ACE-GPU: Tackling Choke Point Induced Performance Bottlenecks in a Near-Threshold Computing GPU*

Tahmoures Shabanian, Aatreyi Bal, Prabal Basu, Koushik Chakraborty, Sanghamitra Roy
USU BRIDGE LAB, Electrical and Computer Engineering Department, Utah State University, Logan, UT, USA
{tahmoures,aatreyi.bal,prabalb}@aggiemail.usu.edu,{koushik.chakraborty,sanghamitra.roy}@usu.edu

ABSTRACT

The proliferation of multicore devices with a strict thermal budget has aided to the research in Near-Threshold Computing (NTC). However, the operation of a Graphics Processing Unit (GPU) at the NTC region has still remained recondite. In this work, we explore an important reliability predicament of NTC, called *choke points*, that severely throttles the performance of GPUs. Employing a cross-layer methodology, we demonstrate the potency of choke points in inducing timing errors in a GPU, operating at the NTC region. We propose a holistic circuit-architectural solution, that promotes an energy-efficient NTC-GPU design paradigm by gracefully tackling the choke point induced timing errors. Our proposed scheme offers 3.18× and 88.5% improvements in NTC-GPU performance and energy delay product, respectively, over a state-of-the-art timing error mitigation technique, with marginal area and power overheads.

CCS CONCEPTS

• **Hardware** → **Physical design (EDA); Timing analysis; Very large scale integration design; Power and energy;**

KEYWORDS

Near-Threshold Computing, Choke Point, Timing Violation

ACM Reference Format:

Tahmoures Shabanian, Aatreyi Bal, Prabal Basu, Koushik Chakraborty, Sanghamitra Roy. 2018. ACE-GPU: Tackling Choke Point Induced Performance Bottlenecks in a Near-Threshold Computing GPU. In *ISLPED '18: ISLPED '18: International Symposium on Low Power Electronics and Design, July 23–25, 2018, Seattle, WA, USA*, Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 6 pages. <https://doi.org/10.1145/3218603.3218644>

1 INTRODUCTION

Evolution of Graphics Processing Units (GPUs) from specialized graphics chips to generalized computing devices has ushered a new era in parallel computing. GPUs offer a substantial performance improvement, over CPUs, for highly parallel applications. However,

*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.

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISLPED '18, July 23–25, 2018, Seattle, WA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5704-3/18/07...\$15.00

<https://doi.org/10.1145/3218603.3218644>

the extensive thread-level parallelism in GPUs are usually accompanied with a large power consumption [15]. To avoid hitting the power wall, while still maintaining the benefits of parallel computing, low-power GPUs are the need of the hour. Consequently, Near-Threshold Computing (NTC) has emerged as a promising design paradigm for energy-efficient GPUs [5]. In this paper, we explore a significant reliability concern of the GPUs, operating at the NTC region (NTC-GPUs). NTC-GPUs, besides offering a high energy-efficiency, suffer from a substantial process variation (PV) induced delay variation [19]. Moreover, the large die size of a GPU makes it more susceptible to within-die PV, compared to a CPU [3]. The problem is further complicated at NTC due to a manifold increase in the number of stream cores, compared to traditional GPUs, operating at super-threshold (STC) voltages [5]. Our work in this paper focuses on *Choke Points*—a crucial PV-inflicted performance bottleneck in NTC-GPUs. A choke point is a small group of PV-affected logic gates in a circuit path, that dominates the path delay, so as to transform a short delay path into a critical path [8]. Choke points are created due to a drastic gate delay variation at NTC. Being an artifact of the fabrication process, the formation of choke points cannot be precisely anticipated or tackled at the design time [4]. As a result, a dynamic adaptive technique is required to tackle choke points in fabricated chips. Recently, Bal et al. have proposed an in-situ timing speculation technique to predict and recover from choke point induced timing errors in a simple out-of-order CPU pipeline at NTC [4]. In this work, we show that naively applying such an error recovery mechanism to a SIMD¹ pipeline does not boost the NTC performance to a similar extent as [4] (Section 5). Hence, employing a cross-layer methodology, we propose a choke point resilient NTC-GPU architecture, referred to as *Adaptive Choke Error-resilient GPU (ACE-GPU)*. Exploiting the recent history of timing errors in the GPU compute units (CUs), ACE-GPU employs a novel thread mapping strategy to minimize the occurrences of future timing errors, while incurring minimal overheads. To the best of our knowledge, this is the first work that analyzes choke point induced performance bottlenecks in NTC-GPUs, as well as, proposes a timing error mitigation technique to improve the GPU power-performance. The following are the key contributions of this work:

- We elaborate why existing timing errors mitigation techniques are grossly ineffective in alleviating the choke point induced performance loss in an NTC-GPU (Section 2.2).
- We explore the impacts of choke points in GPU power-performance at various operating conditions, with an existing timing error mitigation scheme (Section 2.4).

¹GPUs are comprised of many single instruction multiple data (SIMD) units, executing several parallel threads simultaneously.

Figure 1: Figure 1(a) shows that the presence of choke points increases the delays of the decode and execute stages, compared to the respective nominal delays of those stages. Figure 1(b) and 1(c) exhibit the drastic increase in the performance penalties and energy consumptions respectively, for RecursiveGaussian, as more CUs are employed to cope with the decreasing VF levels towards NTC. For figure 1(b) and 1(c), the results are normalized to the corresponding STC values.

- Our proposed NTC-GPU design paradigm—ACE-GPU—augments a baseline GPU architecture in order to tackle choke point induced timing errors in GPUs (Section 3).
- Using a range of general purpose GPU (GPGPU) applications, we evaluate the performance benefit, energy-efficiency gain, and overheads of ACE-GPU, compared to traditional timing error mitigation techniques (Section 5).

2 MOTIVATION

In this section, we illustrate choke point as an important reliability problem in NTC-GPUs. Section 2.1 elucidates the significantly higher susceptibility of GPUs to choke point induced timing errors. We discuss the limitations of existing timing error mitigation techniques in Section 2.2. Using a cross-layer methodology (Section 2.3), we present the choke point induced delay variabilities in GPU pipelines (Section 2.4), and motivate the design of ACE-GPU (Section 2.5).

2.1 Background

Due to a greater sensitivity to PV, NTC systems are significantly more susceptible to choke point induced delay variabilities than the STC ones [4, 6]. Furthermore, Aguilera et al. have demonstrated that GPUs are more impacted by within-die PV, compared to traditional multicore CPUs, due to a large number of cores in the former [3]. To exploit the large thread-level parallelism of the GPGPU applications, an NTC-GPU employs more cores than their STC counterparts [5]. Therefore, the probability of timing violations due to the formation of choke points notably increases in an NTC-GPU, thwarting the energy benefits of NTC operation.

2.2 Can Existing Timing Error Mitigation Techniques Tackle Choke Points in GPUs?

Researchers have proposed several circuit-architectural techniques to tackle timing errors in CPUs and GPUs. We briefly discuss why such techniques are ineffective in combating choke points induced timing errors in an NTC-GPU.

- **Timing Guardbands:** Bal et al. have shown that adaptive timing guardband techniques [17] are extremely energy inefficient in tackling choke point induced timing errors in an NTC processor [4]. Hence, employing such strategy is also futile in sustaining a reliable and energy efficient operation in NTC-GPUs.
- **Dynamic Error Detection and Correction:** Deploying shadow latches to dynamically detect timing errors (e.g., Razor [10]), and replaying the errant instructions, is one of the most popular and

effective techniques to improve the timing resiliency of a system. However, Razor does not identify the error patterns in a circuit which is crucial to effectively predict the recurring timing errors engendered by choke points [4]. Hence, existing Razor-based error detection and correction techniques are unlikely to efficiently deal with timing errors in an NTC-GPU.

- **Dynamic Choke Sensing:** By uncovering a unique relation between the instruction metadata and the corresponding sensitized choke points, Bal et al. have recently proposed an adaptive timing error speculation and recovery scheme for NTC processors [4]. However, a naive adoption of Bal’s technique in an NTC-GPU, leads to a significant performance loss (Section 5). This is due to the fact that a timing error in a single execution unit of a GPU, stalls all the SIMD lanes of a pipeline, thus effectively multiplying the error rate by the degree of parallelization.

