# Run-Time Technique for Simultaneous Aging and Power Optimization in GPGPUs

Xiaoming Chen[1], Yu Wang[1], Yun Liang[2], Yuan Xie[3], Huazhong Yang[1]
[1]Department of EE, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China
[2]Center for Energy-efficient Computing and Applications, School of EECS, Peking University, Beijing, China
[3]Department of CSE, Pennsylvania State University, Pennsylvania 16802, USA
chenxm05@mails.tsinghua.edu.cn, yu-wang@tsinghua.edu.cn, ericlyun@pku.edu.cn, yuanxie@cse.psu.edu, yanghz@tsinghua.edu.cn

## ABSTRACT

High-performance general-purpose graphics processing units (GPGPUs) may suffer from serious power and negative bias temperature instability (NBTI) problems. In this paper, we propose a framework for run-time aging and power optimization. Our technique is based on the observation that many GPGPU applications achieve optimal performance with only a portion of cores due to either bandwidth saturation or shared resource contention. During run-time, given the dynamically tracked NBTI-induced threshold voltage shift and the problem size of GPGPU applications, our algorithm returns the optimal number of cores using detailed performance modeling. The unused cores are power-gated for power saving and NBTI recovery. Experiments show that our proposed technique achieves on average 34% reduction in NBTI-induced threshold voltage shift and 19% power reduction, while the average performance degradation is less than 1%.

## Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Single-instruction-stream, multiple-data-stream processors (SIMD)*

## General Terms

Design, Performance

## Keywords

General-purpose graphics processing unit (GPGPU); Negative bias temperature instability (NBTI); power

## 1. INTRODUCTION

Nowadays, general-purpose graphics processing units (GPGPUs) have been adopted as promising hardware accelerators for high performance computing. State-of-the-art GPGPUs have hundreds of cores, large register files, and high memory bandwidth. Collectively, these computing and memory resources provide massive thread-level parallelism. A wide range of applications have enjoyed the attractive computing capability offered by GPUs [1–4]. In recent years, there is also a rapid adoption of GPUs in mobile devices like smart phones. The system-on-chip solutions that integrate GPUs with other processing units are available in the mobile market, such as NVIDIA Tegra series with low-power GPUs [5] and Qualcomm's Snapdragon series with Adreno GPUs [6].

Power and energy have emerged as the first-order design constraints for modern computing systems, especially for those battery-operated mobile devices [7]. To support the massive parallelism, GPUs employ a large number of cores and register files. However, operating many hardware resources consumes power significantly. Modern high-performance GPUs usually consume more than 200W of power. Thus, the high computing capability of GPUs comes at high power and energy consumptions.

In addition, negative bias temperature instability (NBTI) has become a serious aging effect in nano-scale integrated circuits [8]. NBTI gradually increases the threshold voltage ($V_{th}$) of pMOS transistors when they are negatively biased, leading to delay degradation and potential timing violations. The degradation rate increases with the time during which a pMOS transistor is stressed. NBTI could cause about 25% performance degradation after 3 years at 45nm technology [8]. The degradation rate will be even larger in more advanced technologies.

In this paper, we propose a framework that simultaneously optimizes aging and power for GPGPUs at run-time . Our technique is based on the observation that many GPGPU applications achieve optimal performance with only a portion of cores due to either saturated memory bandwidth or shared resource contention [9, 10]. It is apparent that power and energy consumption can be saved if unused cores are power-gated during the execution of applications. For NVIDIA architectures, we choose to do power gating at the granularity of streaming multiprocessors (SMs) which include streaming processors (SPs), register files, and on-chip memory storages. More importantly, power-gating has been shown as an effective technique to mitigate NBTI-induced aging, since pMOS transistors undergo recovery during the period of power-gating. Thus, through power gating, we can optimize aging and power together. In order to achieve this goal, we propose an algorithm that determines the optimal number of SMs for GPGPU applications. The algorithm is assisted with detailed performance modeling and parameters obtained online including the application problem size and tracked NBTI-induced $V_{th}$ shift. The contributions of


*DAC '14* , June 01-05 2014, San Francisco, CA, USA

http://dx.doi.org/10.1145/2593069.2593208.

this paper are summarized as follows.

