Origami: Folding Warps for Energy Efficient GPUs

Mohammad Abdel-Majeed
University Of Southern California
abdelmaj@usc.edu

Daniel Wong
University of California, Riverside
dwong@ece.ucr.edu

Justin Kuang*
Stanford University
kuang@alumni.stanford.edu

Murali Annavaram
University of Southern California
annavara@usc.edu

ABSTRACT

Graphical processing units (GPUs) are increasingly used to run a wide range of general purpose applications. Due to wide variation in application parallelism and inherent application level inefficiencies, GPUs experience significant idle periods. In this work, we first show that significant fine-grain pipeline bubbles exist regardless of warp scheduling policies or workloads. We propose to convert these bubbles into energy saving opportunities using Origami. Origami consists of two components: Warp Folding and the Origami scheduler. With Warp Folding, warps are split into two half-warps which are issued in succession. Warp Folding leaves half of the execution lanes idle, which is then exploited to improve energy efficiency through power gating. Origami scheduler is a new warp scheduler that is cognizant of the Warp Folding process and tries to further extend the sleep times of idle execution lanes. By combining the two techniques Origami can save 49% and 46% of the leakage energy in the integer and floating point pipelines, respectively. These savings are better than or at least on-par with Warped-Gates, a prior power gating technique that power gates the entire cluster of execution lanes. But Origami achieves these energy savings without relying on forcing idleness on execution lanes, which leads to performance losses, as has been proposed in Warped-Gates. Hence, Origami is able to achieve these energy savings with virtually no performance overhead.

CCS Concepts

•**Computer systems organization** → **Single instruction, multiple data;** •**Hardware** → **Power and energy;**

Keywords

GPUs, Leakage power, Power gating, SIMT lanes

*work was done when Justin was a student at USC

Copyright Abdel-Majeed, Wong, Annavaram 2016. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proceedings of the 2016 International Conference on Supercomputing

1. INTRODUCTION

Graphics processing units (GPUs) use the SIMT (single instruction multiple threads) execution model, that allows many of the execution lanes to share a single program counter to execute the same instruction but on different data elements concurrently. This approach simplifies control, thereby enabling high energy efficiency (performance/watt). GPUs have been extended 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, they suffer reduced efficiencies due to a wide variation in resource utilization and diverse parallelism demands [15, 19]. Resource underutilization reduces power efficiency, a growing concern reflected by the recent research activity in this area [5, 6, 15, 16, 28].

There exists two types of fine-grain idleness in GPU execution pipelines. Spatial idleness exists in the execution units in the form of inactive lanes caused typically by branch divergence. Temporal idleness exists in the form of pipeline bubbles caused by uncoalesced instruction scheduling and heterogenous resource usage. Many prior works have tackled the inefficiencies that cause spatial idleness through reducing branch divergence by reforming warps [14, 13], issuing multiple branch paths simultaneously [9, 29], or by embracing branch divergence for energy efficiency through power gating of individual lanes [34]. To tackle inefficiencies due to temporal idleness, prior work [6] proposed scheduling techniques to coalesce short idle periods to create longer idle cycles to efficiently power gate clusters of execution units.

Despite these enhancements to improve GPUs hardware efficiency on various fronts, there still exist significant fine-grain pipeline bubbles. In this work, we first show that these fine-grain pipeline bubbles exists regardless of workloads or scheduling policies. These fine-grain pipeline bubbles are a major source of wastage in execution lane energy. Existing power gating solutions such as [6, 34] require longer stretches of idleness to overcome the power gating overheads. In [6] the authors were able to achieve 80% of the leakage power reduction opportunity that is possible for execution units. But the need for long idleness to enable power gating was achieved through one performance degrading technique called Blackout. Blackout forces an execution unit to stay idle until the energy saving from power gating at least matches the energy overhead due to power gating. Blackout forces idleness even if an execution resource is required for computation thereby degrading performance in some benchmarks, as quantified in their work. Hence, while [6] was able

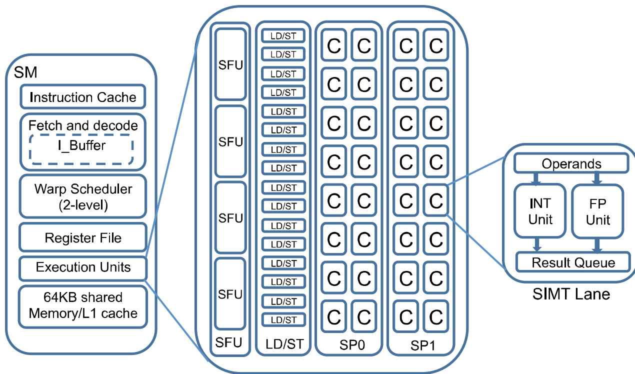


Figure 1: Baseline Nvidia GTX480 details

to achieve nearly the full potential for leakage energy savings it came with an average of 6.8% performance loss.

In order to improve the efficiency of execution units in the face of fine-grain pipeline bubbles, we propose *Warp Folding*. Warp Folding splits a warp into two half-warps, which are scheduled in succession, in order to fill these fine-grain pipeline bubbles. Warp Folding in turn creates idleness in half of the execution unit lanes, which can be leveraged for energy efficiency gains through power gating without the need for forcing additional idleness through techniques such as Blackout [6].

Warp Folding is then enhanced with folding-aware scheduler to minimize performance overhead. We present the *Origami Scheduler*, which schedules warps based on the instruction type and active mask pattern in order to issue warps with similar type and active masks to coalesce idleness across the time domain. Warp Folding and Origami scheduler together serve to maximize the fine-grain idleness opportunities through spatial-temporal manipulations of threads. We refer to these two techniques collectively as Origami.

The rest of the paper is organized as follows: Section 2 provides background and motivation for our proposed techniques. Section 3 discusses the proposed techniques and the required microarchitectural support. Section 4 presents the simulation methodology and results. We discuss the related work in section 5 and conclude in section 6

2. BACKGROUND AND MOTIVATION

2.1 GPU Architecture

In this paper we use NVIDIA Fermi-like architecture[3] as the baseline GPU model. The GPU consists of multiple streaming multiprocessors (SMs). Figure 1 shows the main pipeline stages within an SM. Each SM has tens of execution resources, which can be subdivided into special function units (SFUs), load/store units, INT and FP units. INT and FP units are clustered into groups called shader processors, named SP0 and SP1 in the figure. Each SM uses SIMT execution model [3] that allows many of the execution lanes to share a single program counter to execute the same instruction but on different data elements concurrently. The SIMT execution model is driven by warps, a group of 32 threads executing the same instruction with different input operands values. Each thread has its own execution unit and architectural register file which form a SIMT lane.

Each warp has up to 32 active threads. However, during program execution the number of active threads may be fewer than 32 due to branch and memory divergence. For example, during branch divergence, the threads within the warp can diverge into the taken and the not taken paths. As a result, the scheduler executes the threads on the taken path and the not taken path sequentially over different issue cycles. An inactive thread within a warp does not need to activate the register file and the designated SIMT lane resources assigned to that thread.

