SwiftGPU: Fostering Energy Efficiency in a Near-Threshold GPU Through a Tactical Performance Boost

Prabal Basu Hu Chen Shamik Saha Koushik Chakraborty Sanghamitra Roy
 USU BRIDGE LAB, Electrical and Computer Engineering, Utah State University
 {prabalb, hu.chen, shamik.saha}@aggiemail.usu.edu,
 {koushik.chakraborty, sanghamitra.roy}@usu.edu

ABSTRACT

In this paper, we investigate the challenges of preserving energy-efficiency in a *Near-Threshold Computing (NTC)* GPU. Two key factors can significantly undermine the efficacy of GPUs at NTC: (a) elongated delays at NTC make the GPU applications severely sensitive to *Multi-cycle Latency Datapaths (MLDs)* within the GPU pipeline; and (b) process variation (PV) at NTC induces a substantial performance variance. To address these emerging challenges, we propose SwiftGPU—an energy-efficient GPU design paradigm at NTC. SwiftGPU dynamically adjusts the degree of parallelization, and the speed of the MLDs within each stream core of the GPU. The proposed scheme achieves an average of $\sim 15\%$ improvement in energy-efficiency over an ideal PV-free GPU, operating at the Super-Threshold regime. SwiftGPU incurs marginal area, wire-length and power overheads of 0.65%, 2.6% and 3.7%, respectively.

1. INTRODUCTION

GPUs have demonstrated substantial performance advantages over CPUs, by exploiting large *thread level parallelism* in data-intensive, highly parallel applications. With this immense performance advantage, a GPU's power consumption has also grown steadily, reaching 300W [12]. On the other hand, recent advances in *Near Threshold Computing (NTC)*, where the supply voltage is set slightly above the device threshold voltage, have shown a great promise in radically curtailing the chip power consumption. This fantastic improvement in energy efficiency of NTC circuits does come with a steep performance loss. To compensate for the performance loss in a single device, NTC circuits generally employ more devices to execute in parallel [5]. Consequently, a GPU is of particular interest at NTC, as it is built to exploit parallelism.

While conceptually intriguing, a GPU design for NTC presents two fundamental challenges. First, elongated delays in NTC circuits make the GPU applications severely sensitive to *Multi-cycle Latency Datapaths (MLDs)* within the GPU pipeline. When a GPU thread heavily utilizes one of these MLDs, like the functional units for transcendental operations, for example [10], the entire application performance can become latency sensitive, obliterating the advantages of parallel execution. Second, *Process Variation (PV)* presents a tremendous

design challenge for NTC systems. In fact, a wider spatial-spread of the cores of a GPU can further exacerbate the core-wise performance asymmetry from PV at NTC.

To tackle these challenges, we identify the unique opportunities arising in the NTC regime. For example, the emerging NTC era ushers an intriguing possibility of up to 10X improvement in circuit performance, at a moderate energy cost [15]. Such a large performance enhancement was never a possibility at STC, since STC circuits already operate near their minimal delay region [15]. By systematically exploiting such device level characteristics at the circuit-architecture layer, we propose *SwiftGPU: an energy-efficient GPU design paradigm for NTC*. Inherently PV-aware, SwiftGPU tackles key challenges of NTC GPU designs, by dynamically speeding up the MLDs, and manipulating the thread level parallelization. Collectively, these techniques mitigate the performance sensitivity to MLDs, and performance imbalance from PV, substantially improving energy efficiency at NTC.

We make the following contributions in this paper:

- We uncover the emerging performance and energy-efficiency hazards as GPUs operate at NTC (Section 2).
- We propose SwiftGPU, a collection of low-overhead design time and run-time solutions, to address the emerging challenges in an NTC GPU. By detecting the utilization pattern of MLDs in a stream core, SwiftGPU selectively speeds up their execution in the stream cores to improve the NTC GPU energy efficiency (Section 3).
- Using an elaborate cross-layer methodology (Section 4), we demonstrate an average improvement of 14.8% in energy efficiency, over an ideal PV-free STC GPU, across a range of emerging GPGPU applications. Using synthesis, place and route of a GPU RTL, augmented with SwiftGPU, we find the area, wire-length and power overheads to be 0.65%, 2.6% and 3.7%, respectively (Section 5).

