SM can be broadly divided into three stages:

Fetch and decode: The instruction fetch and decode logic has an instruction buffer that stores decoded instructions. The instruction buffer is divided between warps. Each entry in the instruction buffer has a valid bit (V in figure 1a),



(a) Gating circuit (b) Break-even time (c) Power gating state machine

Figure 2: Power gating overview

the decoded instruction bits (Dec_INST), and a ready bit (R) to indicate that the instruction is ready for execution. The decoded instruction field includes the instruction type that determines which execution unit type that instruction requires for execution, namely an integer unit (INT), floating point unit (FP), special purpose functional unit (SFU), or load/store unit (LD/ST). The ready bit is set at the issue stage using the scoreboard logic and resource availability. The scoreboard logic maintains information regarding whether input operands for an instruction are already available in register file or if they are being generated by an instruction already in the pipeline.

Two-level warp scheduler: In this work the baseline uses the Two-level warp scheduler [12]. In the Two-level warp scheduler, all warps waiting on long latency events, such as memory accesses, are placed into a *pending warps set*. The *active warps set* holds all the warps that are either waiting on a short latency dependency or whose input operands are already available in the register file. The Two-level warp scheduler only issues an instruction from the active warps set when all the input operands are ready for that instruction. In order to improve performance in GPGPUs, more than one scheduler can be integrated within an SM. In GTX480, two schedulers are integrated in each SM and each scheduler can issue one ready warp per cycle as long as there are no structural hazards.

Execution units: Figure 1a shows the block diagram of the execution units inside each SM. The execution units contain two shader processors (SP), 16 LD/ST units for memory operations and four SFUs for the special arithmetic functions like sine and cosine. Each SP contains 16 SIMT lane cores (called CUDA cores) running at double the core clock frequency. As a result, each SP can run 32 concurrent threads over one issue cycle. Each CUDA core contains one integer unit and one floating point unit.

2.2 Power Gating

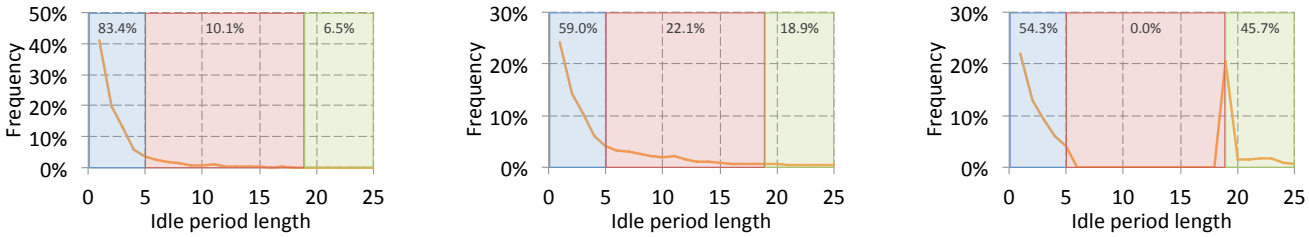
Power gating is a technique that is used to cut off the leakage current that flows through a circuit block. Power gating is implemented by adding a properly sized header transistor (between Vdd and the circuit block) or footer transistor (between the circuit block and Gnd) as shown in figure 2a. When the transistor is OFF, the circuit block will be power gated and there will be no path from Vdd to Gnd, resulting in a very small leakage current. When the power gating transistor is ON, then the circuit block will operate normally.

Figure 2b illustrates the cumulative energy savings and energy overheads when applying conventional power gating as described in [13]. The solid green curve represents the *cumulative energy savings* from reducing leakage in the circuit. At time t_0 the power gating signal is enabled and the switch will turn off at t_1. The leakage energy savings begin increasing at time t_1 and will continue to accumulate with time as seen by the raising solid curve. The leakage savings stop at time t_4 when the sleep transistor is turned back ON to bring the circuit back to active state and the circuit block wakes up.

However, there is a dynamic energy penalty for switching the sleep transistor on and off. The red dashed curve represents the *cumulative energy overhead* due to power gating. The black dotted line labeled *Eoverhead* shows the total energy overhead for each power gating instance. Where the dotted black line intersects with the energy savings curve (at t_2) is called the break-even time (BET), which is the minimum number of consecutive power gated cycles that are required to compensate for the energy overhead of the power gating switch [13]. If the block is turned ON before t_2, then the power gating overhead exceeds the leakage energy savings resulting in net negative energy savings.

The time between t_3 and t_4 is the wakeup delay, which is the minimum number of cycles required to return the operating voltage range to Vdd. At t_4, the functional unit is fully powered up and operational. Recent studies on power gating of execution blocks has estimated the wakeup delay to be around 3 cycles and break-even time to be between 9 and 24 cycles [13]. Since the circuit block cannot wakeup instantly when a request arrives, the wakeup penalty can lead to performance degradation. In our experimental evaluation we use wakeup delay of 3 cycles. In [13] break-even values of 9,14,19 and 24 were explored. In our experimental evaluation we use a value of 14 cycles as the break-even time.

Power gating state machine: Figure 2c shows the state machine for the power gating controller. As long as the block is busy, the block will stay in the $Idle\_detect$ state. As soon as the block is idle for at least *idle-detect* cycles, the block will be moved into the $Uncompensated$ state and the circuit block will be power gated. The $Uncompensated$ state means that the power gating overhead has not been compensated and the energy overhead of operating the sleep switch exceeds the total leakage savings. The block will stay in the $Uncompensated$ state as long as the number of cycles is less than the break-even time. After the break-even time, the block is moved to the $Compensated$ state. In conventional power gating, the controller will move the block to the $Wakeup$ state at any time if the block is needed for execution. In the $Wakeup$ state, the block needs *wakeup delay* cycles before switching back to the $Idle-detect$ state again.



(a) Conventional power gating (b) Gating Aware scheduler (c) Blackout Power Gating

Figure 3: Idle period length distribution with 5 cycle idle-detect and 14 cycle break-even time for `hotspot`

If the wakeup happens when the block is in *Uncompensated* state, then the total energy saved due to the power gating attempt will be negative.

3. POWER GATING CHALLENGES IN GPGPUS

In this paper we will focus on leakage energy saving for CUDA cores, comprising of INT and FP units. The techniques presented can also be applied to SFUs. However, SFU instructions are relatively rare and hence, conventional power gating scheme will be sufficient to recover most of the wasted leakage energy in SFUs. Furthermore, SFUs account for only 2.5% of total execution unit static power consumption. The relatively large number of INT and FP units, make them the primary target for leakage energy savings, compared to SFUs.