2.2 Static Energy's Growing Importance in GPUs

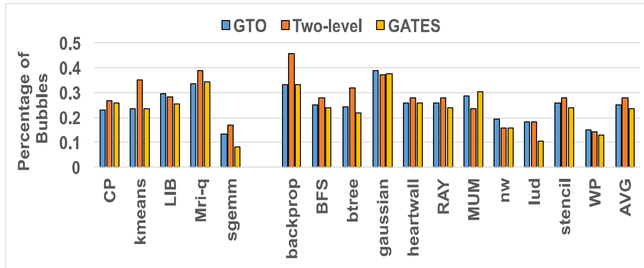
The power consumption breakdown of Nvidia GTX480 shows that the execution units consume 20.1% of the total GPU power [22], which surpasses the power consumption of memory and register file at 17.8% and 13.4%, respectively. Recent work [6] showed that execution units leakage power stands at almost 50% of the total GPU power. As technology scales it is expected that the leakage power contribution to the overall power continues to increase. A recent study [18] showed that even the introduction of FinFET transistor based designs are unable to stop the steady progression of the leakage power as a growing source of power waste. In addition, the newer generations of the GPU architecture like Kepler [4] has more SIMT lanes integrated inside the same SM. For example, Kepler [4] has 192 SIMT lanes in one SM while Fermi [3] has 32 SIMT lanes per SM. Hence, execution unit power is a growing fraction of the overall GPU power.

Power gating is a technique used to cut off the leakage current that flows through a circuit block [21]. Power gating can be enabled by integrating a single transistor at the header (between the Vdd and the circuit) or the footer (between the circuit block and the Gnd) of the circuit block. When the power gating transistor is ON the circuit block operates normally. The block is power gated by turning OFF the power gating transistor. A dynamic power overhead is associated with each power gating transition from ON to OFF and from OFF to ON. In order to compensate for the power gating overhead, a circuit block must be gated for at least a minimum number of cycles, called the break-even time [17]. To reduce negative power gating events in [17] the authors suggested that a circuit block must stay idle for a minimum number of cycles, called idle-detect time, before the gating is enabled. This suggestion is based on a simple heuristic that if a unit is idle for idle-detect cycles then it may be more likely to stay in idle state for break-even time. Note that if a service request for a gated block arrives before break-even time has elapsed then it results in a net power loss if the circuit block decides to service that request immediately.

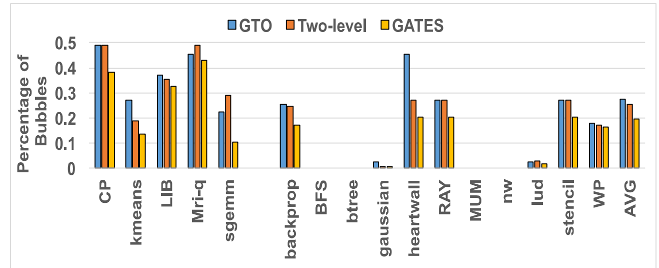
In this paper we focus on leakage energy saving for execution units, comprising of integer (INT) and floating point (FP) units. We excluded special function units (SFU) from this work because they exhibit long idle periods and conventional power gating [17] is already well suited to exploit the long idle periods.

2.3 Prevalence of Pipeline Bubbles

Now we demonstrate the pervasiveness of fine-grain pipeline bubbles that exists in GPGPUs regardless of workloads or warp schedulers. The pipeline bubbles exist because the scheduler was unable to issue two instructions back to back



(a) INT pipeline



(b) FP pipeline

Figure 2: Percentage of bubbles shorter than 10 cycles normalized to the total execution time

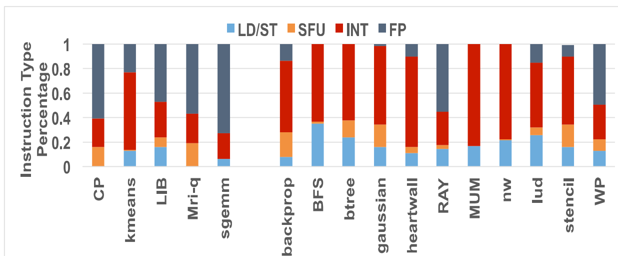


Figure 3: Workloads instruction type breakdown

to the same pipeline. Note that in designs evaluated in this work many of the execution units have an initiation interval of one cycle and hence there is no design imposed limit on using the execution units in consecutive cycles. But due to data hazards (such as read-after-write) and/or insufficient parallelism the pipeline bubbles are pervasive. These pipeline bubbles lead to leakage energy losses.

Figure 2 shows the percentage of idle cycles where the execution units (INT and FP units) have bubbles that last less than 10 cycles, normalized to the total execution time. Short idle cycles (less than 10 cycles) present a difficult challenge to traditional power gating techniques as they require a minimum breakeven time before the leakage energy savings can overcome the power gating overhead. The figure shows the results for three different warp schedulers, namely GTO, two-level warp scheduler [15] and gating aware two-level warp scheduler (GATES) scheduler [6]. Such schedulers are agnostic to resource usage, and therefore suffer many short pipeline bubbles universally across all benchmarks. On average, with the two-level scheduler, the execution unit experiences short pipeline bubbles 25% of the time for integer and floating point units, respectively.

As shown in figure 3, these workloads have a diverse mix of instructions. When these diverse mix of instructions are interspersed then pipeline bubbles occur in execution resources. In this context,GATES scheduler [6] was proposed to schedule instructions based on execution unit resource type in order to coalesce resource usage, leading to longer idle periods for any given unused resource. These longer idle periods are then utilized for power gating purposes at the SP level. But even with the GATES scheduler, we see that short pipeline bubbles are 23% and 19% for integer and floating point units, respectively. GATES targets eliminating intermediate length bubbles which are amenable to power gating, leaving very short bubbles intact. This prevalence of fine-grain pipeline bubbles, regardless of the warp scheduler and workloads, presents a challenge (and an opportunity) to improve GPU power efficiency.

3. ORIGAMI: CONVERTING PIPELINE BUBBLES INTO ENERGY SAVINGS OPPORTUNITY

In order to leverage these fine-grain pipeline bubbles and convert them into energy savings opportunities, we present *Origami*. Origami consists of two components: Warp Folding and the Origami scheduler. A simplified overview of Origami is presented in figure 4 to illustrate the various components. Figure 4a shows warps (in blue) that are executed in the integer and floating point pipelines, along with the active mask of each warp (the active mask is represented by the width of the blue line for illustrative purposes). The gaps seen between two warps are the pipeline bubbles where there are no available warps for execution in that warp scheduling window. This scenario, which is a typical state of execution units, shows that there are many fine-grain pipeline bubbles interspersed throughout the execution pipeline. The goal of Origami is to convert these wasted idle cycles into long stretches of idleness that can be harnessed for energy savings. The Origami scheduler uses a two step scheduling policy. First it schedules warps based on the instruction type. As shown in figure 4b, by scheduling based on instruction type, we can coalesce all the INT or FP instructions to appear close in time. Thus we can squeeze out course-grain idleness with type based coalescing which traditional power gating techniques can leverage for energy savings (show in green). Once the instruction type based scheduling is complete it still leaves plenty of short pipeline bubbles that are beyond the reach of traditional power gating. Furthermore, in the presence of divergence there is also lane level idleness that is dispersed. Origami then uses a second level scheduling that issues warps with similar active masks. As shown in figure 4c, scheduling by active mask can stretch lane-level idleness in order to extract more lane-level power gating opportunity. Finally, with Warp Folding, we can convert fine-grain pipeline bubbles into contiguous idle lanes to maximize energy savings as shown in figure 4d. In the rest of this section, we discuss each component of Origami.