2. MOTIVATION

In this section, we explore the emerging efficiency hazards for GPUs, operating at NTC (Section 2.1). Using a rigorous cross-layer methodology (Section 2.2), we analyze the GPU performance trends at NTC (Section 2.3), and uncover the potency of a strategic performance boost, to improve the energy-efficiency of NTC GPUs (Section 2.4).

2.1 Efficiency Hazards in NTC GPUs

An increase in the number of Compute Units (CUs)¹, is expected to recoup the performance loss due to the frequency degradation at NTC. A key research question is, *whether the*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

DAC '16, June 05-09, 2016, Austin, TX, USA

Copyright 2016 ACM. ISBN 978-1-4503-4236-0/16/06 ...\$15.00

DOI: <http://dx.doi.org/10.1145/2897937.2898100>

¹A *Compute Unit* is the fundamental unit of computation, comprising multiple SIMD units, referred as *stream cores*.

emerging GPGPU applications can exploit the increased thread level parallelism at NTC, to sustain the STC efficiency. We discuss a few performance bottlenecks, that plague energy-efficient operation of NTC GPUs.

- **Nature of Datapath Usage:** Compute-intensive GPGPU applications are likely to exhibit a high MLD utilization and increased sensitivity to their latencies. Consequently, operating a GPU at NTC will prove to be an inefficient design choice, unless we can selectively speed up the execution of the MLDs.
- **Thread-level Data Dependency:** As an NTC GPU has more cores than its STC counterpart, each core in the former will have less concurrent threads to execute, potentially escalating the data-dependency within a CU. The increased data-dependencies can make the execution *latency sensitive*, degrading the application performance.
- **Process Variation:** NTC circuits are susceptible to within-die process variation, severely altering the CU frequencies and leakage power of the GPU. According to Chang et al., the maximum within-die frequency variation at 22-nm, can be $\sim 200\%$, while operating at the NTC region [4]. It is imperative to efficiently counteract the effect of PV in an NTC GPU, to sustain a high performance per watt.

2.2 Methodology

We use Multi2Sim 4.2 [23] to model AMD’s Evergreen architecture GPU – Radeon 5870. We use the GPGPU benchmarks from AMD’s Accelerated Parallel Processing SDK suite [1]. Based on the frequency scaling factors in previous works [22, 21], we assume the CU frequency at NTC to be 25% of the CU frequency at STC. To maintain *identical theoretical compute bandwidth across the NTC and STC systems*, we set the number of CUs at NTC to be 4X as that of the STC. The original MLD latencies are between 4 - 40 cycles, for both STC and NTC. To model process-variation, we use VARIUS-NTV [8]. We defer a detailed methodology description to Section 4.

2.3 NTC Performance Trend

Figure 1 shows the performance results of three cases: NTC (PV-free NTC GPU), NTC-PV (PV-infected NTC GPU) and NTC-PV-Speedup (PV-infected NTC GPU with 4X MLD boost), normalized to a baseline PV-free STC GPU. All the benchmarks exhibit a worse performance in the PV-infected NTC GPU, compared to the baseline. We observe a maximum performance degradation of 90% in *EigenValue*. We also notice that the solo impact of PV, brings about severe performance loss with respect to the PV-free NTC GPU. However, 8 out of 17 benchmarks, perform better than the baseline, when all the MLDs are statically boosted by a factor of 4X. For example, *SobelFilter* exhibits a staggering 80% performance improvement with the MLD speedup. The performance degradations in *EigenValue* and *FastWalshTransform*, are also reduced from about 90% to about 60%, by the MLD speedup.