As described earlier, Figure 1b shows the average energy breakdown for the INT and FP units. The first two bars show the baseline energy consumption breakdown when no power gating is applied. Static energy accounts for nearly 50% of the total INT energy and 90% of the FP energy. The large proportion of static energy in the FP unit is due to the relatively low usage of the FP units compared to the INT units. Thus, there is a large potential for reducing static energy with power gating of these two units.

The last two bars show the energy breakdown after applying conventional power gating [13] with an idle-detect time of 5 cycles and a break-even time of 14 cycles. Conventional power gating reduces INT and FP unit energy by 11% and 29%, respectively. However, even after applying conventional power gating the static power consumption still accounts for 31% of total INT unit energy consumption and 61% of the total FP unit energy consumption. It is important to bring attention to the power gating overhead component in the last two bars. This component of energy consumption comes from the extra power burned to turn on and off the sleep transistor. Power gating overhead accounts for 11% and 29% of the INT and FP units overall energy, respectively. We will highlight the reasons for high power gating overhead and missed power gating opportunity in this section and present solutions to alleviate these issues in the next section.

3.1 Need for longer idle periods

In order for power gating to be effective, it is not sufficient to just have idle periods, rather it is critical to have long enough idle periods so that power gating can achieve net energy savings. In traditional microprocessor functional units, the majority of idle periods lengths are many 10s of cycles [10]. Since the idle duration is longer than the typical break-even time, conventional power gating is an excellent option to reduce static power in traditional microprocessors.

GPGPUs typically have many ready warps with a diverse instruction mix ready for execution. The Two-level warp scheduler schedules ready warps from the active warps set without taking into consideration what other instruction types have been issued prior to the current issue cycle. As a result, different instruction types get issued within a short scheduling window. Interspersing different instruction types results in idle period lengths in the order of a few cycles for any given execution unit type in GPGPUs. Figure 3a shows the idle period length distribution in cycles of a representative GPGPU benchmark, `hotspot` [8]. The data in the figure is partitioned into three regions. The left-most region (in blue) represents the idle period lengths which falls within *idle-detect* time. The middle region (in red) represents the idle period lengths which falls within *idle-detect* and *idle-detect + BET*. The right most region (in green) represents idle period lengths which are longer than *idle-detect + BET* cycles. For this specific benchmark, 83.4% of the idle periods are less than the *idle-detect*, and only 6.5% of the idle periods are longer than *idle-detect + BET* cycles. In conventional power gating, only those idle windows that are in the last category lead to positive energy savings. The first category represents wasted idle periods that cannot be power gated due to their short duration. The middle range represents the set of idle periods that will result in net energy loss (or at best energy neutral) if conventional power gating is used. While the results presented in this figure correspond to the `hotspot` benchmark, similar patterns can be found in all other benchmarks in our experiments. What is important to note here is that unlike conventional microprocessor functional units, the majority of idle periods are only a few cycles long.

Figure 4 illustrates the shortcomings of current warp scheduling and its implication on power gating techniques. In this simplified illustration, the active warps set contains 10 warps with a mix of integer and floating point instructions. The order of instructions in the set is shown in the figure at the top. We assume each instruction is a simple add instruction, each instruction has a latency of four cycles, and initiation interval is one cycle. These are the default parameters in GPGPUSim's configuration file for Fermi [6]. The Two-level warp scheduler would issue warps from the front of the active warps set without regard to instruction type as shown in the center figure. For instance, in cycle three an FP instruction is available at the top of the active warps set which is issued to the FP unit. As a result, the FP unit, which was idle during the first two cycles, starts executing an instruction in cycle three. Similarly, the floating point

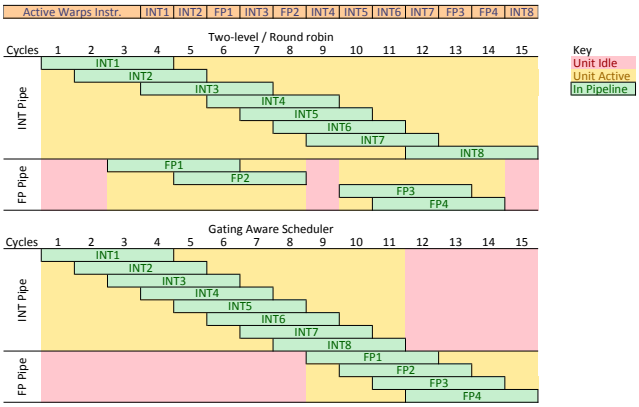


Figure 4: Effect of Warp Scheduler on Idle Cycles

unit is assigned another instruction to execute in cycle five. After the first two floating point instructions are completed, the FP unit has one idle cycle followed by the execution of two additional instructions. As a result, during the entire 15 cycle window the FP pipeline experienced three idle periods of two, one, and one cycle(s), which are too short for conventional power gating to take advantage of.

4. GATING-AWARE TWO-LEVEL SCHEDULER (GATES)

The data presented in the previous section points to the need for a technique that can coalesce short idle periods (the first two idle period ranges in Figure 3a) into fewer but longer idle periods. Such a technique will shift the distribution of idle periods into the right most range in the figure where power gating is beneficial. To accomplish this goal we propose a gating-aware Two-level scheduler (GATES) which takes into account previously issued instruction types in determining which ready warp to issue next. GATES prioritizes issuing the same instruction type as was issued in prior issue cycle to coalesce the utilization and idle periods of integer and floating point units. GATES will keep issuing instructions from the same type as long as there are ready warps in the active warps set. GATES switches to a warp with different instruction type when there are no more ready warps in the active warps set with the same instruction type as the one issued in the previous issue cycle. Note that GATES does not lead to starvation as long as there is some dependency between INT and FP instructions. Eventually all independent instructions of a given type will be exhausted leaving room for the other instruction type to start issuing. The designer can also set a large maximum switching time threshold to force a scheduler to switch priorities at the end of the threshold.

When GATES is applied to our previous illustrative example in figure 4, all the INT instructions would be issued first, and when there are no additional INT instructions, the FP instructions would be issued. As shown in the bottom figure the INT pipeline now has four consecutive idle cycles, while the FP pipeline has eight consecutive idle cycles. By coalescing instruction type, we remove isolated bubbles from the execution unit pipeline and create longer idle periods to increase opportunities for power gating.