3.1 Warp Folding

We will first discuss in detail how Warp Folding can convert fine-grain pipeline bubbles into energy savings opportunity. Then in the next section, we will discuss in detail

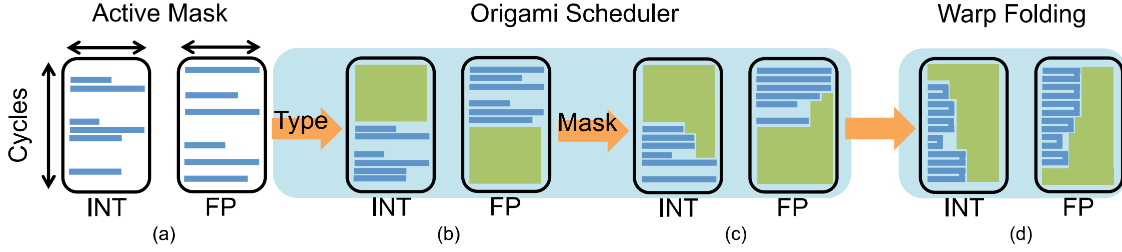
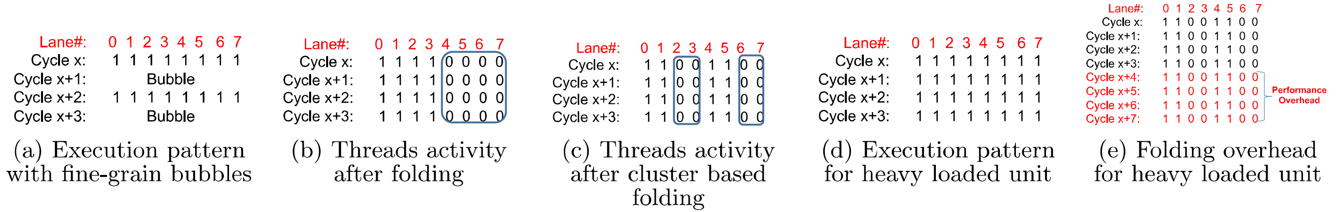


Figure 4: Origami consists of the Origami scheduler and Warp Folding.



(a) Execution pattern with fine-grain bubbles
(b) Threads activity after folding
(c) Threads activity after cluster based folding
(d) Execution pattern for heavy loaded unit
(e) Folding overhead for heavy loaded unit

Figure 5: Effect of Warp Folding on SIMT lanes activity

how the Origami Scheduler can extract course-grain idleness, leaving behind only fine-grain idleness that Warp Folding can utilize.

We will use figure 5 to illustrate how Warp Folding can be used to convert wasteful pipeline bubbles into useful idleness. Figure 5a illustrates the scenario where two instructions with a full active mask are issued to an INT pipeline but these instructions are issued two cycles apart (similar to the bottom of figure 4c). The intervening cycle between the two instructions is idle. This scenario occurs when the scheduler does not have back-to-back instructions to issue to the same SP unit. Such a scenario can occur due to instruction mix, resource availability (structural hazards) elsewhere in the microarchitecture, and stalls due to data hazards.

Warp Folding splits up the 32 threads in a warp into two half-warps, which are then reshuffled to use the same active lanes and then issued in succession. The two half-warps will be scheduled back-to-back to the same SP unit. Each half-warp will be using half of the SIMT lanes in the designated SP. The other half will be idle and will not be activated while the half-warps are scheduled.

Figure 5b shows an example of Warp Folding. Warp Folding transfers the bubbles in the lower order lanes to the higher order lanes of the designated execution pipeline. The transferred bubbles coalesce with the already existing bubbles in the higher order half to generate a longer sequence of bubbles that creates a higher potential for power gating (similar to the bottom of figure 4d). Because we are scheduling two half-warps back to back, the two half-warps must be issued with a delay equal to the initiation interval of the pipeline for that particular instruction.

In order to take advantage of Warp Folding, the threads within the second half-warp should be scheduled on the same SIMT lanes as the first half-warp used. As a result, half of the SIMT lanes are inactive when the half-warps are scheduled. Under unconstrained Warp Folding, the higher order half of the SIMT lanes' data will be directed to the lower order half of the SIMT lanes. This requires a large multiplexer with significant delay and a complexity. Hence, rather than using unconstrained Warp Folding across all 32 lanes, in this work we fold within a SIMT cluster of 4 lanes to minimize hardware overhead. This is demonstrated in figure 5c where warps are folded within each 4-lane cluster. As shown the threads within the same cluster are split into two sub-warps where each sub-warp has half of the threads in each cluster. This design is simpler because the threads assigned to a certain cluster are moving only between the lanes within the cluster. Hence the wiring overhead and the additional multiplexers delay are mitigated.

3.1.1 Warp Folding Policies

Warp Folding too naively may translate into a performance overhead. While the ideal situation for Warp Folding is to have a bubble between each scheduled warp, this may not always be the case. For example, figures 5d and 5e show the case where Warp Folding has a negative impact on performance. This scenario occurs when multiple instructions are issued back-to-back in consecutive cycles to the same pipeline. Hence, in order to avoid impacting performance, we have to take into account runtime execution resource utilization when deciding on when to fold warps. In this work we explore different Warp Folding policies. Warp Folding can fold *aggressively* for the *minority* instruction type and *conservatively* for the *majority* instruction type.

The Warp Folding process starts by counting the number of instructions of each instruction type (INT or FP). For this purpose we augment the scheduler to include instruction type (INT, FP, LDST, SFU) count information within each group. If the INT instructions count is higher than FP instruction count then warps with INT instructions (majority type) are *conservatively folded* and warps with FP instructions (minority type) are *aggressively folded* into two half-warps. The fundamental intuition is that when there are more instructions of a given type, that instruction type should be given more resources for execution, even at the expense of leaving some pipeline bubbles as is. On the other

hand, an instruction type with fewer instructions can be curtailed more aggressively to use fewer execution resources, and thus more pipeline bubbles are created for energy savings. Since we are applying aggressive and conservative folding techniques based on the number of instructions of a given type, the warp scheduler should also be made folding-aware as will be discussed later.

Aggressive folding targets folding instruction type (INT or FP) that has lower count. There are typically more pipeline bubbles, and more opportunities to fold warps. We empirically observed that folding more than 70% of the time can lead to performance loss as some of the dependent instructions have to wait for the folded warps to finish. Hence, we selected a 70% threshold. Therefore, in aggressive folding, we fold the selected instruction type for 70% cycles of each phase. In the remaining 30%, the warps are not folded.

Conservative folding targets folding the instruction type that has higher instruction count already. Since there are more instructions of this type in the pipeline already it is important to allow these instructions more resources to finish their execution even at the expense of some missed power gating opportunities. Using empirical measurement we selected a 40% threshold. Thus warps are folded for 40% cycles of each phase. In the remaining 60%, the warps are not folded.