2.4 Significance

Our initial results reveal that the emerging GPGPU applications are very sensitive to various NTC hazards, as well as, to the variations in the MLD latencies (Section 2.3). Such a performance sensitivity can have a profound impact on the energy efficiency of the GPUs. Figure 2 shows a preliminary investigation of this impact. With a static 4X boost applied to the MLDs, the PV-infected NTC GPU has a 31.5% reduction in the average energy consumption, which is 3.5% lower

than the energy consumed by the PV-free STC GPU. This energy benefit comes from a substantial reduction in the leakage energy, a crucial manifestation of the MLD boost. Inspired by these circuit-architectural insights, we will embark on exploring SwiftGPU—a novel energy-efficient GPU design paradigm at NTC. SwiftGPU incorporates *Self-Adaptive Sprint (SAS)* technique, to selectively speed up the processing in the stream cores, to improve the NTC GPU efficiency.

Figure 1: Performance of NTC GPU (Normalized to STC GPU).

Figure 2: Energy of an NTC GPU (Normalized to an STC GPU).

3. SELF-ADAPTIVE SPRINT IN NTC GPUS

In this section, we present Self-Adaptive Sprint (SAS), a novel technique to promote energy-efficient execution, by tackling the NTC hazards in GPUs. We discuss the overview of SAS in Section 3.1 and present its circuit-architectural aspects in Sections 3.2 and 3.3. We conclude with the dynamic control strategies of SAS in Section 3.4.

3.1 Overview

Figure 3 gives an overview of our proposed SAS technique. The *SAS Controller* dynamically manages the execution speed of the CU MLDs. To tackle the impact of PV, we adopt a number of crucial design strategies in SAS, ranging from the use of tunable voltage rails, to a meticulous selection of the MLD speeds. To support several datapath speeds, we augment the underlying power-delivery network to allow three different supply voltage rails: V_{dd_H} , V_{dd_M} and V_{dd_L} , respectively. The SAS controller monitors the runtime hardware utilization of various CU MLDs, and dynamically adjusts the MLD speed to improve the energy efficiency of the entire system. We next outline a few key design aspects in SAS.

3.2 Circuit-level Support for SAS

Providing multiple voltage-rails in the CU datapaths raises several critical design challenges, discussed next.

- We need to select the specific voltages offered in the rails based on the technology node, circuit level delay charac-

Figure 3: Overview of SAS.

teristics of underlying devices, and associated overheads. For example, for the 22-nm technology node and a nominal frequency of 175 MHz, we determine that Vdd_H , Vdd_M and Vdd_L can be set to 0.6V, 0.42V and 0.35V, respectively, to enable 4X, 2X and nominal (1X) sprint speeds in the CU MLDs. We employ three off-chip voltage regulators for the nominal and higher voltage rails [16]. Using a transition-time test setup similar to [16], we observe that the switching between different supply voltages can complete within one cycle (5.7 ns) of the NTC GPU.

- The voltage required to sustain a specific sprint, is likely to encounter a spatial variance due to the PV, compounding the complexity of online boost. To address this problem, we first ascertain and record the Vdd_H , Vdd_M and Vdd_L , for all the CUs, after fabrication. We endow each CU with its own low-overhead on-chip voltage regulator [6], to control only its MLD sprint. As the three off-chip regulators can deliver the required boost at the nominal frequency, such on-chip regulators can essentially work at a narrow voltage range, preserving a high conversion efficiency. Moreover, the energy overhead associated with the voltage conversion, is significantly reduced due to (a) relatively low power consumption of the MLDs (compared to the entire CU), and (b) sporadic occurrence of the MLD boost. During the kernel execution, the SAS controller boosts the MLDs, by selecting the appropriate voltage for a CU.
- We need to carefully consider the transition time between different voltage rails. To accomplish runtime transitions, we augment the voltage rails with a set of low-overhead level-shifters, connected to the CU MLD components. Such a level-shifter consists of 24 transistors, and only adds a marginal delay to the original MLDs [17].

3.3 Micro-architecture Support for Functional Correctness

Allowing variable latencies in the datapaths creates several intriguing design challenges in the micro-architecture, in order to retain its functional correctness.