GATES would be effective if there exists sufficient number of active warps with a good instruction mix of integer and floating point instructions to allow the scheduler ample opportunities to rearrange warps. Figure 5a shows the instruction mix for a large number of GPGPU workloads. Except for a couple of pure integer workloads (such as `lavaMD`), most benchmarks have a sufficient mix of integer and floating point instruction types.

Figure 5b shows the maximum and average number of active warps available during runtime. Majority of benchmarks have a large number of active warps during runtime allowing ample opportunities for rearranging ready warps. Only 5 out of 18 benchmarks have fewer than ten active warps on average.

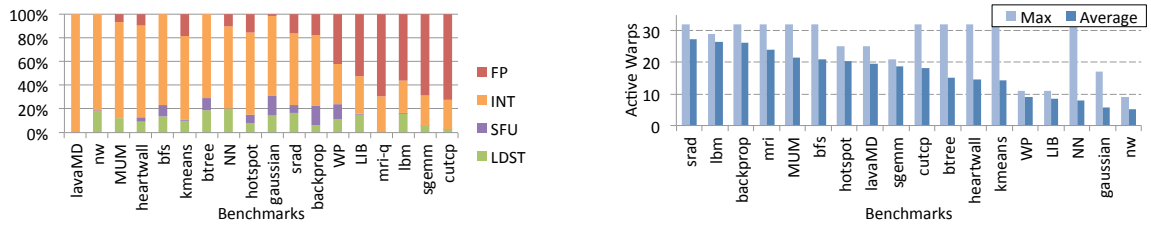
4.1 GATES Implementation Details

In this section we describe the microarchitectural support needed for implementing GATES. We extend the default Two-level warp scheduler with two enhancements: (1) per instruction type active warps subset, and (2) a dynamic priority-based instruction issue scheme.

Per instruction type active warps subset: Since GATES prioritizes issuing instructions of a specific type, we propose to logically split the active warp set into four active warp subsets, namely integer (INT), floating point (FP), special function unit (SFU) and load/store (LDST) subsets. Each subset is associated with the corresponding execution resource. This partitioning of the active warp set can be done logically, rather than physically separating the set, by adding two bits per entry of the active warps set. The two bits in each entry specify the execution unit needed for executing the corresponding warp instruction. Since instructions entering the active warps set are already decoded, the decoder can simply set the two-bit execution unit type as part of the decoded instruction information.

Instruction issue priority: The instruction issue arbiter inside the warp issue logic is modified with a simple priority-based issuing algorithm which assigns each instruction type an issue priority. We ordered the instructions in our implementation as: INT/FP, LDST, SFU, FP/ INT. The ordering implies that either INT or FP is given the highest priority first. If INT is given the highest priority, then FP will be given the lowest priority and vice-versa. This ordering always separates integer and floating point instructions to the two ends of the priority. The ordering priority between LDST and SFU is not relevant to this work, but we gave LDST a higher priority over SFU assuming memory operations have longer memory access latency. Fermi's instruction scheduler is capable of issuing two instructions per cycle. The only time an integer and floating point instructions are issued in the same cycle is when there is just one of either INT or FP instruction in highest priority, no LDST or SFU instructions, and one (or more) of instructions of INT/FP that are not in the highest priority. By pushing INT and FP instructions to the two ends of scheduling priority, units with lowest priority will enjoy longer idle periods. Furthermore, warps that need lowest priority unit will accumulate until a priority switch at which time there will be many ready warps that need the same execution unit type.

Dynamic priority switching: Instead of using static instruction priority, the priority ordering is switched dynamically during workload execution. We initialize INT as the highest priority and FP as the lowest priority. During execution if the INT active warp subset is empty while the FP active warp subset is not empty, then the priority is switched



(a) Instruction mix

(b) Runtime active warps set size

Figure 5: GPGPU workload characteristics

between INT and FP. Similarly, if FP is the highest priority and the scheduler sees that the FP active warp subset is empty and the INT warp subset is not empty, then INT is given the highest priority and FP is given the lowest priority.

GATES creates new idle periods and also lengthens existing idle periods by coalescing the bubbles in the functional unit pipelines. Figure 3b shows the effect of using GATES on the idle period length distribution. With GATES, 59.0% of idle periods are wasted due to idle-detect window (down from 83.4% compared to the basic Two-level warp scheduler). A larger portion of idle periods were moved into the power gating safe region, 18.9% (up from 6.5%) of idle periods are now longer than *idle-detect+BET*. While GATES was successful in creating positive power gating opportunities, unfortunately some of the increased idle periods moved into the negative energy savings region (centre region), moving up to 22.1% (from 10.1%). Recall that in this region the functional unit is power gated after idle-detect window but is woken up before the break-even time has passed. One potential solution to this issue is to naively increase the idle-detect window, thus lowering the possibility of the net energy loss scenario, but this would also result in more wasted idle periods. Clearly, there is a need to better address the negative energy savings region, which is the focus of next section.

5. BLACKOUT POWER GATING

In this section we propose a modified power gating scheme called *Blackout*. When a unit is power gated, it is placed into a *blackout state*, where the unit cannot be woken up until it has been power gated for at least the break-even time, even if there are ready instructions. Blackout completely eliminates the net energy loss occurrences. Figure 3c shows the combined effect of GATES with Blackout power gating on the idle cycle distribution. Blackout power gating essentially pushes all idle cycles within the middle region into the rightmost region by forcing idleness of execution units. In the case illustrated in the figure, 45.7% of idle cycles now result in net energy savings, a 7x increase compared to conventional power gating.

By forcing execution units to be power gated for break-even time, even when there are ready instructions, conventional wisdom tells us that it will most likely lead to performance penalties. But due to the unique execution environment of GPGPUs, Blackout does not suffer from performance penalties as feared. The primary reason is that GPGPUs have a variety of heterogeneous execution resources (INT, LDST, FP, and SFUs) coupled with a good mix of available instructions that are ready to be issued. When one execution resource type is forced idle, work can still be completed by the other execution resource types. As a result, the performance penalty due to forcing idleness on execution resources is hidden, leading to minimal performance impact.

Furthermore, the trends in GPGPU design shows that even a single execution resource type is going to be split into multiple clusters. For instance, in Fermi architecture there are two clusters of INT and FP units organized into two SPs as shown in Figure 1a. The more recent Kepler architecture uses six clusters of INT and FP organized as six SPs [4]. Similarly, AMD's GCN architecture currently has four clusters of SP-like SIMD pipelines in each SM-like core [3]. Considering these developments, we will propose an enhanced Blackout mechanism that specifically takes advantage of the clustered GPGPU architectures to further reduce the performance losses due to Blackout.