Adaptive folding: In addition to the instruction count, we take the resources utilization into account to make sure that folding is not enabled when the workload is highly utilizing the resources. Figure 5e shows an example where it is not recommended to fold. So during the current phase we keep track of the total issued instructions. At the beginning of the next phase the total issued instructions count in the current phase is considered to decide if we should enable the aggressive and the conservative folding policies during the next phase or they should be disabled. Hence, the policy will adapt to the application behavior and make sure that Warp Folding does not result in performance loss. For example we decided to disable Warp Folding when the total number of issued instructions in the previous phase is 90% of the maximum possible issued instructions. In the results section we will discuss the folding ratios for each workload and how the adaptive folding can be used to avoid harmful scenarios.

Phase length: Folding one warp creates only a two-cycle bubble in the higher order lanes. In order to create longer idle windows in the higher order lanes, Warp Folding must be continuously activated for at least N_{Fold} cycles or greater to exploit power gating opportunities. Thus the decision to either fold or not is made at beginning of each phase.

If the value of folding period N_{Fold} is small, it may force the power gated units to wake-up before the break-even time is elapsed. On the other hand, if N_{Fold} is large we cannot issue warps for extended period to power gated lanes. Hence, the value of N_{Fold} is selected based on the power gating parameters. The minimum folding length that can translate the threads inactivity into savings is shown in Equation 1.

$$N_{Fold} = N_{pipeflush} + N_{idledetect} + N_{breakeventime} \quad (1)$$

Where $N_{\text{pipeflush}}$ is the number of pipeline stages in an execution unit. Recall that each execution unit, such as INT, FP and SFU, is pipelined and instructions have different latencies based on the opcode. Hence, once an instruction is issued to the execution unit the unit cannot be power

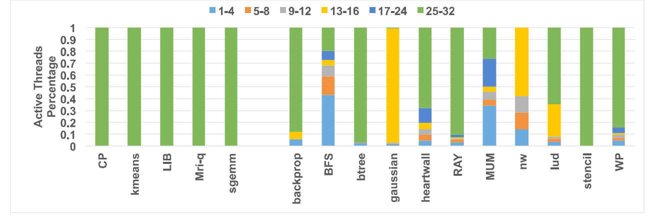


Figure 6: Threads activity breakdown

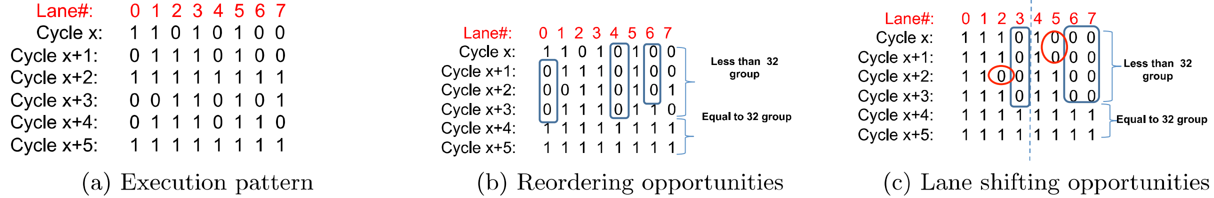
gated for $N_{\text{pipeflush}}$ cycles. Based on the default instruction latency numbers in GPGPUsim [7] configuration for fermi GTX480 architecture, $N_{\text{pipeflush}}$ is typically about five cycles for most instructions, and for a few instructions it can reach up to 19 cycles. Only divide instruction has longer than 19 cycles latency, but this instruction occurs rarely in most workloads we analyzed. Hence, in this work we conservatively chose 19 cycles for $N_{\text{pipeflush}}$. $N_{\text{idledetect}}$ is the number of idle cycles before power gating can be activated and its value is set to 5 in our experiments based on prior research that empirically showed that 5 cycles is a good trade off to reduce unnecessary power gating while still capturing most opportunities [17]. $N_{\text{breakeventime}}$ is the minimum number of power gating cycles needed to compensate the power gating overhead and it has been shown to be in the range of 9-19 cycles in [17]. Hence, the value of N_{Fold} should be at least 50 cycles. As discussed earlier, since we enable the conservative folding policy only for 40% of the time, the phase length N_{phase} should be at least 120 cycles(i.e. N_{phase} * 40% >50). We did sensitivity analysis on the phase length by changing N_{phase} value between 150-500 cycles. Our results show that the phase length of 250-400 cycles provides the optimal trade offs between the energy savings and the performance overhead. Longer phases reduce the proposed techniques adaptation to application behavior and results in a small degradation in performance. In our simulations we used N_{phase} of 300 cycles.

3.2 Origami Scheduler

While Warp Folding can convert fine-grain pipeline bubbles into energy saving opportunities, the Origami Scheduler can extract coarse-grain idleness. The Origami scheduler squeezes together idleness at the coarse grain level across execution units (through type-aware scheduling as shown in figure 4b) and lanes (through lane-activity-aware scheduling and lane shifting as shown in figure 4c). This way the scheduler can coalesce resource usage and avoid scheduling warps that uses higher order lanes. This approach leaves fine grain idleness that Warp Folding can efficiently extract as discussed in the previous subsection.

As a first step the Origami scheduler classifies warps based on instruction type, and prioritizes issuing instructions of a given type. As such it enables INT and FP execution resources, to be either utilized or idle for longer stretches of time, as has been demonstrated in [6].

In the second step the Origami scheduler uses lane aware scheduling. The need for lane aware scheduling exists because of inactive SIMT lanes due to branch divergence or insufficient parallelism. Figure 6 breaks down the number of active threads in each issued warp. As shown, the workloads on the left side of the figure do not exhibit lane level idleness. However, as shown on the right side of the figure,



(a) Execution pattern (b) Reordering opportunities (c) Lane shifting opportunities

Figure 7: Reordering and lane-shifting effects on power gating opportunities

other workloads exhibit lane level idleness due to divergence. For example, 40% of the issued warps in the heartwall workload do not use all the 32 SIMT lanes. Figure 7a shows an illustration of the threads activity of different SIMT lanes when underutilized warps are issued. When under utilized warps are issued arbitrarily the fine grain bubbles in each SIMT lane can be unaligned. To exploit lane level idleness more efficiently the Origami scheduler uses lane-aware scheduling to coalesce lane-level idleness. Origami scheduler divides the warps that are scheduled on the SP units into two groups. The first group has the warps with less than 32 active threads and we refer to this group as *less than 32 group*. The second group has the warps with 32 active threads and we refer to this group as *equal to 32 group*.

At the beginning of each phase the scheduler checks the number of warps in each group (=32 or <32) and select the group with the higher number of warps. This approach ensures that at the beginning of each phase the scheduler is likely to find instructions from the same group for the upcoming N consecutive cycles. Figure 7b shows the effect of the Origami scheduler grouping on the power gating opportunities. As shown, the Origami scheduler avoids mixing fully active warps with under utilized warps.

Lane Shifting: The last step in Origami schedule is lane shifting. The previous step simply avoids interspersing fully active warps with underutilized warps. Thus for a contiguous stretch some of the lanes in a warp are active, but they are not necessarily the same lanes. Lane shifting is the technique that aligns the inactive lanes to be the same by moving computation across SIMT lanes.