1) We propose a run-time framework for simultaneous aging and power optimization for GPGPUs. Our framework can dynamically power gate a portion of SMs based on their NBTI-induced degradation rates, problem size of applications, and the number of SMs for optimal performance.

2) We propose an efficient algorithm for deriving the optimal number of SMs. Our algorithm is guided by the detailed performance modeling which models the computation, memory, and thread scheduling. The algorithm is executed at run-time and can deal with different problem sizes.

3) We demonstrate that our approach can mitigate NBTI-induced $V_{th}$ degradation by 34% on average and reduce the power consumption of SMs by 19% on average, while the average performance degradation is less than 1%.

# 2. RELATED WORK

**NBTI Optimization for GPUs:** There is a body of NBTI optimization techniques for general digital circuits [11–15]. However, till now, there are very few studies on NBTI optimizations for GPUs [16, 17]. Recently, Rahimi *et al.* developed a compiler-directed technique to uniformly distribute the stress of instructions across computing units to mitigate the NTBI-induced degradation [16]. However, their compiler technique is tightly coupled with the very long instruction word-based AMD GPU architecture. In addition, their static workload characterization without run-time information (such as problem size) may lead to wrong optimization decisions. In contrast, our performance estimation technique models generic design features of modern GPGPUs including computation cycles, memory stalls, warp scheduling, and cache effects. More importantly, our technique makes optimization decisions based on the problem size, which is only available at run-time. In the experiments, we will demonstrate that different problem sizes can lead to different optimization solutions and results. NBTI optimization for register files in GPUs was also studied [17].

**Performance Modeling and Power Optimization for GPUs:** To efficiently exploit GPUs' computing capability, it is very important to understand the performance bottlenecks through performance modeling. Hong and Kim developed an analytical model for generic GPGPU architectures [18]. The model has been further extended with power modeling [19]. Other models based on work flow graph [20], quantitative models [21], and memory access patterns [22] were also proposed in the literature. Energy consumption becomes another important concern for GPGPUs. Researchers have explored different power optimization techniques including pre-fetching [23], efficient memory- and instruction-level designs [24, 25]. Power gating is an efficient technique for power and energy savings. Power gating can be used for leakage reduction in GPUs while maintaining the target frame rate of display [26]. Leakage reduction in GPU's caches by power gating was studied [27]. A power gating technique taking the opportunity of idle periods during execution to power gate idle execution units (such as integer units and floating-point units) has been proposed [28]. Our power gating technique works at the granularity of SMs, so it is easier to implement and only uses a small number of sleep transistors. Furthermore, our method chooses to power gate the SMs with higher NBTI-induced degradation so that we can mitigate the NBTI effects and save power simultaneously.

# 3. BACKGROUND

**NBTI Modeling:** NBTI becomes a serious aging mechanism in modern integrated circuits. We adopt a widely used NBTI model [29], which is based on the reaction-diffusion mechanism [30]. When a pMOS transistor is negatively biased ($V_{gs} = -V_{dd}$), some Si-H bonds in the interface are broken, the H atoms diffuse away and the Si atoms act as interface traps and consequently, $V_{th}$ increases. This is called stress phase and can be expressed as:

$$\Delta V_{th,stress} = \left(K_v\sqrt{t_{stress}} + \sqrt[2n]{\Delta V_{th0}}\right)^{2n} \quad (1)$$

When the pMOS transistor is turned off ($V_{gs} = 0$), the H atoms diffuse back and anneal the existing traps, so the $V_{th}$ shift can be partly recovered:

$$\Delta V_{th,recov} = \Delta V_{th,stress}\left(1 - \frac{2\xi_1 t_e + \sqrt{\xi_2 C t_{recov}}}{2t_{ox} + \sqrt{C t_{total}}}\right) \quad (2)$$

The definition of the parameters used in the model can be found in [29]. The NBTI-induced degradation rate strongly depends on $V_{gs}$, $V_{th}$ and the duty cycle ($\frac{t_{stress}}{t_{stress}+t_{recov}}$).