Next, we briefly discuss our methodology to demonstrate the choke points induced delay variabilities in an NTC-GPU.

2.3 Methodology

We use the MIAOW GPU RTL [1], which is modeled on the AMD’s Southern Island GPU architecture, as our experimental platform. To synthesize the decode and execution unit of a CU, we use the 15-nm FinFET library from NanGate [16]. In order to model PV at STC and NTC for FinFETs, we use the VARIUS [18], VARIUS-NTV [12] and VARIUS-TC [13] models. We employ our home-brewed Statistical Timing Analysis tool to study the delay of the sensitized paths of the decode and the execution unit. For a conservative estimate, we consider PV-induced delays in randomly chosen 1% of the gates of a circuit. We discuss a detailed methodology in Section 4.

2.4 Results

Figure 1(a) shows the choke point induced additional delays in decode and execution stages of a CU. The delay values are normalized to the corresponding PV-free cases. Following are the three crucial observations from Figure 1(a). 1) The impact of choke points varies across different pipe stages. For example, at 0.45V, the decode stage exhibits an additional delay of ~2.8%, while the execute stage has an additional delay of ~18%. This variation is observed at other operating voltages too. The subsequent results presented in this paper are performed on the *execute stage* due to its greater susceptibility to choke point induced delay variations. 2) The choke point induced delays monotonically increase as the operating voltage approaches NTC values. For example, the additional delay of the execute stage at 0.45V is ~4.7× compared to that at 0.85V. This result confirms

that the choke point induced delay variations are more aggressive at NTC, than at STC. 3) There is no choke point induced additional delay for the decode stage at 0.85V. This is because, choke points may not always give rise to new critical paths in a fabricated circuit.

An increase in the number of CUs can improve the NTC performance by effectively exploiting the thread-level parallelism of the GPGPU benchmarks [5]. However, timing errors due to choke points can introduce a staggering performance penalty in an error correction scheme, severely throttling the performance of NTC. Figure 1(b) shows the variation in the performance penalties, as *RecursiveGaussian*, a GPGPU benchmark, executes with different number of CUs (with 100% utilization), endowed with *Razor* as the timing error mitigation scheme. The operating voltages considered for 16, 32, 64 and 128 CUs are: 0.85V, 0.65V, 0.55V and 0.45V, respectively. The leftmost and the rightmost bars in Figure 1(b) represent STC and NTC operating conditions, respectively. The results are normalized to the penalty for 16 CUs. We notice that an $8\times$ increase in the number of CUs, increases the performance penalty by $\sim 10.8\times$. This result reveals the inefficacy of *Razor* in tackling aggravated choke point induced timing errors at NTC.

For the same experiment, Figure 1(c) demonstrates a $\sim 5.3\times$ increase in the energy consumption at NTC (128 CUs), with respect to STC (16 CUs). A large fraction of this energy consumption at NTC comes from leakage energy. As leakage energy is proportional to the execution time, the high performance penalties at NTC (Figure 1(b)) diminish the energy-efficiency benefit of an NTC-GPU.

2.5 Significance

Our motivational results demonstrate a tremendous impact of choke points on the performance, as well as, energy consumption of the NTC-GPUs. Hence, in order to efficiently exploit the vast SIMD resources of an NTC-GPU, we need to explore a design paradigm that can dynamically predict and avoid imminent timing errors in the CUs, while incurring minimal performance and power overheads. Next, we explore a novel GPU design paradigm, that reclaims the energy-efficiency advantage of NTC.

3 ADAPTIVE CHOKE ERROR-RESILIENT GPU (ACE-GPU)

In this section, we discuss ACE-GPU, a novel design paradigm to tackle choke point induced timing errors in NTC-GPUs. We present the design overview in Section 3.1, and elaborate the design components in Section 3.2.