However, unrestricted lane shifting across all SIMT lanes incurs penalties in transferring register data across lanes. GPUs currently cluster the SIMT lanes into four lanes per cluster for design simplicity [15, 19]. Hence, we exploit this clustering and restrict lane shifting to be within the cluster boundaries. All the shift operations move active threads to the left most available lane. Moving the threads from their designated lanes to other lanes have been used before. The authors in [19] used this feature to *replicate* the thread execution on an empty lane to enable the intra-warp DMR technique proposed in that paper. However, in this work we move the threads from their designated lanes to an empty lane within the same cluster to improve the power gating opportunities. Limiting the shifting to the cluster boundaries may reduce some opportunities for power gating, but it still provides significant power savings with minimal hardware cost.

Figure 7c shows the effect of applying lane shifting on each cluster of 4 SIMT lanes. In the first cycle the active thread running on lane 3 is shifted to lane 2 in the first cluster. Also, the active thread running on lane 5 is shifted to lane 4 in the second cluster. By shifting all active threads to the left most inactive lane within each cluster we created a new four cycle idle window for lane 3 in the first cluster. In the second cluster we extended the three cycle idle window of lane 6 to four cycles. As shown in the illustration, lane shifting improves the power gating potential by doing the best effort to align the threads within each cluster. However, the restriction of shifting lanes within a cluster leads to some missed opportunities. For instance, if lane shifting was allowed across all 32 lanes, without any cluster restriction, then the two isolated inactive threads, marked as two red ovals in the figure, could have been aligned to create a three cycle idle window.

3.3 Optimizations

Avoiding Starvation: Separating warps into groups may lead to starvation if warps continue to enter the currently prioritized group for scheduling. In reality such scenarios don't occur in applications since eventually warps from the active group will be stalled or switch to be part of the inactive groups warps (i.e. due to change in the active mask activity and/or instruction type). Hence, the default scheduler eventually cannot issue from the current active group due to stall and data dependencies and it will be forced to switch to a different group of warps.. However, to bound starvation length, the scheduler is forced to switch between groups every M phases, where each phase runs for N_{phase} cycles. Thus after $M * N_{phase}$ cycles, a group switch is automatically initiated. This forced switch gives a chance for the warps in the other groups to be scheduled. As a trade off between performance impact and power savings opportunity we selected M to be 2. As a result, the scheduler is allowed to issue from the same group for two phases at most.

Avoiding unnecessary warp folding: Every time Warp Folding is enabled, the warp is divided into two half-warps scheduled back to back on the same lanes. However, benchmarks like gaussian, nw, heartwall and MUM have many scheduled warps with less than 16 active threads due to insufficient thread level parallelism. In this case the original warp need not to be folded. This simplification will disable the need to schedule the two sub-warps over two different issue cycles. The decision to check whether the entire upper or lower half of the active mask vector is idle can be done at the scheduler stage. The active mask is visible to the scheduler and the scheduler will be able to decide if a half-warp has all zeros when a warp is folded.

Eager power gating: The conventional power gating scheme [17] forces each unit to wait for an idle-detect period before switching to the uncompensated state. The reason to wait for idle-detect period is to observe sufficient idleness history before activating power gating. However, the pro-

posed Warp Folding approach guarantees that once a warp is folded, then the instruction type associated with that warp is continuously folded for the entire phase duration. Hence, rather than waiting for idle-detect window the power gating logic can immediately initiate gating of an unused lane. We call this approach *eager power gating*. In the *eager power gating*, whenever Warp Folding is enabled this information is conveyed from the scheduler to the execution unit's power gating logic. As soon as the first idle cycle is detected in the upper half of the SIMT lanes and Warp Folding enable signal is turned ON the power gating logic immediately gates the upper half of the SIMT lanes within each cluster.

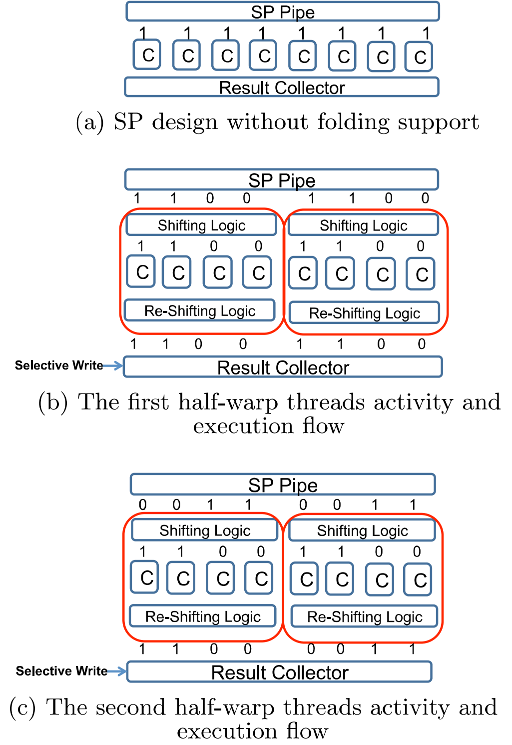
3.4 Architectural Support

In order to enable Warp Folding, the warp scheduler and folding logic should be modified to be able to decide when to fold and how to fold. INT-Count and FP-Count counters are used to hold the count of the warps with INT and FP instructions, respectively. The values of the INT-Count and the FP-Count are used to decide on which instruction type will be aggressively folded and which instruction type will be conservatively folded at the beginning of each phase. A two-bit instruction type field is added to the decoded instruction in the instruction buffer to annotate the instruction type.

In case of Warp Folding, half of the threads will be scheduled in the current cycle and the other half will be scheduled in the next cycle. For the first half, the threads execute on their designated SIMT lanes without any change. For the second half, the threads are shifted to the same SIMT lanes used by the previous threads. In order to make the hardware implementation simpler and avoid using a very large crossbar, each 4 SIMT lanes are clustered together [15] and a 4 to 1 shifting logic is assigned to each SIMT lane. To honor this clustering limitation, instead of folding the threads at the middle of the active mask as shown in figure 5b, the fold will be at the cluster level as shown in figure 5c.

In order to make sure that the operands that belong to the threads in the second half-warp will not be overwritten, the SP pipe register will be marked as busy till the second half-warp is scheduled. At the end of the execution of a folded warp, a re-shifting process should take place to return the output data to its original threads. Both the shifting and the re-shifting operations are controlled by the active mask of the scheduled threads. The only difference is that the re-shifting logic shifts the output to the right to return the threads to their original place.

Figure 8 shows the Warp Folding steps in the context of two clusters for simplicity. Figure 8a shows the basic architecture when a fully active warp (i.e. has an active mask of 11111111) has been scheduled. Figure 8b shows the modified pipe with the shifting and the re-shifting logic added per a 4 SIMT lanes cluster. The figure shows the active mask of the threads in the first half-warp before and after the shifting logic. As shown, the threads activity of the first half-warp is 1100 1100. Since they are already mapped to run on the lower order SIMT lanes the threads mapping will not be affected by the shifting logic. On the other hand the second half-warp threads activity will be 0011 0011 as shown in Figure 8c. During the second half-warp execution the shifting logic shifts the active threads to the lower order SIMT lanes. As a result, after the shifting logic the active mask will be 1100 1100 instead of 0011 0011. After the threads finish their execution they will be remapped to their original location by the re-shifting logic.



(a) SP design without folding support

(b) The first half-warp threads activity and execution flow

(c) The second half-warp threads activity and execution flow

Figure 8: Warp Folding execution steps and detailed threads activity

It is important to note that the proposed Warp Folding approach does not impact LD/ST instructions. Since warps are folded at the beginning of the EX stage and unfolded at the end of the EX stage and before the write-back stage Warp Folding does not have any impact on memory coalescing.