- A multi-cycle datapath may complete its computation within a single cycle under a sprint mode. However, many of the existing CU MLDs are pipelined, thereby preventing the high-speed computation to propagate to the output. To resolve this issue, we dynamically allow intermediate pipeline registers in these datapaths to become transparent, forwarding the high-speed computation to the output.
- We must allow issuing dependent instructions in a timely manner under the sprint mode, so as to avoid unnecessary

delay. Instruction execution is often tracked using a fixed set of countdown registers. We carefully alter these countdown registers, to reflect speedy execution and subsequent issue of dependent instructions in the pipeline.

- During the transition between different voltage rails, the pipeline operations may become unstable. To prevent incorrect execution, we initiate a pipeline flush before a transition. The overheads from these pipeline flush operations directly influence our SAS control strategy (e.g., high overhead prohibits frequent transitions), discussed next.

3.4 Dynamic Control of SAS

A key challenge in the design of SAS is dynamically deciding the right boost for speeding the datapaths in a CU. We outline a few aspects of this intriguing research challenge.

3.4.1 Necessity of Dynamic Control

The utilization of MLDs in the GPU processors can vary substantially, altering the effect of their boost on the GPU performance. A GPU application consists of one or more kernels. We observe a spatio-temporal variation in the core pipeline utilization, during a given kernel execution. Three major elements of these variations are: (a) spatial variation of a given MLD utilization across different *thread groups*²; (b) temporal variation of a given MLD within a single thread; and (c) different utilization of MLDs across thread groups. Moreover, process variation gives rise to a large difference among the maximum operating frequencies of the CU cores. To exploit the widely varying performance gains from MLD speedups, and the PV induced performance variation, we explore dynamic control of SAS.

3.4.2 Estimating Maximum Boost

Our initial investigation has revealed that a static boost of 4X, can effectively tolerate the PV, while delivering a comparable or better performance compared to a PV-free STC GPU (Section 2.3). We also notice that the performance sensitivity to MLD boost, steeply declines at the higher sprint speeds. Moreover, over-boosting the MLDs, can potentially aggravate the overall performance per watt, due to its substantial energy overhead. Therefore, in this work, we limit the maximum sprint speed to be 4-times the nominal sprint.

3.4.3 Predicting Boost

We use the kernel characteristics, the MLD utilization pattern and PV induced performance heterogeneity, to estimate the amount of boost.

Kernel Characteristics: Our preliminary investigation (Section 2) reveals that the performance sensitivity of a GPU to an MLD, depends on the number of concurrent threads executing on the stream core. A small number of concurrent threads, is unlikely to amortize the MLD latency due to a greater thread-level data-dependency across the CUs (Section 2.1). Given the block size (b) and the volume of the concurrent blocks (V) of the kernel, as well as, the volume of the stream cores on the GPU (V_c), the number of concurrent threads on a stream core (t_c) can be expressed as: $t_c = b \frac{V}{V_c}$.

A boost in a CU MLD is expected to improve the performance of the GPU only if, t_c is lower than the MLD latency. When a kernel is launched, we identify its dimensions (usually specified in the kernel), and set the maximum sprint

²We define *thread group* as the collection of threads belonging to the same instruction instance. It is equivalent to *warp* according to NVIDIA and *wavefront* according to AMD.

Algorithm 1 Set Maximum Sprint Speed

```
1: if  $t_C < \text{modalFULatency}/2$  then
2:    $\text{maxSprintSpeed} = 4X$ 
3: else
4:    $\text{maxSprintSpeed} = 2X$ 
5: end if
```

speed (maxSprintSpeed) of an MLD, according to Algorithm 1, where modalFULatency is the modal latency of the FUs (4 cycles in this work).

MLD Utilization: Due to variation in the utilization of the MLDs, we must be able to control the sprint speed of each MLD independently. To predict the sprint speed of each MLD, we track the usage of each MLD during every interval (100,000 cycles). At the end of each interval, we compare the usage of each MLD to a threshold (a certain percentage of the *instruction bundles*³). The MLDs with a higher usage than the threshold value for a particular interval, are given a sprint boost in the next interval.