In this section we explore two policies, *Naive Blackout* and *Coordinated Blackout*. Both policies are implemented on top of GATES, which was discussed in the previous section.

Naive Blackout: In this policy once the unit is idle for at least idle-detect number of cycles, the unit is placed in blackout mode. The wakeup mechanism differs from conventional gating scheme. Compared to the conventional power gating state machine shown in figure 2c, Naive Blackout will not have a state transition from the uncompensated state to the wakeup state. The only transition to the wakeup state takes place from the compensated state. Once a unit enters the blackout state, the scheduler will simply avoid issuing instructions to the execution unit until after the break-even time is over. Once the break-even time is over, the scheduler is allowed to issue a ready instruction, if any, to the gated unit and trigger a wake up for the gated unit.

Coordinated Blackout: As mentioned earlier, clustered integer and floating point units are now common in GPGPUs. Coordinated Blackout takes advantage of the clustered architecture. In the description below we assume the baseline architecture has 32 integer and floating point units that are clustered into two groups of 16 units each. When both clusters of a given type (integer or floating point) are in an active state, Coordinated Blackout simply uses idle-detect window to detect idle cycles. If the idle cycles of a given cluster exceed the idle-detect window, then that cluster is placed in power gating mode and enters blackout state. Once a single cluster enters blackout mode, the second cluster will *not* use idle-detect cycles any longer in determining its power gating state. The second cluster will instead check the number of active warps waiting in the active warps subset associated with that execution resource. If no warp is waiting in the active warp subset, then the second cluster enters blackout state immediately, even if its idle cycle window is less than the idle-detect window. On the other hand, if a single warp is waiting in the active warp subset then the second cluster will not power gate even if the idle period

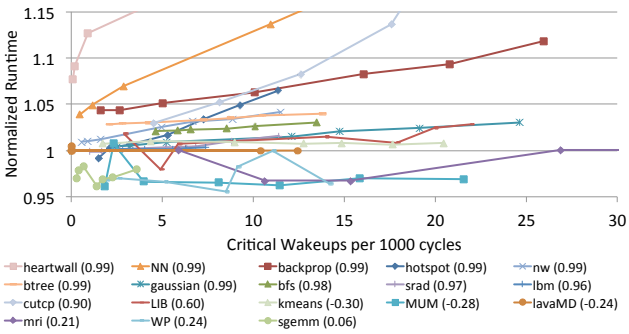


Figure 6: Critical wakeups correlation

length exceeds the idle-detect window.

The Coordinated Blackout mechanism makes the assumption that if there is a warp waiting in the active warp subset, then it is likely to become ready relatively soon. Since one of the clusters of that execution resource type has already entered the blackout state, it is best to avoid putting the second cluster in blackout state to avoid the performance penalty associated with waking up the unit. Hence, this approach improves performance by avoiding excessive wakeups associated with power gating, and saves power by avoiding the power gating overhead. Therefore, at least one of the two clusters will be always ON whenever there is a warp in the associated active warp subset. With Coordinated Blackout, GATES instruction priority switching policy is also extended to switch instruction priority type if both execution units of the highest priority type are in blackout.

5.1 Reducing Worst Case Blackout Impact with Adaptive Idle Detect

Till now, all of our proposed approaches use statically fixed idle-detect window size. Hence, once an idle-detect window is selected it is not changed at runtime. Further improvements to Blackout can be achieved by allowing the idle-detect window to be dynamically changed based on perceived performance loss at runtime due to Blackout. Hence, Blackout is augmented with an Adaptive idle detect mechanism that dynamically adjusts the idle-detect window to match an application's runtime behavior. This mechanism will rely on simple metrics to infer performance loss.

Inferring performance loss: It is not possible to precisely track the performance loss due to Blackout. However, it is possible to use secondary metrics as a proxy for inferring potential performance loss due to Blackout. While we explored a number of metrics, for simplicity, we describe one simple metric that we used in our evaluation. We use a metric called *critical wakeups* to measure performance loss. Critical wakeup is defined as a wakeup that occurs the moment the blackout period ends. This metric implies that there was at least one instruction that was blocked in the active warps subset waiting for its corresponding unit to finish its break-even time before wakeup. The reason why this metric is only a proxy for performance loss is that we do not know how long the instruction has been waiting (it could be have just entered the ready state, or a few cycles ago in the middle of the blackout period). Furthermore, not every blocked instruction leads to a performance loss because instruction execution delay does not always fall in the critical path latency.

Figure 6 shows the correlation of critical wakeups per 1000 cycles and performance loss for each benchmark across a range of static idle-detect values (0-10). The Pearson correlation coefficient (r) is displayed next to each benchmarks name. As can be seen, 11 benchmarks have strong correlation ($r > 0.9$) between critical wakeups and performance loss, showing great confidence that by regulating critical wakeups, we can limit performance loss. Some benchmarks (`kmeans`, `MUM`, `lavaMD`, `mri`, `WP`, and `sgemm`) have relatively low correlation. The reason for this low correlation is that these benchmarks do not suffer from any performance loss due to Blackout to begin with and hence changing idle-detect window is neither beneficial nor harmful.

Algorithm: Adaptive idle detect breaks execution time into epochs (in our case, 1000 cycles). During each epoch, a counter keeps track of the number of critical wakeups that occur. At the end of the epoch, if the number of critical wakeups is greater than a defined threshold, then the idle-detect time will be incremented by one. We explored various threshold values and empirically determined that a value of five gives the best balance between performance and energy savings. During a 1000 cycle window, if there are more than five critical wakeups for a given unit type, then its idle-detect time will be incremented by one. By increasing the idle-detect, we will power gate more conservatively, therefore, decreasing the number of critical wakeups.

Idle-detect is decremented conservatively every four epochs only if we do not go over the target critical wakeup. Our approach quickly increases idle-detect (react quickly to performance critical phases) and slowly decreases idle-detect. To prevent run away idle-detect values we bound the value to be between 5-10 cycles. We also explored unbounded idle-detect values and found that bounded idle-detect yields better tradeoff between performance and energy savings.

Since each application has its own instruction mix and scheduling order, in Adaptive idle detect there will be different idle-detect values for INT and FP and each of these values may change over time. As a result, each unit will automatically reach the appropriate idle-detect value that gives the best combination of power savings and performance.

6. ARCHITECTURAL SUPPORT

Figure 7 shows the architectural support for GATES, Coordinated Blackout and Adaptive idle detect mechanisms. The additional hardware support needed is shaded in different colors in the figure for each of the three proposed enhancements.