We implemented the lane shifting logic and the required multiplexers in RTL to estimate the overall delay of the additional logic. The results show that the lane shifting logic delay is significantly smaller than the typical execution stage delay. However, we pessimistically assume that the lane-shifting and re-shifting logic need an extra two pipeline stages, one for each operation. Figure 9 shows the updated pipeline after adding the two additional pipeline stages along with the additional hardware that is required by each stage. For simplicity, we show a block level design rather than the full circuit level detail as used in our RTL implementation. In the shifting stage we have the shifting logic. In the normal case the active mask is used to decide which lanes are active and which lanes are inactive(i.e. when the mask is 0 then the lane will be inactive and when the mask is 1 the lane will be active). However, when the shifting and the folding techniques are applied the lane shifting logic shifts the input values to different lanes. As a result the original active mask should be shifted accordingly to reflect the change in the lane activity. The logic at the bottom of the shifting stage is responsible for generating the new active mask. The new logic uses the *sub-warp# (sub-warp0 or sub-warp1)* and the *Split* signal to generate the new active mask. The new

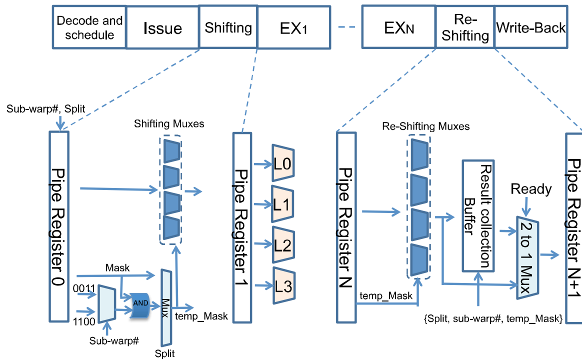


Figure 9: Modified GPU pipeline

active mask is ANDed with the original active mask(labeled Mask in the figure) to generate the correct threads activity labeled as *temp_Mask* in the figure. *temp_Mask* is carried with the threads through the pipeline and is used by the *re-shifting stage*.

The *re-shifting* stage has the re-shifting logic and an additional result collection buffer. The result collection buffer collects the output data from each sub-warp before sending the output data of the 32 threads on the the result bus at the same time. If Warp Folding is not enabled then the output data will bypass the collection buffer. All the results in the evaluation section are based on the execution model where two additional pipeline stages are added to our implementation compared to a baseline without the two additional stages.

Origami scheduler requires the warps to be classified into *less than 32 group* and *equal to 32 group*. To enable this classification, as soon as a warp is fetched into the instruction buffer we use an all-ones detector logic on the active mask that is already stored in the SIMT stack. We set a one-bit grouping field in the instruction buffer to enable this classification. The detector logic can be shared across all warps within an SM and it only needs access to the SIMT stack. The detection logic is also off the critical path since instructions spend multiple cycles in instruction buffer after fetch before they are scheduled for execution. In this work we assume the base machine can schedule two instructions every cycle. Hence, at most all-ones detection logic must be able to set the one bit field for two instructions every cycle. Once the classification bit is set by the all-ones detector we use a counter to count the number of warps pending in each group. We use two five bit counters to count the number of warps in each group. Also to keep track of the idle-detect time of each SIMT lane a 3-bit counter is added per-lane. The dynamic power of the 3-bit counter is 0.1uW per SIMT lane, about 0.2% power overhead per SIMT lane.

4. EVALUATION

4.1 Evaluation Methodology

We evaluated our proposed techniques for performance and energy saving using GPGPU-Sim v3.02 [7]. 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 units per SM. Each SP unit contains 16 double-frequency SIMT lanes, each with individual INT and FP pipelines (total of 32 SIMT lanes per SM) .GPUWattch[22] is used for power estimations. We selected benchmarks to cover a wide range of scientific and computation domains from several benchmark suites including Rodinia [10], Parboil [2], and ISPASS [7]. For all the power gating results presented in this section, unless specified otherwise, we assume a default idle-detect window of five cycles and a break-even time of fourteen cycles and N_{phase} value is selected as 300 cycles. The base machine uses a two-level scheduler, implemented as described in [15].

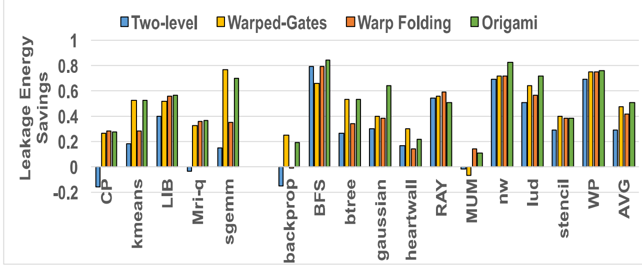
4.2 Energy Impact

Figure 10 shows the static energy savings after accounting for the power gating overhead for the integer and floating point pipelines. The results are normalized to the baseline machine that uses the two-level scheduler and does not apply power gating at the execution units level. 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 scheduler (the first bar in Figure 11) saves 29% and 34% of the leakage energy for integer and floating point units, respectively. When applying Warp Folding on top of the two-level scheduler, we are able to save 41% and 43% of the integer and floating point pipelines leakage energy respectively. This demonstrates the effectiveness of Warp Folding in converting wasteful fine-grain pipeline bubbles into useful power gating opportunities. In addition, Warp Folding is able to eliminate the negative energy savings for backprop, CP and MUM workloads by folding the warps long enough to guarantee a positive net energy savings every time we power gate.

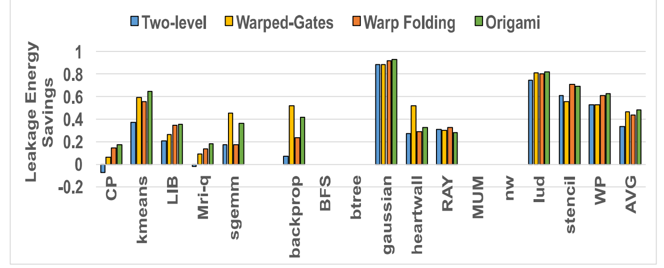
When using Warp Folding with Origami scheduling, the energy savings increase to 49% and 46% for the integer and floating point pipelines, respectively. The extra savings are due to the ability of the Origami scheduler to coalesce fine-grain bubbles in each pipeline by scheduling the instructions based on the instructions type and threads activity. Overall, the proposed Origami technique is able to save 22% and 15% more leakage energy when compared to the baseline machine. Figure 10 also shows the static energy savings of Warped-Gates [6]. Origami technique is able to save leakage energy slightly better than the Warped-Gates technique. However, as stated before, the main goal of the Origami solution is to achieve better savings than the Warped-Gates techniques without degrading the performance as in Warped-Gates as will be shown in the next subsection.

To estimate the overall savings at the execution units level we extracted the leakage power and the per access energy for the FP and the INT pipelines from the GPUWattch [22]. Then we multiplied the energy numbers by the number of times the FP and INT pipelines are activated for each application. Our estimates show that the total static energy normalized to the total energy of the execution units(i.e static + dynamic) accounts for 50% of the total execution units energy. Hence applying our proposed techniques saves 25% of the execution units total energy. In addition, since the execution units in the GTX480 account for almost 20% of the total GPU power [22], our proposed techniques are able to save 5% of the the total GPU power.