Spatial Performance Heterogeneity: We exploit the performance variance of the CU cores in SAS, in order to mitigate the impact of PV. For example, we power-gate the slowest CUs, if they are not utilized by the running application. We adopt a two-pronged adaptation for the remaining CUs, executing a kernel. First, the sprint of the slowest CUs is controlled by their dynamic MLD utilization patterns. Second, to ensure a spatially balanced performance, we optimize the sprint of the faster CUs, based on their respective frequency differences from the slowest CU.

3.4.4 Improving the Energy Efficiency of SAS

The energy overhead associated with a higher sprint boost, can potentially reduce the effective performance per watt. In order to sustain an energy-efficient execution, we create a low-overhead monitoring network, that periodically adjusts the sprint speed, by approximately assessing the energy-efficiency of the last epoch. If a new sprint-speed offers better efficiency (reflected by a certain percentage improvement), we maintain the new sprint-speed till the next check. A degradation in efficiency leads to a switch-over to the older sprint-speed for an extended period of time. We optimize the iteration interval and sprint-speed, based on specific kernel behavior.

For a decrease in sprint speed, we double the MLD latency slots in each stream core. At the same time, we power-gate half of the active stream cores within a CU, and migrate their threads to the remaining cores [7]. In this way, each thread group will be folded and execute on the remaining cores in a round-robin fashion. On the other hand, an increase in sprint speed causes the threads to sprawl onto more stream cores. Upon thread migration, the dirty register data on a stream core will be written back to the local memory. This write-back can take up-to 1024 cycles, depending on the size of the dirty register data. We include this overhead while evaluating the performance of SAS (Section 4).

4. METHODOLOGY

Figure 4 shows the overview of our methodology. We consider multiple layers (e.g. architecture, device and circuit layer), to rigorously evaluate the efficacy of our techniques. Specific components of each layer are briefly outlined next.

4.1 Architecture Layer

We use Multi2sim as our architectural simulator [23]. Depending on the kernel dimensions, as well as, the FU usage

³An *instruction bundle* is a set of simultaneously issued instructions in a CU.

Figure 4: Overview of Our Cross-layer Methodology.

in each interval, we dynamically change the MLD latency (Section 3.4). For each benchmark, we run the GPU kernel for 1 iteration, considering all the possible MLD speedups. The statistics from Multi2sim is used as inputs to GPUWatch [11], to evaluate the power consumption of the GPU components. The performance and energy consumption of each interval is used to calculate the overall energy efficiency of SAS. Table 1 lists the architectural parameters for the STC and NTC GPUs, respectively. The thread group size at STC is statically set to 64. At NTC, we adjust the thread group size for different benchmarks, to exploit the increased volume of the stream cores. From STC to NTC, we increase the volume of the stream cores, as well as, decrease their frequency, by a factor of 4X. With this configuration, we expect the memory traffic to be similar in the STC and NTC GPUs. Therefore, we use the same configurations for the memory components, outside of the stream core (i.e. L2 cache and Device Memory), for both STC and NTC GPUs.

Parameter	STC	NTC
Frequency	700MHz	175MHz
# of stream cores	320	1280
Thread Group Size	64	64/32/16
MLD latency	4- 40 cycles	4- 40 / 1- 10 cycles
Local memory	32Kb, latency: 2-cycles	32Kb, latency: 2-cycles
L2 cache	8x256Kb, latency: 20 ns	8x256Kb, latency: 20 ns
Device Memory	B/W: 25.6GB/s, latency: 100 ns	B/W: 25.6GB/s, latency: 100 ns

Table 1: GPU configuration.