3.1 Design Overview

Figure 2 portrays the conceptual overview of ACE-GPU. The Ultra-Threaded Dispatcher (UTD), an integral component of a GPU, is responsible for assigning the thread blocks to the CUs [20]. In ACE-GPU, we alter the baseline thread block assignment policy of the UTD, to a choke point aware assignment strategy. This approach is fundamentally different than stalling the thread block execution to avoid imminent timing errors [4]. We augment the baseline GPU architecture with a *Choke-error Monitor Unit (CMU)* and a *Black List Unit (BLU)*. CMU is responsible for the identification, correction and prediction of choke point induced timing errors. The primary components of CMU are *Choke Error Sensing Table (ChEST)*, *Decision Unit Interface (DUI)* and *Threshold Comparator*. On the other hand, BLU works in tandem with CMU and UTD, to detect and avoid

Algorithm 1 Working Principle of DUI

```

1: procedure DUI(timing_info[IDs][opcodes])
2:   for all IDs  $\in$  timing_info do
3:     if (ID  $\in$  ChEST) then
4:       if (timing_info[ID][opcode]  $\in$  ChEST[ID]) then
5:         ChEST[ID][opcode].count++
6:       else
7:         if (opcode_field.full()) then
8:           opcode_field.evict(least_occurring_opcode)
9:         end if
10:        opcode_field.insert(opcode)
11:      end if
12:    else
13:      ChEST.evictPseudoLRU(ID)
14:      ChEST[ID][opcode].count = 1
15:      ChEST[ID].error_count++
16:    end if
17:  PDCP(ID)
18: end for
19: end procedure

```

the CUs that are severely impaired by choke point induced timing errors. Different components of CMU and BLU, along with their respective working principles are described next.

3.2 ACE-GPU Components

CMU and BLU dynamically monitor the timing errors occurring in different CUs, and communicate with the UTD to make an efficient thread block assignment that improves the NTC-GPU performance.

3.2.1 Choke-error Monitor Unit (CMU). CMU receives timing error details from each of the CUs, and records them for future references. Then, it populates the BLU with the IDs of the CUs that are rendered unreliable due to the severe impact of choke points. The roles of the components of CMU are explained next.

- Choke Error Sensing Table (ChEST):** The ChEST records the errors occurring in each CU, in terms of tuples of errant opcodes. In our implementation, each tuple corresponds to a CU ID, three errant opcodes encountered by that CU, and their number of occurrences (i.e., the number of times that opcode is executed on the CU, not the number of timing errors it caused). Additionally, each tuple has the total error count, which records the number of the timing errors detected in that CU. The ChEST is implemented as a Content Addressable Memory (CAM), to facilitate the table lookup. The number of entries in the ChEST is a trade-off between the associated overhead and the accuracy of sensing recurrent timing errors. A large entry size ensures a relatively high sensing accuracy, at the cost of relatively high area and power overheads. On the other hand, a smaller ChEST reduces the sensing accuracy thereby increasing the incurred penalty cycles. In our evaluations, we consider the entry size for ChEST to be 10 (for 128 CUs). The UTD looks up the ChEST before assigning thread blocks to CUs, in order to avoid imminent recurrent timing errors.
- Decision Unit Interface (DUI):** The DUI manages the error sensing, as well as, error correction inside the CMU. Algorithm 1 shows the working principle of the DUI. Upon receiving timing error information from the CUs, it updates the record of the encountered errors in the ChEST. The errors are detected with the help of double-sampling latches [10]. The DUI logs the errant opcodes, along with the corresponding CU ID, in the ChEST. If the ID is already listed in the ChEST, the DUI logs the opcode in the corresponding tuple of the ChEST. In case the tuple is already

Figure 2

Algorithm 2 Performance Degradation Control Procedure

```

1: procedure PDCP(ID)
2:   error_count = ChEST[ID].error_count + 1
3:   if (BLU[ID].tag == 0) then
4:     error_count += BLU[ID].old_error_threshold
5:   end if
6:   if (error_count > error_threshold) then
7:     if (BLU.size() == 0.1 * total_num_CU) then
8:       if (error_threshold < max_error_threshold) then
9:         for IDs ∈ BLU do
10:          if (BLU[IDs].tag == 1) then
11:            BLU[IDs].tag = 0
12:            BLU[IDs].old_error_threshold =
13:              error_threshold
14:          end if
15:        end for
16:        error_threshold += delta_threshold
17:      else
18:        BLU.randomReplaceCU(ID)
19:      end if
20:    else
21:      BLU[ID].tag = 1
22:      ChEST.evict(ID)
23:    end if
24:  end procedure

```

full, a new errant opcode replaces an existing opcode with least number of occurrences. Simultaneously, the DUI updates the total error count of the corresponding CU. If the ChEST becomes full, a new CU ID can replace an existing CU in the ChEST using a pseudo-LRU policy. Once an error is detected at the CU, the DUI stalls the current execution of the CU, and re-assigns the thread block to ensure an error-free execution (Section 3.2.3).