GATES: The base machine architecture is shown in Figure 1a. Each entry in the active warps set has a ready bit which is set whenever all the input operands are ready for that warp. The ready bit is used by the baseline Two-level warp scheduler to issue an instruction to the execution units. GATES requires each active warp entry to have two additional bits indicating the instruction type of the decoded instruction. The two-bit instruction type is set by the instruction decoder. Existing scheduling mechanisms already need the resource requirement information of each instruction for resource reservation purposes. Hence this two-bit instruction type may already be present in decoded instruction bits, in which case, GATES can make use of the existing information.

GATES modifies the instruction issue arbiter to determine which instruction type has the highest priority for schedul-

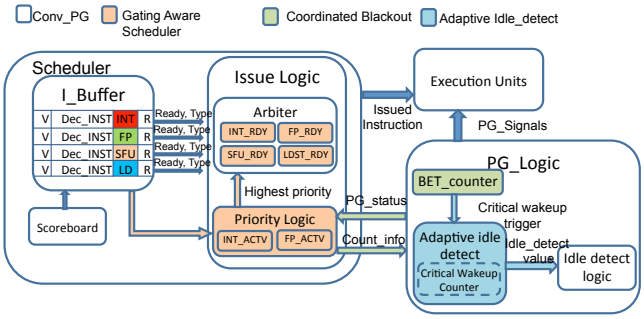


Figure 7: Architectural support for GATES, Blackout, and Adaptive idle detect

ing. In order to dynamically determine instruction priority, the active warp set is enhanced with two additional counters: INT_ACTV and FP_ACTV. These counters are incremented every time the corresponding instruction type enters the active warp subset and decremented whenever an instruction leaves the active warp subset. The instruction priority logic will make use of these counters to dynamically determine the higher instruction priority to pass to the instruction arbiter. For instance, if INT_ACTV is zero and FP_ACTV is non-zero then the scheduler switches the highest priority to FP, and vice-versa.

The current scheduling priority is stored as a two-bit value indicating the highest priority instruction type (either an INT or FP). Note LDST and SFU have fixed priority and hence when the highest priority instruction type between INT or FP is known then the other instruction type will become the lowest priority. Hence the instruction arbiter uses this two-bit value to determine the total priority ordering. Based on the current priority, GATES identifies N instructions to fit the issue width of N. In the Fermi architecture the value of N is two.

To quickly find the N ready instruction we added four counters to the issue logic. These counters count the number of *ready* instructions of each instruction type that are present in each active warp set. As mentioned earlier, when an instruction enters the active warp set, it is not necessarily ready for execution. It may be waiting for short latency input dependencies to be satisfied from a previously issued instruction. It is the job of the scoreboard to identify when the input operands of an active warp entry are ready, at which point it sets the ready bit. Whenever the ready bit is set for a given warp, the corresponding instruction type counter is also incremented. For the Fermi architecture used in our experiments, each counter is five bits wide, since at most 32 active warps are present in the active warp set. Thus, there are four 5-bit counters (shown as INT_RDY, FP_RDY, LDST_RDY, SFU_RDY in Figure 7).

The priority scheduler looks at the current priority and these counters to see what instruction types must be scheduled next. For instance, if the highest priority is INT then it looks at INT_RDY to see if there are at least two INT instructions ready in the active warp set. If so, it will scan the active warp set to identify the two INT instructions for scheduling. Note that even in the base machine the scheduler has to scan the active warp set to identify ready instructions for scheduling. Hence, our enhanced scheduler only needs to scan and match the two bit instruction type information with the current instruction priority. Thus in a single scan the two instructions that will be issued can be identified, just as in the base machine without adding any additional scans on the active warp set. Once the instruction types are issued, the corresponding ready counters are decremented appropriately.

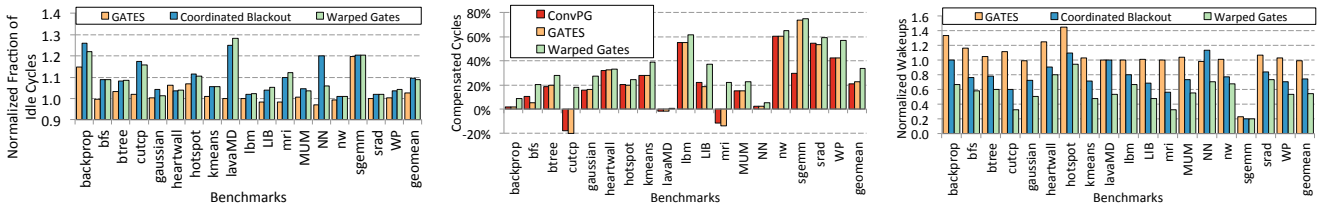
Similarly, if the highest priority is INT but INT_RDY shows only one ready warp, then the second issue slot will be filled with either LDST, SFU or FP instruction, in that order.

Conventional power gating: We assume that the conventional power gating technique is implemented as shown in [13]. This approach uses idle-detect logic and the ready-instruction detect logic. The idle-detect logic can be implemented as a counter that will be incremented every time an idle cycle is detected and cleared whenever a ready instruction is detected. Whenever the counter hits the idle-detect threshold, the power gating logic will trigger the power gating signal for that specific unit.

Blackout power gating: To support Blackout power gating, each of the power gated units is associated with an N-bit count down counter, called the blackout counter. Recall that all 16 integer units within a cluster are operated by a single power gating switch. Hence, for the design shown in Figure 7 we need four N-bit counters per SM, one counter per each cluster (two integer and two floating point clusters). The size of the counter must accommodate the break-even time for a given power gating design. The counter will be loaded with the break-even time as soon as the unit is power gated. Since most execution units need less than 24 cycles for break-even time in our current implementation, we need a 5-bit counter to store the break-even time whenever a cluster is power gated. As long as the value of the counter is not zero the unit will remain power gated and GATES will not assign an instruction to that unit.

Coordinated Blackout: Coordinated Blackout requires the knowledge of whether one of the two clusters is already in blackout state. In which case, the second cluster will not enter blackout as long as there is at least one instruction waiting in the active warp subset. To achieve this, the INT_ACTV or FP_ACTV counters available in the priority logic are used. Note that RDY instruction counters can not be used since an instruction may be in active warp set but not yet ready for execution. When one of the two clusters of a given type is power gated, then the Coordinated Blackout logic will check INT_ACTV or FP_ACTV counters to see if at least one instruction of the given type is in the active warp subset. If so, then the idle-detect mechanism is disabled to prevent the second cluster from entering the power gating state. On the same note, if there are zero instructions waiting in the active warp subset then the second cluster is immediately put to blackout state, even if the idle period is less than the idle-detect threshold.