4.2 Device Layer

We use VARIUS-NTV [8] to model PV-induced CU-wise performance variation. Based on a cumulative distribution of the CU speed for 2,000 NTC GPU instances, we randomly generate a collection of 80 CUs, to represent a PV-affected NTC GPU. The fastest CU in our NTC GPU can run at 3X higher speed than the slowest CU. To evaluate the NTC energy consumption, we perform HSPICE simulations on various basic gates and circuits, for the 22-nm technology node [26]. The simulation parameters are adjusted according to the VARIUS-NTV model. We use: (a) a canonical 31-stage FanOut-of-4 (FO4) inverter-chain, and, ISCAS85 circuits, to represent various combinational logic in GPUs; (b) a 6T-SRAM cell and a 10T-SRAM cell [3, 24] to represent the memory configurations at STC and NTC GPUs, respectively. Previous works have shown that interconnect power is $\sim 50\%$ of the core dynamic power at STC [14]. We assume this percentage will remain the same at NTC, as voltage scaling equally affects the interconnect and the core dynamic power.

4.3 Circuit Layer

We amalgamate the power results of GPUWattch and the gate-layer simulation results (Section 4.2), to estimate the power consumed by an NTC GPU. We then scale the power values of each GPU component into NTC, according to the power scaling trend we have obtained for the representing circuits (Section 4.2). To evaluate the overheads of our technique, we augment a reference GPU RTL [2] with SAS. We feed the power characteristics of the basic gates to Synopsys Design Compiler, to obtain the power consumption of both SAS and the reference GPU design. Following synthesis, we perform place and route with Cadence Encounter, to get a more accurate estimation of the hardware overheads of SAS.

5. EXPERIMENTAL RESULTS

In this section, we analyze the efficacy and overheads of various comparative schemes (Section 5.1). We present the empirical evaluation of the usage threshold for SAS (Section 5.2), discuss the performance and energy-efficiency of various schemes (Section 5.3 and 5.4), and present the hardware implementation overheads of SwiftGPU (Section 5.5).

5.1 Comparative Schemes

- **Frequency Screening (FS):** This scheme loosely models the technique proposed in [9]. The GPU is run at a higher frequency, by disabling the slowest stream cores.
- **Thread Gather (TG):** This scheme resembles the core power-gating technique proposed in [7]. The threads are squeezed to a lesser number of CUs, to reduce the idle time in each CU (Section 1). The idle CUs are power-gated.
- **Static Sprint Execution (SSE):** This scheme employs a static single sprint mode for each benchmark. The sprint mode is set according to the dimensions of the executing kernel.
- **SAS:** This is our proposed technique. It incorporates TG and dynamic sprint control, described in Section 3.4.

5.2 Selecting Threshold For Dynamic Sprint

We empirically select a single usage threshold for all the FUs in our dynamic sprint control (Section 3.4.3). We explore three different thresholds: 20%, 30%, 40%, and measure their respective energy-efficiencies in terms of the Energy Delay Product (EDP). The sprint speed is ascertained by the dimensions of the executing kernel. Figure 5 shows that, all the thresholds have marginal differences in energy efficiency (less than 5%), in 10 out of 17 benchmarks. However, a threshold of 20% offers remarkably better energy-efficiencies in a few benchmarks (e.g. *BitonicSort*). So, we choose 20% as the usage threshold in our dynamic sprint control.

5.3 Performance Results

Figure 6 depicts the performance of various comparative schemes (Section 5.1). The results are normalized to an ideal PV-free STC GPU, without any enhancements. Both FS and TG deliver remarkably degraded performance than our techniques (SSE and SAS), as they employ less number of cores, and do not perform sprinted MLD execution. Although both FS and TG offer an overall similar performance, they extensively diverge for several benchmarks. SSE globally outperforms all the other schemes. The performance of SAS is lower than that of the SSE for most of the benchmarks, as sprint execution is dynamically throttled in SAS. *Despite a 75% reduction in operating frequency, the average performance of SAS*

enabled NTC-GPU is only 13.4% lower than the baseline PV-free STC GPU, across all the benchmarks.