We also analyzed the reasons for improvement in the energy savings by looking at the number of power gating events

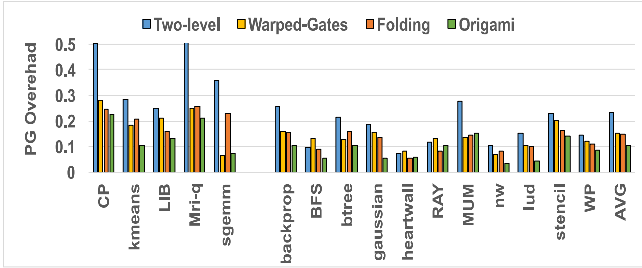


(a) INT pipeline

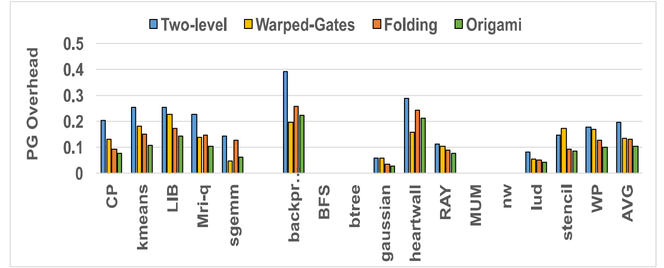


(b) FP pipeline

Figure 10: Execution units leakage energy savings (10% of the total GPU energy)



(a) INT pipeline



(b) FP pipeline

Figure 11: Execution units power gating overhead

that were activated. As mentioned before, each switch to the power gating state adds additional power overhead equal to the break-even time. Also each switch out of the power gating state adds additional delay cycles because of the wakeup latency. Using Origami we are able to coalesce many short idle periods into fewer but longer idle periods. In addition the proposed Warp Folding policies keep folding the warps continuously for at least N_{Fold} cycles that are required to amortize the power gating overhead.

As a result, the number of power gating events reduced but at the same time the amount of time spent in each power gating phase is longer. Figure 11 shows the power gating overhead for the integer and the floating point pipelines. The power gating overhead is simply the number of power gating events multiplied by the break-even time. The first column shows the overhead of traditional power gating scheme. The second column shows the power gating overhead for the Warped-Gates [6] technique. Compared to traditional power gating scheme, Warped-Gates is able to reduce the power gating overhead for the integer and floating point pipelines by 6% and 9%, respectively. Warp Folding alone is able to reduce the power gating overhead for the integer and floating point pipelines by 7% and 9% respectively. When combined with the Origami scheduler, power gating overhead is reduced by 9% and 14% respectively.

4.3 Performance Impact

Our proposed Warp Folding techniques apply different folding policies based on the demand for the resources. Hence the folding distribution is different for each workload. Figure 12 shows percentage of time the warps are folded. The first column shows the percentage for the integer pipeline and the second column shows the percentage for the floating point pipeline. On Average, 35% of the warps are folded. As mentioned before the folding decision depends on the instruction mix and the resource utilization within each workload. For example, mri-q workload has more FP instructions than INT instructions. Hence, the INT instructions have been folded 60% of the time while the FP instructions have been folded only 35% of the time. In addition, and due to the adaptive policy, some workloads, like backprop and heartwall, have not been folded more than 20% because the adaptive policy disabled Warp Folding during high utilization phases.

Figure 13 shows the execution time of the proposed techniques normalized to the execution time without power gating. The first column shows the performance overhead when the conventional power gating technique is used [17]. The second column shows the performance overhead of Warped-Gates technique [6]. The third and the fourth columns show the performance overhead of the Warp Folding only and the Origami techniques. As shown, Warped Gates has 6.8% increase in the execution time when compared to the baseline machine. The reason for the performance loss is the blackout technique [6] that shuts down the entire SIMT lanes cluster during the power gating state. On the other hand, applying Warp Folding naively reduces the performance overhead by 1% over Warped-Gates. When integrated with the type-aware scheduling and the lane-aware scheduling enabled by the Origami scheduler the overall performance overhead goes down to only 0.5%. Hence, Origami is able to sustain the energy savings without impacting average performance across our workloads. The performance improvements are due to the adaptive folding policies that takes into account the application utilization. Surprisingly, some workloads like backprop and btree show performance improvement. The reason

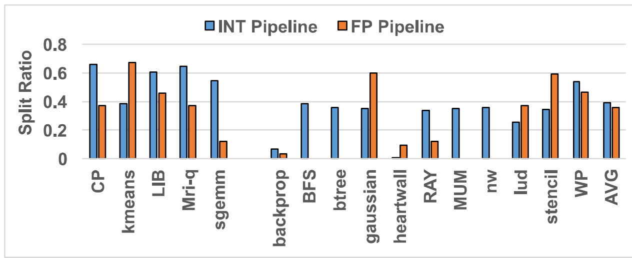


Figure 12: Percentage of time Warp Folding is enabled

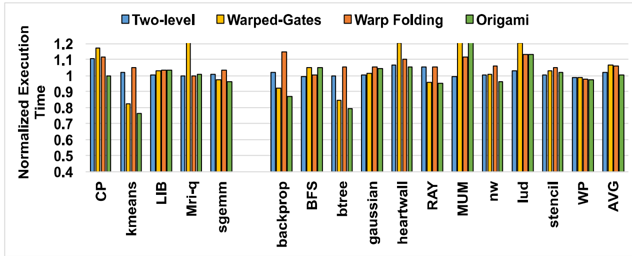


Figure 13: Execution time normalized to the baseline machine with no power gating

for this anomalous behavior is that dynamically dividing the warps into scheduling groups based on the instruction type and active mask results in a different warp scheduling order, which in some cases reduces the contention on the memory bus, which in turn improves performance.

4.4 Sensitivity Studies

We have also done an extensive set of sensitivity studies to varying power gating parameters. When the wakeup delay is increased to 9 cycles the performance overhead increases by 1.4%. When the break-even time is increased to 19 cycles the degradation in performance is less than 0.2%. On the other hand, The total energy savings for the INT and the FP pipelines degraded only by 4% and 1% when the break-even time is 19 cycles and the wakeup delay is 9 cycles,respectively.

4.5 Hardware Overhead

For the power and area overhead we implemented the extra blocks needed by the proposed idea in RTL. The blocks include the all-ones detection logic, the additional blocks in the lane-shifting and re-shifting stages and the extra counters used by the scheduler. We used Synopsis design compiler and the NCSU PDK 45nm [1] to synthesize the additional logic. The dynamic power of the additional logic is 0.64 mW per SIMT lane. The dynamic power of the additional logic is counted every time the logic is activated by issuing a warp to one of the SP units. Since there maybe a deviation in the power numbers generated by our synthesis and GPUWattch, we normalized our power overhead to the execution units overhead by synthesizing our own execution unit built from 1 adder (INT/FP) + 1 multiplier(INT/FP) + 1 logic unit and the power overhead is 1.6% of the power of the execution units. Also we measured the power overhead for each benchmark based on the power numbers reported by GPUWattch [22] for that benchmark. The results show that the power overhead is less than 1% of the power of the execution units. The area of the additional logic is 0.055 mm^2 and the total area of the execution unit is 32.7mm^2. As a result, our total area overhead is 0.15%.