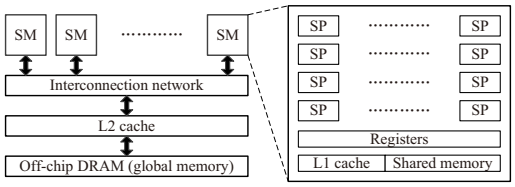



Figure 1: GPGPU architecture.

**GPGPU Architecture:** We take NVIDIA GPUs as an example to illustrate the GPGPU architecture as shown in Fig. 1. It represents a generic design of modern GPU architecture. A GPU consists of several streaming multiprocessors (SMs), and each SM is composed of streaming processors (SPs), L1 cache, shared memory, and special functional units. The access latency to off-chip global memory is much longer than the access latency to on-chip storages. A program running on GPUs is called a kernel. GPUs execute kernels in a single instruction multiple thread (SIMT) manner. GPU threads are organized into warps (32 threads on NVIDIA GPUs and 64 threads on AMD GPUs), which are further organized into thread blocks. Each thread block is assigned to one SM for execution, and one SM can handle multiple concurrent thread blocks. Warps are the fundamental scheduling units and GPUs use context switches between warps to hide the long off-chip memory latency.

Most kernels are designed to be parametric to handle different problem sizes. In this paper, we define problem size as the input size of GPU applications. The problem size, known at run-time, determines the number of invoked threads. For example, for matrix-based applications, the matrix dimensions usually determine the number of threads; for vector-based applications, the vector lengths determine the number of threads. Hence, compared with static analysis, our run-time technique can optimize power and aging according to different problem sizes.

# 4. MOTIVATION

Ideally, we expect the long latency to off-chip memory can be completely hidden by the massively parallel architecture of GPUs. However, in reality, many GPU application-

s are memory-bound, or have a very low computation-to-memory ratio. For such kernels, their best performance can be achieved with only a portion of SMs due to bandwidth saturation or cache/memory contention. In Fig. 2, we show the ratio of compute utilization to bandwidth utilization for different applications. The experiments are performed on NVIDIA GTX580 and the results are collected using NVIDIA Visual Profiler. As shown, for most of the applications, their compute utilization is lower than the memory bandwidth utilization. We further evaluated the performance of LU and RNG using GPGPU-Sim [31], under different number of SMs. The results are shown in Fig. 3. For LU, the best performance is achieved with 15 SMs. However, its performance saturates with about 12 SMs and further increasing SMs does not improve the performance much. For RNG, the best performance is achieved with 12 SMs and increasing the number of SMs results in high cache and memory contention and thus degrades the performance.

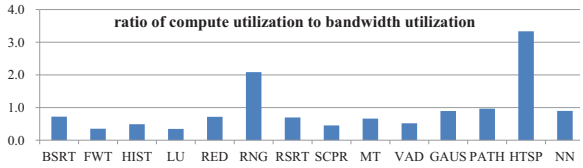
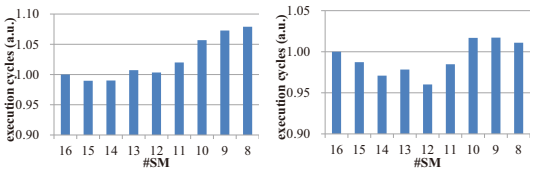



Figure 2: Ratio of compute utilization to bandwidth utilization, measured by NVIDIA Visual Profiler.




(a) benchmark LU (b) benchmark RNG

Figure 3: Execution cycles, using different number of SMs.

The above observations suggest that we can use a portion of SMs to maintain the same performance or achieve better performance compared with using all the SMs, and power gate the rest of SMs for power and energy saving. Meanwhile, the power-gated SMs will undergo recovery so the NBTI-induced degradation can also be mitigated. Power gating is widely used for power saving in computing systems. Modern power gating techniques have quite small penalties. For example, Kaul *et al.* [32] proposed a power gating technique with only 1 cycle wake-up latency.

# 5. RUN-TIME AGING AND POWER OPTIMIZATION

In this section, we first present the framework of our run-time aging and power optimization technique. We then describe the details of our performance modeling and optimization algorithm.