5.4 Energy-Efficiency Results

Figure 7 shows the energy benefits of all the comparative schemes, compared to the baseline PV-free STC GPU. For most benchmarks, both FS and TG consume more energy than our techniques. A major source of this energy over-consumption comes from leakage, which is a significant fraction of the total energy in NTC circuits [20, 5], and proportional to the application execution time. However, in several benchmarks (e.g. *PrefixSum*), our techniques consume more energy than FS or TG. Such results signify the inefficacy of employing sprint execution in the slow cores. Therefore, it is prudent to power-gate the slower cores, and execute all the threads in the faster cores, at a higher frequency. Across all the benchmarks, SSE always has the higher energy consumption than SAS, as the former always enables sprint execution in all the MLDs. Overall, the average energy consumption of SAS is 33.6% lower than the baseline STC GPU.

Figure 5: Energy-efficiency of different Usage Thresholds. (Lower is better.) (Three thresholds are tested: TH0(20%), TH1(30%), TH2(40%). Results are normalized to TH0.)

Figure 6: Performance Comparison (Higher is better).

Figures 8 illustrates the energy efficiency of the comparative schemes, in terms of EDP. The results are again normalized to the baseline STC GPU. The escalated performance and the reduced energy consumptions of our schemes, compound to an overwhelming benefit in EDP, compared to FS and TG. SAS shows an 18.9% improvement in EDP over SSE, as SSE is more power-hungry than SAS. *SAS also outperforms the baseline STC GPU, with an EDP improvement of 14.8%.*

5.5 Implementation Costs

The hardware cost of SAS can be attributed to two major factors: the sprint execution infrastructure (Sections 3.2 and 3.3) and the SAS control framework (Section 3.4). As usually available in modern GPUs, we exclude the implementation cost of pipeline flush and instruction replay, as well as, the performance counters, that track the FU usage. Evaluated with our cross-layer methodology, the area, wire-length and power overheads for SAS implementation are 0.65%, 2.6%

Figure 7: Energy Comparison (Lower is better).

Figure 8: Energy-efficiency Comparison (Lower is better). and 3.7%, respectively, compared to the original GPU.

6. RELATED WORK

Several existing works, that aim at increasing the GPU energy-efficiency in the STC era, can be broadly classified into two categories: (a) throttling the hardware to save energy; and (b) refining thread-scheduling to increase the utilization of GPUs. In the first category, Wang et al. propose to save energy by dynamically power-gating the GPU caches, as cache arrays have a large leakage energy [25]. DVFS and core power-gating have also been employed to improve the throughput of a GPU under a power budget [7]. Also, Ma et al. improve the energy efficiency of the GPU-CPU architecture by dynamically distributing the workloads, and throttling the frequency of the GPU cores and memory [13]. In the second category, Pichai et al. improve GPU efficiency by replacing a warp, that encounters a TLB miss, with another warp [19]. Narasiman et al. propose a large warp and two-level warp scheduling to increase the utilization of the GPU resources [18]. However, no works in these categories, have explored the challenges that can arise for GPUs, operating at NTC.

Previous works have also explored the principles of the NTC region, as well as, its opportunity in microarchitectures. Pinckney et al. have assessed the limitations of parallelized near-threshold computation [20]. Dreslenski et al. have comprehensively analyzed the various challenges in NTC and suggested possible solutions [5]. Marković et al. have revealed that V_{dd} will be the most effective performance knob in NTC circuits [15]. But, to the best of our knowledge, no previous work explores the challenges and performance/energy benefits of Near-Threshold execution of GPUs.

7. CONCLUSION

In this paper, we propose SwiftGPU—an ingenious GPU design paradigm, to tackle the potential performance and energy-efficiency hazards at NTC. SwiftGPU employs SAS, that dynamically sprints the MLDs based on the dimensions of the GPU kernel, as well as, the MLD usage pattern during the kernel execution. Evaluated with our cross-layer method-

ology, our scheme achieves an average of $\sim 15\%$ improvement in energy-efficiency with marginal hardware overheads, over an ideal PV-free GPU, operating at STC.