To adapt to high-error situations, the DUI employs a *Performance Degradation Control Procedure (PDCP)*, described in Algorithm 2. If the number of CUs in the BLU exceeds 10% of total CUs, the DUI increases the threshold value, considered by the Threshold Comparator. After that, it flushes the existing entries of the BLU, and stores the current threshold for which, the BLU is flushed. This stored threshold value is required to calculate the total number of timing errors, lest the flushed CUs once again appear in the ChEST. If the current threshold is beyond a preset maximum, it is no longer incremented. In that case, a CU in the ChEST, with the total error being more than the current threshold, can replace one of the existing CUs in the BLU, chosen randomly. However, we have not encountered this situation in our simulations.

- **Threshold Comparator:** The Threshold Comparator continuously monitors the total error counts of all the CUs in the ChEST. As soon as the total error count of a CU exceeds the threshold predefined in the comparator, the CU ID is removed from the ChEST, and added to the BLU. We dynamically update the threshold in our implementation, if a certain percentage of the total CUs are blacklisted. However, the threshold value cannot be increased beyond a preset upper limit.

Figure 2: The CMU and BLU form the backbone of the ACE-GPU architecture. The CMU performs the error management, including error logging and avoidance. The magnified version of the CMU shows its components. The Choke Error Sensing Table (ChEST) contains the error logs for the previously encountered error instances, and facilitates error avoidance. The Decision User Interface (DUI) is the brain of the CMU making all the error management decisions. The Threshold Comparator marks the CUs that are severely affected by choke points and logs them in the Black List Unit (BLU). The CUs listed in the BLU are not considered for imminent thread block assignment by the Ultra-Threaded Dispatcher (UTD).

Algorithm 3 UTD Assignment

```

1: procedure UTD(BLU, ChEST)
2:   ID_list = [1,...,128]
3:   for ID ∈ ID_list do
4:     opcode = opcode_to_be_assigned
5:     if ID ∈ BLU then
6:       next
7:     end if
8:     if ID ∈ ChEST && opcode ∈ ChEST[ID] then
9:       next
10:    end if
11:    ID.assign(opcode)
12:    break
13:  end for
14: end procedure

```

3.2.2 Black List Unit (BLU). The BLU records the CU IDs whose total error count exceeds a given threshold of errors. Any CU ID listed in the BLU is not considered for thread block assignment. The UTD looks up the BLU, before assigning a thread block to a CU. The size of the BLU is fixed at 10% (empirically determined) of the total CU count. If the number of blacklisted CUs exceeds the BLU size, the BLU is flushed by setting the valid tags of all of the CU IDs to zero, and the threshold is increased in the comparator. Like the ChEST, the BLU is also implemented as a CAM.

3.2.3 Ultra-Threaded Dispatcher (UTD). We modify a baseline round-robin UTD to work harmoniously with the CMU and the BLU. Algorithm 3 displays the remodeled UTD assignment. First, we consider each thread block to be composed of SIMD threads of only one type of opcode [14]. Second, before assigning a thread block to a CU, the UTD looks up the BLU. A CU, listed in the BLU, is discarded by the UTD. Next, it searches for the CU in the ChEST entries. Upon getting a match, the CU is considered for the thread block assignment, only if the opcode, corresponding to the thread block under consideration, is not present in that CU's tuple in the ChEST. If this condition is not met, the UTD then randomly selects one of the remaining CUs in the pool. When a thread block encounters a timing error in a CU, the UTD reschedules the errant thread block based on Algorithm 3. Rescheduling a thread block incurs a performance penalty that is dictated by the number of pipeline stages in the GPU.

We consider the performance and hardware overheads of ACE-GPU components in our evaluations (Section 5).

4 METHODOLOGY

In this section, we describe our comprehensive cross-layer methodology, to implement and evaluate ACE-GPU.