Adaptive idle detect: Adaptive idle detect technique keeps track of the number of critical wakeups during each epoch using a critical wakeup counter. The counter will be incremented every time an execution unit gets a signal to wake up at the same cycle the break-even time counter hits zero. At the end of the epoch, the value of the critical wakeup counter will be compared to a pre-defined threshold, which was empirically set to five as described earlier . If the counter value is greater than the threshold then the idle-detect value will be incremented and will be loaded into the idle-detect register used in baseline power gating technique. Note that in the baseline the idle-detect value was a



(a) Idle cycles (b) Compensated cycles (c) Normalized wakeups (PG overhead)

Figure 8: Increasing power gating opportunity for integer units. Floating point units exhibit similar trends.

fixed value that could be hard-coded into the logic. However, for Adaptive idle detect we need a register that can be incremented or decremented in an epoch.

7. EVALUATION

7.1 Methodology

We evaluated our proposed techniques for performance and energy saving using GPGPU-Sim v3.02 [6]. We used the default Nvidia GTX480-like configuration provided with GPGPU-sim. The baseline architecture, with a core clock of 700MHz, contains 15 SMs with two SP units, four SFUs, and 16 LDST unit per SM. Each SP unit contains 16 double-frequency CUDA cores, each with individual integer and floating point pipelines (total of 32 CUDA cores per SM). The default warp scheduler is the Two-level scheduler with 48 warps per SM and capable of issuing two warps per cycle per SM. GPUWattch [16] and McPAT [17] are used for power estimations. We selected eighteen benchmarks to cover a wide range of scientific and computation domains from several benchmark suites including Rodinia [8], Parboil [2], and ISPASS [6]. For all the power gating results presented in this section, unless specified otherwise, we assume a default idle-detect window of 5 cycles and a break-even time of 14 cycles.

7.2 Increasing power gating opportunities

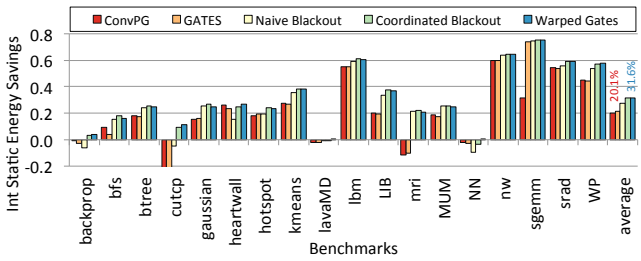
The following naming convention describes the techniques evaluated and applies to all figures: *ConvPG* refers to conventional power gating with Two-level scheduler (section 2.1); *GATES* refers to the GATES scheduler + conventional power gating (section 4); *Naive Blackout* and *Coordinated Blackout* (section 5) refers to GATES + Naive Blackout and GATES + Coordinated Blackout, respectively. Finally, we collectively refer to the combination of all of our proposed techniques as *Warped Gates*. Warp gates refers to GATES + Coordinated Blackout + Adaptive idle detect (section 5.1).

Extracting idle cycles through GATES and Blackout: Figure 8a shows the effectiveness of our proposed techniques at extracting idle cycles. The y-axis shows the fraction of idle cycles (idle cycles/execution cycles) normalized to the fraction of idle cycles extracted by the Two-level warp scheduler. The numbers reported here are for the integer unit, but the improvements are similar for the floating point unit. GATES alone was able to extract 3% more idle cycles than the baseline Two-level warp scheduler. These extra idle cycles represent the idle cycles that were extracted by coalescing pipeline bubbles. Coordinated Blackout increases the normalized fraction of idle cycles by 10%. Since this is a normalized fraction, the increase is not just due to increase in idle cycles but it is also due to reduced execution time, which leads to increased fraction of idle cycles.

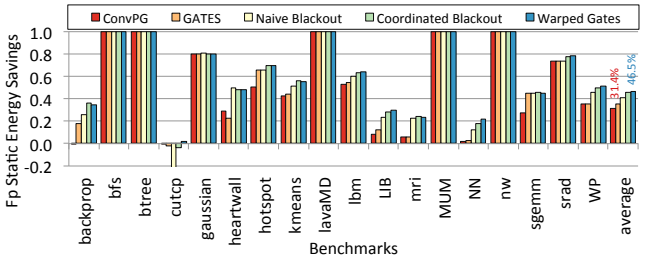
In most cases warped gates achieves slightly reduced fraction of idle cycles compared to Coordinated Blackout. This result is due to the fact that warped gates reduces execution time penalty of power gating by activating fewer power gating events. But the reduction in execution time is outpaced by the reduction in idle cycles. In other words, with warped gates idle cycles were reduced faster than execution time. This result is not a surprise since in many applications reducing the idle cycles does not correspondingly reduce the execution cycles, since not all idle cycle reductions are in the critical path of a program execution. For the same reason, in `heartwall` the fraction of idle cycles decreased with Coordinated Blackout compared to GATES. In this benchmark, with Coordinated Blackout the execution time decrease was outpaced by idle cycle decrease.

It is also interesting to note that GATES and Blackout provide savings under different operating regimes. When there are plenty of instructions in the active warp set, GATES allows instruction reordering to effectively improve idle cycle windows. However, when there are only a few active warp entries then GATES is unable to find opportunities for instructions reordering. In this case, Blackout allows instruction build-up whenever a functional unit is power gated. When the power gated unit comes out of blackout, then there are more opportunities for GATES to reorder instructions again. Hence, they both complement each other in effectively increasing the idle cycle windows.

Increasing time in compensated state. Figure 8b shows the percentage of cycles that the execution unit stays in the compensated power gating state for the integer unit for various techniques. Bars with negative values represents workloads where the execution unit was in an uncompensated state more than a compensated state. This figure demonstrates the effectiveness of our technique at extending the idle period length. Some benchmarks (`backprop`, `lavaMD`) tend to have highly utilized functional units and thus have very few idle cycles. Hence these benchmarks do not need any static energy saving techniques. There are a few benchmarks, such as `cutcp` and `mri`, that spends a significant amount of time in an uncompensated state with either conventional power gating or GATES. In these benchmarks many wakeups occur before break-even time. However, Warped Gates was able to significantly increase time spent in a compensated state that GATES alone cannot achieve. Warped Gates is able to achieve significant increase in the percentage of time units are in compensated state. The geometric mean of cycles in compensated state is 20.9% for conventional power gating, 22.6% for GATES,



(a) Int unit



(b) FP unit

Figure 9: Static energy impact of proposed techniques.

and 33.5% with Warped Gates.