Acknowledgments

This work was supported in part by National Science Foundation grants (CAREER-1253024, CCF-1318826, CNS-1421022, and CNS-1421068). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

- [1] AMD Accelerated Parallel Processing (APP) Software Development Kit .
- [2] MIAOW GPU. <http://miaowgpu.org>.
- [3] CALHOUN, B., AND CHANDRAKASAN, A. A 256-kb 65-nm Sub-Threshold SRAM Design for Ultra-Low-Voltage Operation. In *JSSC* (March 2007), vol. 42, pp. 680–688.
- [4] CHANG, L. AND OTHERS Practical Strategies for Power-Efficient Computing Technologies. *Proceedings of the IEEE* 98, 2, 215–236.
- [5] DRESLINSKI, R. G. AND OTHERS Near-Threshold Computing: Reclaiming Moore’s Law Through Energy Efficient Integrated Circuits. *Proceedings of the IEEE* 98, 2 (2010), 253–266.
- [6] HE, X. AND OTHERS SuperRange: Wide operational range power delivery design for both STV and NTV computing. In *DATE* (2014), pp. 1–6.
- [7] J. LEE, V. SATHISHA AND OTHERS Improving Throughput of Power-Constrained GPUs Using Dynamic Voltage/Frequency and Core Scaling. In *PACT* (Oct. 2011).
- [8] KARPUZCU, U. R. AND OTHERS VARIUS-NTV: A microarchitectural model to capture the increased sensitivity of manycores to process variations at near-threshold voltages. In *DSN* (2012), pp. 1–11.
- [9] KRIMER, E. AND OTHERS Lane decoupling for improving the timing-error resiliency of wide-SIMD architectures. In *ISCA* (2012), pp. 237–248.
- [10] LEE, V. W. AND OTHERS Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proc. of ISCA* (2010), pp. 451–460.
- [11] LENG, J. AND OTHERS GPUWatch: enabling energy optimizations in GPGPUs. In *Proc. of ISCA* (2013), pp. 487–498.
- [12] LUCAS, J. AND OTHERS How a single chip causes massive power bills GPU-SimPow: A GPGPU power simulator. In *Proc. of ISPASS* (April 2013).
- [13] MA, K. AND OTHERS GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures. In *ICPP* (2012), IEEE, pp. 48–57.
- [14] MAGEN, N. AND OTHERS Interconnect-Power Dissipation in a Microprocessor. In *Proc. of SLIP* (2004), pp. 7–13.
- [15] MARKOVIC, D. AND OTHERS Ultralow-Power Design in Near-Threshold Region. *Proceedings of the IEEE* 98, 2 (2010), 237–252.
- [16] MILLER, T. N. AND OTHERS Booster: Reactive Core Acceleration for Mitigating the Effects of Process Variation and Application Imbalance in Low-Voltage Chips. In *HPCA* (2012), pp. 1–12.
- [17] MOHANTY, S. P., AND PRADHAN, D. K. ULS: A dual- V_{th} /high-kappa nano-CMOS universal level shifter for system-level power management. *JETC* 6, 2 (2010).
- [18] NARASIMAN, V. AND OTHERS Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *MICRO* (2011), ACM, pp. 308–317.
- [19] PICHAI, B. AND OTHERS Architectural Support for Address Translation on GPUs. In *Proceedings of ASPLOS* (2014).
- [20] PINCKNEY, N. AND OTHERS Assessing the Performance Limits of Parallelized Near-Threshold Computing. In *DAC* (2012), pp. 1143–1148.
- [21] PU, Y. AND OTHERS Misleading energy and performance claims in sub/near threshold digital systems. In *Proc. of ICCAD* (2010), pp. 625–631.
- [22] SILVANO, C. AND OTHERS Voltage island management in near threshold manycore architectures to mitigate dark silicon. In *Proc. of DATE* (2014), pp. 1–6.
- [23] UBAL, R. AND OTHERS Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *PACT* (Sep. 2012).
- [24] WESTE, N., AND HARRIS, D. *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Addison-Wesley Publishing Company, USA, 2010.
- [25] Y. WANG, S. ROY, AND N. RANGANATHAN. Run-time Power-gating in Caches of GPUs for Leakage Energy Savings. In *DATE* (March 2012).
- [26] ZHAO, W., AND CAO, Y. Predictive Technology Model, June 2012.