4.1 Device Layer

We simulate the HSPICE models of the basic logic gates (NAND, NOR, Inverter) to estimate their delay distributions, in the presence

Figure 3: Figure 3(a) shows the number of timing errors that each application encounters under different comparative schemes. The corresponding performances and energy-delay products of the applications are depicted in figure 3(b) and 3(c), respectively. The results are normalized to the corresponding values with Razor [10].

Parameters	Configurations
No. of CUs	128
Supply Voltage	0.45V
CU Frequency	400MHz
L2 Cache	8×768 KB, latency: 20 ns
Global Memory	B/W: 264 GB/s, latency: ~300 ns

Table 1: NTC-GPU configurations.

of PV, at different supply voltages. We use the 16-nm Predictive Technology Models (PTM) for the simulations. To estimate within-die PV, arising from both random and systematic variations, we consider the VARIUS [18] and VARIUS-NTV [12] models for STC and NTC, respectively. Further, we utilize the VARIUS-TC model to incorporate the FinFET characteristics [13]. We perform Monte Carlo simulations to gauge the propagation delay variations for 10,000 instances of each logic gate. The delay values are used in the circuit layer (Section 4.2) to realize choke points in the circuit.

4.2 Circuit Layer

In this layer, we perform two principle tasks. First, we augment an open-source reference GPU RTL [1] to implement the components of our proposed ACE-GPU architecture. We synthesize the reference and augmented GPU RTLs using Synopsys Design Compiler [7]. The VF values considered for synthesis at STC and NTC are (0.85V, 900MHz) and (0.45V, 400MHz), respectively. Second, we feed the synthesized netlists and the input vectors (Section 4.3), into our in-house Statistical Timing Analysis (STA) tool. The STA tool contains a library of the delay distributions for the basic logic gates at different operating voltages, obtained from HSPICE simulations (described in Section 4.1). The STA tool performs a timing analysis of the sensitized paths in the circuit netlist, for the given input vectors. Consequently, we obtain a clear idea of the impact of choke points on the path delays of a fabricated chip at the runtime. We use the resultant delay reports to evaluate the efficacy of the comparative schemes (Section 5.1).

4.3 Architecture Layer

We model an AMD Southern Island GPU in the Multi2Sim architectural simulator [20]. The GPU architectural parameters considered in this work are listed in Table 1. We instrument the Multi2Sim codebase to automatically extract the cycle-wise instruction meta-data (viz., opcodes and operands) from an execution unit, while running the GPGPU benchmarks from AMD’s APP SDK suite [2]. These metadata serves as the input vectors to the STA tool for

performing a dynamic path sensitization analysis and evaluating the impacts of choke points (Section 4.2).

5 EXPERIMENTAL RESULTS

In this section, we compare the performance and energy-efficiency benefits of ACE-GPU at NTC, with respect to other timing error recovery schemes in GPU. Section 5.1 highlights our comparative schemes, while Section 5.2, 5.3 and 5.4 discuss the error comparison, the relative performances and energy efficiencies of the schemes, respectively, across 8 GPGPU benchmarks. Section 5.5 presents the hardware overheads of the ACE-GPU of design.

5.1 Comparative Schemes

- **Razor:** This is a popular timing speculation technique that sporadically trims the timing guardband to allow intermittent timing errors in the pipelines [10]. The errors are detected by employing double-sampling latches at the pipeline boundaries. A thread-block reassignment is triggered to correct the timing errors.
- **Dynamic Choke Sensing (DCS):** This scheme offers detection, correction, as well as, prediction of choke point induced timing errors [4]. Originally proposed for a scalar CPU pipeline, we implement DCS for vector processors like GPUs. DCS employs a lookup table (LUT), in the form of a RAM, to store and lookup recurrent timing errors. A choke controller is used to manage the error detection, correction and prediction. For the error avoidance, the choke controller inserts a single stall cycle in the pipeline to allow the errant opcode to finish its execution without any error. Our scheme (ACE-GPU) is fundamentally different than DCS, because, instead of stalling the CUs for one complete cycle, we employ an efficient thread block mapping strategy for an error-free execution.
- **Adaptive Choke Error-resilient GPU (ACE):** This is our proposed scheme that also employs double-sampling latches to detect timing errors. ACE is designed to mitigate choke point induced timing errors in NTC-GPUs. The design of ACE is described in Section 3.