5. RELATED WORK

GPU schedulers: Since the scheduling decision can have a great impact on the GPUs performance and power, GPUs scheduler has been the target for different types of optimizations. The two-level scheduler [15] is an optimization over prior schedulers that placed all pending and active warps in a single queue. The proposed scheduler has a positive impact on performance in addition to its role in reducing the register file cache size proposed in that work. Narasiman et. al., [27] proposed another two-level scheduler that improves performance and reduces stalls due to memory requests by dividing warps into fetch groups. The two level scheduler gives priority to each fetch group and rotates fetch groups whenever a long latency event occurs. By limiting the scheduling decisions to warps within just one fetch group the number of concurrent memory requests generated are reduced thereby reducing memory bottlenecks. Rogers [31] proposed a warp scheduler to improve cache locality and to reduce cache conflicts. Jog [20] proposed a warp scheduler to enable efficient pre-fetching policies. In our work we are proposing a new power gating aware scheduling scheme that improves the power gating opportunities with a negligible performance overhead.

CPU Power aware schedulers: Power aware schedulers for CPUs and multicore systems have been studied extensively [8, 32, 25, 26]. Previous work focused on dynamic power aware scheduling and DVFS decisions based on available task and service time. Our proposed schedulers are based on the GPU execution model and targets energy efficiency through improving the power gating potential of the GPU execution units.

Power gating: Power gating techniques have been widely applied in microprocessors [17, 23, 24], caches [12, 33], and NOCs [11, 35]. In this work we showed that applying power gating at the SM level is conservative and there are plenty of power gating opportunities when the technique is applied at SIMT lanes level. Specifically, we showed that instead of relying on existing idleness for power gating opportunities, as in prior work, we create opportunities by folding warps to create contiguous idleness in high order lanes.

GPU power saving: Power efficiency of the GPUs micro-architectural blocks has been extensively studied. At the register file level, several works [5, 15] proposed techniques to save dynamic and static power of the GPUs register file using circuit level and micro-architectural techniques. At the execution units level, Leng [22] explored clock gating and DVFS to save dynamic power of the execution units based on the mask activity and execution phases. Gilani [16] proposed several techniques to save the execution units and the register file dynamic power. The proposed technique takes advantage of the similarity in the data values in the GPU workloads to save power. Also they proposed combining two simple instructions into one composite instruction that can be executed by an enhanced fused-multiply-add units. The static power was not considered in Leng's and Gilani's work, which is the focus of this paper. The authors in [34] proposed detecting the divergence pattern of the running warps and run the warps with the same patterns to create gating opportunities. Their approach is only applica-

ble when the under utilized active masks appear repeatedly while our proposed approach works even when applications have a fully utilized active mask. In this work we applied lane shifting that is able to create the matching opportunities without any need to lookup for exact matching.

Warp Size: The authors in [30] proposed the variable warp sizing (VWS) technique that targets improving the performance of the divergent applications. In VWS the warp size and the SIMT lane cluster size change at run time in order to improve performance of the divergent application. VWS with a smaller warp size as a base warp unit and then gangs them to form a larger warp. On the other hand, our work assumes that wider warps (32 lanes in our case) is in fact a good base unit and enables folding when there is divergence. As shown in figure 6 and figure 2 we evaluated our techniques using divergent and non divergent applications and the applications with no divergence still have significant pipeline bubbles. Our evaluation show that folding warps has no performance impact and can provide additional opportunities to power gate. Hence our technique is effective both for divergent and non-divergent codes. Finally, in our approach warp folding is a very localized operation which is done in the execution stage and the folded warps are combined back at the end of the execution stage. As such Warp Folding does not require as extensive pipeline modifications as the VWS approach.

6. CONCLUSION

In this paper we first showed the pervasiveness of fine-grain pipeline bubbles regardless of warp schedulers or workloads. In order to convert these wasteful pipeline bubbles into useful opportunities for energy savings, we proposed Origami . Origami consists of Warp Folding, a technique where warps are split into half-warps, which are scheduled in succession. This creates contiguous chunks of idleness in half of the SIMT lanes, which can be leverage for energy saving opportunity through power gating. Origami also consists of the Origami scheduler, a new warp scheduler that is cognizant of the Warp Folding process and tries to further extend the sleep times of idle execution lanes. By combining the two techniques we show that Origami can preserve energy savings that were achieved by prior approaches, while virtually eliminating all the performance overheads associated with prior GPU power gating solutions.

7. ACKNOWLEDGMENTS

This work was supported by the following grants: DARPA-PERFECT-HR0011-12-2-0020 and NSF-CAREER-0954211, NSF-0834798.

8. 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] Nvidia's next generation cuda compute architecture: Fermi. Technical report, Nvidia, 2009.

[4] Nvidia's 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 International Symposium on High Performance Computer Architecture*, 2013.

[6] M. Abdel-Majeed, D. Wong, and M. Annavaram. Warped gates: Gating aware scheduling and power gating for gpgpus. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

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

[8] 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 *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.*, 2008.

[9] N. Brunie, S. Collange, and G. Diamos. Simultaneous branch and warp interweaving for sustained GPU performance. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[10] 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)*.

[11] L. Chen and T. Pinkston. Nord: Node-router decoupling for effective power-gating of on-chip routers. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[12] 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*, 2002.

[13] W. Fung and T. Aamodt. Thread block compaction for efficient simt control flow. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[14] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[15] 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*, 2011.

[16] S. Gilani, N. S. Kim, and M. Schulte. Power-efficient computing for compute-intensive gpgpu applications. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture*, 2013.

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

[18] H. Jeon. *Resource underutilization exploitation for power efficient and reliable throughput processor*. PhD thesis, University of Southern California, 2015.

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

[20] 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.

[21] J. Kao and A. Chandrakasan. Dual-threshold voltage techniques for low-power digital circuits. *IEEE Journal of Solid-State Circuits*, 2000.

[22] 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.

[23] 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*, 2009.

[24] N. Madan, A. Buyuktosunoglu, P. Bose, and M. Annavaram. A case for guarded power gating for multi-core processors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[25] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[26] D. Meisner and T. F. Wenisch. Dreamweaver: Architectural support for deep sleep. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[27] 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*, 2011.

[28] I. Paul, W. Huang, M. Arora, and S. Yalamanchili. Harmonia: Balancing compute and memory power in high-performance gpus. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[29] M. Rhu and M. Erez. The dual-path execution model for efficient gpu control flow. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture*, 2013.

[30] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler. A variable warp size architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015.

[31] 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*, 2012.

[32] C. Scordino and G. Lipari. Using resource reservation techniques for power-aware scheduling. In *Proceedings of the 4th ACM international conference on Embedded software*, 2004.

[33] Y. Wang, S. Roy, and N. Ranganathan. Run-time power-gating in caches of gpus for leakage energy savings. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, 2012.

[34] Q. Xu and M. Annavaram. Pats: Pattern aware scheduling and power gating for gpgpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 2014.

[35] S. Yue, L. Chen, D. Zhu, T. M. Pinkston, and M. Pedram. Smart butterfly: Reducing static power dissipation of network-on-chip with core-state-awareness. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, 2014.