Power Gating Wakeup and Overhead. Figure 8c shows the number of wakeups generated by each power gating technique normalized to the conventional power gating scheme. The number of wakeups can also be interpreted as the number of idle windows that are power gated. Power gating overhead is directly correlated to the number of wakeups. In general, if we reduce the number of wakeups, we reduce the power gating overhead. As expected GATES alone increases the number of wakeups in some cases. This result shows that GATES in some cases increases the idle cycle window to be just beyond the idle-detect window and hence triggers gating more often which leads to more wakeups. Coordinated Blackout, by design, decreases the number of wakeups by 26% compared to conventional power gating. Finally, warped gates further brings down the number of wakeups by 46% compared to conventional power gating by dynamically changing the idle-detect window to avoid excessive power gating overheads. Thus, our proposed techniques can essentially *reduce power gating overhead in half.*

7.3 Energy Impact

Figure 9a and Figure 9b show the static energy savings by taking into account power gating overhead for the integer and floating point units. The results are normalized to a baseline with no power gating. All floating point results reported in this section excludes integer-only benchmarks which have no floating point activity. Conventional power gating with the Two-level warp scheduler saves 20.1% and 31.4% of static power for integer and floating point units, respectively. Benchmarks such as `backprop`, `cutcp`, `lavaMD`, and `NN` experience negative or no energy savings with conventional power gating since the gating overhead exceeds the static energy savings.

GATES alone with conventional power gating saves 21.5% and 35.2% of static power for integer and floating point units, respectively. Hence, GATES alone creates some additional opportunities to save static energy. In some benchmarks GATES pushes some idle cycles windows to go past idle-detect window but not sufficiently far to overpass the break-even time. In these cases, GATES triggers more power gating events which result in uncompensated power gating. In fact, we already showed this result in figure 8b.

Naive Blackout further increases static energy savings to 27.8% and 41.1% for integer and floating point, respectively. There are just three cases where Naive Blackout leads to lower energy savings (`backprop`, `heartwall` and `NN`). Naive Blackout can potentially power gate too aggressively, leading to higher power gating overhead in these three benchmarks. Coordinated Blackout power gates the second cluster more conservatively than Naive Blackout. Hence, Coordinated Blackout increases static energy savings to 31.5% and 45.6% for integer and floating point units. Warped Gates saves 31.6% and 46.5% for integer and floating point units. Warped Gates saves ~1.5x more static power than conventional power gating across both integer and floating point units.

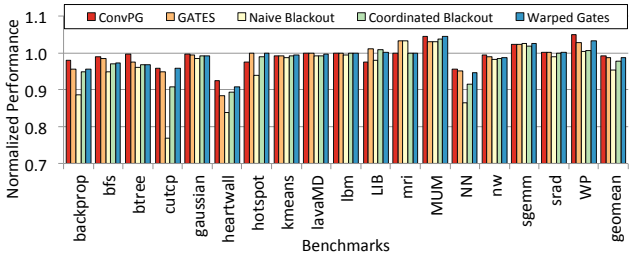


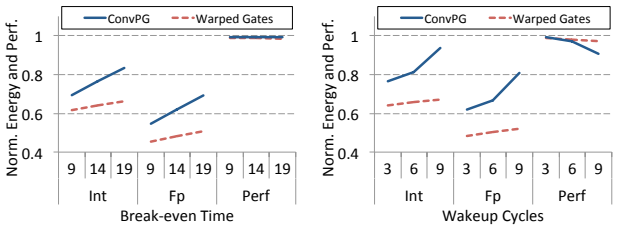
Figure 10: Performance impact

Note that our approaches do not increase the dynamic energy of functional units. The amount of work done (total number of accesses for each functional unit type) is constant per workload, irrespective of power gating. We also accurately modeled the few microarchitectural counters that were added to our design using RTL-level design and synthesis. We accurately measured their dynamic power due to counting activity. Our results show that these counters add less than 0.1% dynamic energy.

To estimate total on-chip energy savings we first estimate the fraction of leakage power that is consumed by the execution units. From GPUWattch, the total on-chip leakage power of GTX480 accounts for 26.87W. The integer units and floating point units account for 0.00557W and 4.40W, respectively. Using these values we estimate that execution units account for 16.38% of on-chip leakage power. Assuming leakage power accounts for 33% of total on-chip power and our technique can save 30% - 45% of static power, we estimate that our technique can save 1.62% - 2.43% of total on-chip power. As technology scaling continues, it is expected that static power will account for an increasing fraction of total on-chip power. If we assume leakage power accounts for 50% of total on-chip power, then our techniques can save 2.46% - 3.69% of total on-chip power.

7.4 Performance Impact

Figure 10 shows the performance impact due to power gating. Conventional power gating and GATES result in similar performance overheads of 1%. Naive Blackout suffers the worst performance overhead of 5% due to its aggressive shutting down of units for break-even time without



(a) BET (b) Wakeup

Figure 11: Sensitivity to BET and Wakeup delay

considering active warps that may be ready soon. Coordinated Blackout alleviates this concern and has 2% performance overhead. By taking into consideration soon-to-be ready active warps it can avoid aggressive power gating events. In spite of this effort, Coordinated Blackout suffers performance losses in certain benchmarks, such as `cutup`, `heartwall` and `NN`. The primary reason is that Coordinated Blackout places both the SP0 and SP1 clusters of INT or FP in blackout state after confirming no instructions are present in the active warps set. Unfortunately, as soon a unit is placed in blackout state in both clusters a ready instruction immediately enters the active warps set. While these are rare cases, warped gates changes the idle-detect window length to avoid even the corner case performance losses. Hence, warped gates achieves virtually the same performance overhead as conventional power gating, but with significantly more energy savings.

7.5 Hardware overhead

We implemented the various counters added to the base machine to enable the proposed techniques in verilog. We synthesized them using NCSU PDK 45nm library [1]. Also we extracted the area of the SM from GPUWattch [16]. An SM occupies 48.1 mm^2. The set of counters occupies 1,210.8 um^2, resulting in an 0.003% area overhead. An SM uses 1.92 W of dynamic power and 1.61 W of leakage power. The counters uses 1.55e-3 W of dynamic power and 1.21e-5 W of leakage power total, accounting for 0.08% dynamic and 0.0007% leakage power overhead.