5.2 Error Comparison

Figure 3(a) demonstrates the relative number of timing errors that each benchmark encounters under different comparative schemes. The Y-axis values are normalized with respect to the number of timing errors with Razor [10]. The efficient error prediction and thread block assignment policy in ACE, lead to significantly lower number of timing error events, compared to Razor and DCS, across

all the benchmarks. The large difference in the number of timing errors between DCS and ACE for some benchmarks (e.g., $\sim 12.6\times$ in *ScanLargeArrays*), is due to the more efficient topology of ChEST in ACE, compared to the LUT in DCS.

5.3 Performance Comparison

Figure 3(b) illustrates the relative performances of the comparative schemes. The results are normalized to the performance of Razor. We notice that, on an average, ACE performs $3.18\times$ better than Razor, across all the GPGPU benchmarks. DCS has a better performance than Razor, as the latter encounters significantly more timing errors due to a lack of error prediction mechanism, incurring severe penalties. On the other hand, ACE performs remarkably better ($1.81\times$, on an average) than DCS. This is because: (a) ACE employs an efficient thread block to CU mapping strategy, obviating the need for a single stall cycle to avoid timing errors. (b) The BLU in ACE prevents assigning thread blocks to CUs that are severely impaired by choke points. For example, in *MatrixMultiplication*, many CUs are highly affected by choke points and hence, get listed in the BLU. However, DCS, lacking a tailored hardware unit like the BLU, is more susceptible to choke point induced performance penalties. (c) Unlike a RAM-based lookup table in Razor, ACE uses a CAM-based one (ChEST) to record the history of recent timing errors, resulting in a faster lookup.

5.4 Energy-Efficiency Comparison

We measure the energy-efficiencies of the schemes using the energy delay product (EDP) metric. Figure 3(c) shows the EDPs of the schemes, normalized to the EDP of the Razor. Our proposed scheme ACE is the most energy-efficient, offering an average of $\sim 88.5\%$ improvement in EDP over Razor. It is imperative to note that the relative power footprint of the hardware for ACE, is one order of magnitude more than those of the DCS and Razor. However, this high power footprint is amortized by an appreciable performance gain in ACE. Moreover, at NTC, a significant fraction of the total energy is contributed by the leakage energy, which is proportional to the application execution time. As the choke point induced timing penalties is the least in ACE (Section 5.3), it dramatically improves the leakage energy consumption of the NTC-GPU, for all the benchmarks. These performance and energy benefits aid to an improvement in EDP, making ACE a very energy-efficient GPU design paradigm at NTC.

5.5 Hardware Overheads

The area and power overheads of ACE-GPU, obtained from synthesis at the NTC operating condition, are 1.42% and 6.17%, respectively. The overheads are calculated compared to the respective values of a baseline CU, with no timing error mitigation scheme. We consider the power overheads in the evaluation of the EDP.

6 RELATED WORK

Recent studies related to our work can be broadly classified into two categories: (a) performance and reliability analysis of CPU cores at NTC, and (b) performance and reliability concerns of GPUs, operating at NTC. In the first category, Dreslinski et al. have explored the bottlenecks, as well as, opportunities of the NTC operation [9]. Karpuzcu et al. have explored the impacts of PV on NTC systems [11]. One of the remarkable reliability challenges, posed by PV at NTC, is *choke points* [8]. Bal et al. have explored this problem and

proposed an adaptive technique, to mitigate choke point induced timing violations in uncore CPUs, operating at NTC [4]. In the second category, Basu et al. have introduced a self-adaptive sprint technique, to mitigate PV induced performance variations in NTC-GPUs [5]. *However, to the best of our knowledge, ours is the first work that explores the impacts of choke points on the performance and energy-efficiency of NTC-GPUs.*

7 CONCLUSION

In this work, we propose ACE-GPU—a novel GPU design paradigm—that combats choke point induced performance bottlenecks at NTC. Using a cross-layer methodology, we show that ACE-GPU offers $3.18\times$ and 88.5% improvements in performance and EDP respectively, over a Razor-based timing error detection and correction scheme, while incurring marginal hardware overheads.