7.6 Sensitivity to power gating parameters

We conducted a sensitivity analysis to various wakeup delays and break-even time values. These results are shown in figure 11.

Regardless of break-even times, warped gates always outperforms conventional power gating with the Two-level warp scheduler. With smaller break-even times, the energy savings gap between warped gates and conventional power gating narrows since the occurrence of negative power gating events decrease. As break-even time increases, the energy savings widens between warped gates and conventional power gating. For example, at a break-even time of 19 cycles, conventional power gating saves only 17% of integer static power, while warped gates saves 33%, a nearly 2x increase. Performance remains relatively constant with varying break-even times.

For higher wakeup delay values, the performance and energy savings for conventional power gating degrades significantly. Recall that conventional power gating results in high number of wakeups due to aggressive power gating of execution units. This results in high performance penalty due to paying more wakeup delays every time, and less power savings as the unit is consuming power when waking up, but not doing any useful work. With a wakeup delay of nine cycles, conventional power gating saves only 6% and 10% of integer and floating point static energy. Warped gates is able to sustain 33% and 48% of integer and floating point static energy savings. Performance-wise, conventional power gating has a nearly 10% performance impact, while warped gates suffers 3% overhead with a nine cycle delay.

8. RELATED WORK

GPGPU schedulers: GPGPU scheduler has been the focus of various optimizations. The Two-level warp scheduler [12] is an optimization over prior schedulers that placed all warps, both pending and active warps, in a single queue. Narasiman [19] also proposed another two-level scheduler that improves performance by dividing warps into fetch groups and rotating fetch groups whenever a long latency event occurs. Rogers [20] proposed a warp scheduler to improve cache locality. Jog [15] proposed a warp scheduler to enable efficient prefetching policies. GATES targets static power savings as the primary optimization criteria. We used the two-level scheduler [12] as our baseline scheduler in all our evaluations and built GATES on top of it.

Power aware schedulers: Power aware schedulers for CPUs and multicore systems have been studied extensively [7, 21]. Previous work focused on dynamic power aware scheduling and DVFS decisions based on available task and service time. GATES takes advantage of the unique GPGPU execution environment to rearrange warps for reducing static energy by increasing idle cycle length.

Power gating: Power gating techniques have been widely applied in microprocessors [13, 18], caches [11], and NOCs [9]. Prior work on GPGPU power gating focused on the granularity of SM cores [22], which works well when an entire SM is idle. But this work shows that there are plenty of opportunities to power gate execution units within an SM, even when an SM is not idle. In our work, we extended the power gating techniques to GPGPU execution units and proposed an enhanced blackout power gating scheme which takes advantage of the GPGPU execution model.

GPGPU power saving: Power efficiency in GPGPUs has been extensively studied. Several works [5, 12, 23] proposed techniques to save power of the GPGPUs register file using circuit level and microarchitectural techniques. Leng [16] explored clock gating and DVFS to save dynamic power of the execution units and register file based on the mask activity and execution phases, but static power was not considered in Leng's work, which is the focus of this paper.

9. CONCLUSION

In this paper we first analyzed the effectiveness of conventional power gating techniques when applied to GPGPU execution units. We showed that the basic Two-level scheduler that frequently intersperses different instruction types leads to short idle periods for a given execution unit type. These short idle periods limit the effectiveness of conventional power gating. We proposed GATES to aggregate instruction issue such that clusters of instructions of the same type are given priority. GATES is effective in extracting and coalescing idle periods. We then evaluated a

new power gating scheme called Blackout to avoid the negative effects of power gating a unit that does not have sufficiently long idle periods, even after applying GATES. We then proposed an Adaptive idle detection approach that dynamically varies the size of the idle detect window before a power gating event is triggered. We call the combined approach of using GATES, Blackout and Adaptive idle detect as warped gates. With negligible area and performance overhead, warped gates saves ∼1.5x more static power than conventional power gating, achieving 31.6% and 46.5% of integer and floating point static energy savings overall.

10. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This work was supported by DARPA-PERFECT-HR0011-12-2-0020.

11. REFERENCES

[1] The freepdk process design kit. http://www.eda.ncsu.edu/wiki/FreePDK.

[2] Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.

[3] Amd graphics cores next (gcn) architecture. Technical report, AMD, 06 2012.

[4] NvidiaâĂŹs next generation cuda compute architecture: Kepler tm gk110. Technical report, Nvidia, 2012.

[5] M. Abdel-Majeed and M. Annavaram. Warped register file: A power efficient register file for gpgpus. In *Proceedings of the 2013 International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, 2013.

[6] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009.

[7] D. Bautista, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato. A simple power-aware scheduling for multicore systems when running real-time applications. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.*, pages 1–7, 2008.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09.

[9] L. Chen and T. Pinkston. Nord: Node-router decoupling for effective power-gating of on-chip routers. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) 2012*, pages 270–281, 2012.

[10] S. Dropsho, V. Kursun, D. H. Albonesi, S. Dwarkadas, and E. G. Friedman. Managing static leakage energy in microprocessor functional units. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, 2002.

[11] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, 2002.

[12] M. Gebhart, D. Johnson, D. Tarjan, S. Keckler, W. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 235–246, 2011.

[13] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 32–37, 2004.

[14] H. Jeon and M. Annavaram. Warped-dmr: Light-weight error detection for gpgpu. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, 2012.

[15] A. Jog, O. Kayiran, A. Mishra, M. Kandemir, O. Mutlu, R. Iyer, and C. Das. Orchestrated scheduling and prefetching for gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[16] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi1. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[17] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[18] A. Lungu, P. Bose, A. Buyuktosunoglu, and D. J. Sorin. Dynamic power gating with quality guarantees. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '09, 2009.

[19] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, 2011.

[20] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, 2012.

[21] C. Scordino and G. Lipari. Using resource reservation techniques for power-aware scheduling. In *Proceedings of the 4th ACM international conference on Embedded software*, EMSOFT '04, pages 16–25, 2004.

[22] P.-H. Wang, C.-L. Yang, Y.-M. Chen, and Y.-J. Cheng. Power gating strategies on gpus. *ACM Trans. Archit. Code Optim.*

[23] W. Yu, R. Huang, S. Xu, S.-E. Wang, E. Kan, and G. Suh. Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading. In *38th Annual International Symposium on Computer Architecture*, pages 247–258, 2011.

[24] J. Zhao and Y. Xie. Optimizing bandwidth and power of graphics memory with hybrid memory technologies and adaptive data migration. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, 2012.