REFERENCES

- [1] 2015. MIAOW GPU - An open source RTL implementation of a GPGPU. <http://miaowgpu.org>.
- [2] 2016. AMD Accelerated Parallel Processing (APP) Software Development Kit. <http://developer.amd.com/sdks/amdappsdk/>
- [3] Paula Aguilera, Jungseob Lee, Amin Farmahini Farahani, Katherine Morrow, Michael J. Schulte, and Nam Sung Kim. 2014. Process variation-aware workload partitioning algorithms for GPUs supporting spatial-multitasking. In *Proc. of DATE*. 1–6.
- [4] Aatreyi Bal, Shamik Saha, Sanghamitra Roy, and Koushik Chakraborty. 2017. Revamping Timing Error Resilience To Tackle Choke Points at NTC systems. 1020–1025.
- [5] Prabal Basu, Hu Chen, Shamik Saha, Koushik Chakraborty, and Sanghamitra Roy. 2016. SwiftGPU: Fostering Energy Efficiency in a Near-Threshold GPU Through Tactical Performance Boost.
- [6] Leland Chang, David J. Frank, Robert K. Montoye, Steven J. Koester, Brian L. Ji, Paul W. Coteus, Robert H. Dennard, and Wilfried Haensch. 2010. Practical Strategies for Power-Efficient Computing Technologies. 98, 2 (2010), 215–236.
- [7] Design Compiler, RTL User, and Modeling Guide. 2001. Synopsys. Inc., see <http://www.synopsys.com> (2001).
- [8] Vivek De. 2015. Fine-Grain Power Management in Manycore Processor and System-on-Chip (SoC) Designs. In *Proc. of ICCAD*. 159–164.
- [9] Ronald G. Dreslinski, Michael Wiecekowsky, David Blaauw, Dennis Sylvester, and Trevor N. Mudge. 2010. Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits. *Proc. of the IEEE* 98, 2 (2010), 253–266.
- [10] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev R. Rao, Toan Pham, Conrad H. Ziesler, David Blaauw, Todd M. Austin, Krisztian Flautner, and Trevor N. Mudge. 2003. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proc. of MICRO*. 7–18.
- [11] Ulya R. Karpuzcu, Nam Sung Kim, and Josep Torrellas. 2013. Coping with Parametric Variation at Near-Threshold Voltages. *Proc. of MICRO* 33, 4 (2013), 6–14.
- [12] Ulya R. Karpuzcu, Krishna B. Kolluru, Nam Sung Kim, and Josep Torrellas. 2012. VARIUS-NTV: A microarchitectural model to capture the increased sensitivity of manycores to process variations at near-threshold voltages. 1–11.
- [13] S Karen Khatamifard, Michael Resch, Nam Sung Kim, and Ulya R Karpuzcu. 2016. VARIUS-TC: A modular architecture-level model of parametric variation for thin-channel switches. 654–661.
- [14] Shin-Ying Lee and Carole-Jean Wu. 2014. CAWS: criticality-aware warp scheduling for GPGPU workloads. In *PACT*. 175–186.
- [15] Jan Lucas, Sohan Lal, Michael Andersch, Mauricio Alvarez-Mesa, and Ben Jurlink. 2013. How a single chip causes massive power bills GPU-SimPow: A GPGPU power simulator. In *ISPASS*.
- [16] NanGate. [n. d.]. . http://www.nangate.com/?page_id=2328.
- [17] Abbas Rahimi, Luca Benini, and Rajesh K. Gupta. 2013. Hierarchically focused guardbanding: an adaptive approach to mitigate PVT variations and aging. In *Proc. of DATE*. 1695–1700.
- [18] S.R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. 2008. VARIUS: A Model of Process Variation and Resulting Timing Errors for Microarchitects. 21 (2008), 3–13.
- [19] Sangwon Seo, Ronald G. Dreslinski, Mark Woh, Yongjun Park, Chaitali Chakrabarti, Scott A. Mahlke, David Blaauw, and Trevor N. Mudge. 2012. Process variation in near-threshold wide SIMD architectures. In *Proc. of DAC*. 980–987.
- [20] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *PACT*.