## 5.1 Run-time Framework

Fig. 4 presents the framework. The inputs to the framework are the CUDA kernel, the application problem size, and the aging rate of each SM. Then, based on these, our framework determines the number of SMs for optimal performance, and also the SMs for power gating. We assume

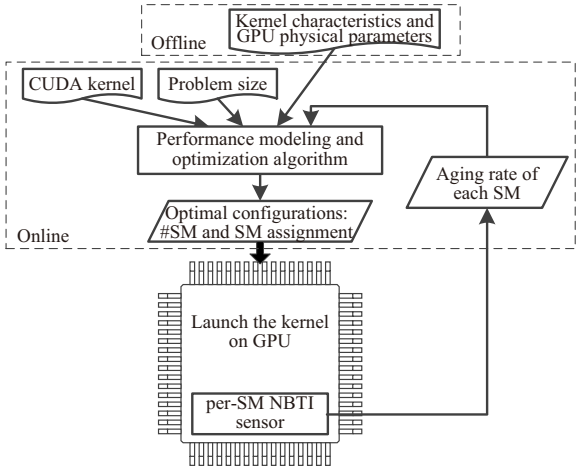



Figure 4: Framework of the proposed technique.

the GPU is equipped with per-SM NBTI sensor [33] which can track the NBTI-induced $V_{th}$ shift.

To minimize the run-time overhead, we analyze the kernel offline to collect the kernel characteristics that do not change with the problem size. These characteristics include memory latencies, computation and memory cycles. We build a parameterized performance model that models the computation, memory cycles at warp-level, warp scheduling, and cache effects. The parameters of our performance model are the problem size and the number of SMs. During runtime, we feed the problem size into the performance model and determine the optimal number of SMs for performance using the model. Then, based on the dynamically tracked degradation rate of each SM, we choose to power gate the SMs with high degradation rate and execute the kernel on the rest of the SMs. During the execution of a kernel, the active/power-gated SMs are fixed and we only change the active/power-gated SMs before launching the kernel.

## 5.2 Modeling and Optimization Algorithm

The goal of the algorithm is to fast and accurately estimate the performance for GPUs. We build our performance model based on the generic analytical model, which is inspired by the concepts of memory warp and memory warp parallelism (MWP) [18]. A warp that is waiting for memory data is called a memory warp. $MWP$ represents the maximum number of memory warps that can be handled concurrently on one SM.

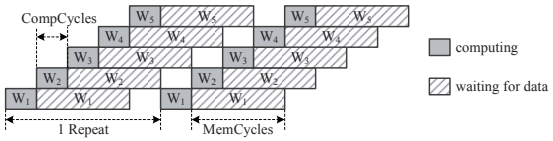



Figure 5: Illustration of warp scheduling (the memory bandwidth is not saturated).

To accurately model the performance, we have to consider warp scheduling. Fig. 5 illustrates a simple case of warp scheduling. Two parameters are defined: $CompCycles$ for the average cycles of each computing period of a warp, and $MemCycles$ for the average cycles of each memory access period of a warp. The memory-to-computation ratio is defined as $R_{MtoC} = \frac{MemCycles}{CompCycles}$. There are at most $R_{MtoC}$

memory warps on one SM at the same moment. It can also be easily deduced that if there are $R_{MtoC}+1$ or more warps running on one SM, the memory latencies can be completely hidden. In this case, the number of total execution cycles is mainly determined by the computing cycles:

$$TotalCycles \approx MemCycles + Repeat \times WarpsPerSM \times CompCycles \quad (3)$$

where $Repeat$ is the number of execution repeats of each warp and can be simply calculated by $Repeat = \frac{TotalWorkload}{TotalWarps}$. $TotalWorkload$ depends on the problem size, which is known at run-time. $TotalWarps$ depends on the total number of threads, which in turn depends on the problem size. $WarpsPerSM = \frac{TotalWarps}{\#SM}$.

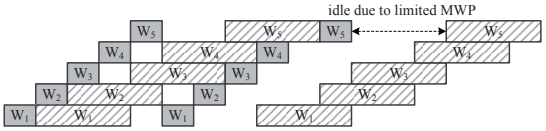



Figure 6: Illustration of warp switches (the memory bandwidth is saturated). $MWP = 2$ in this example.

However, the above case is under the assumption that the memory bandwidth can support $R_{MtoC}$ concurrent memory warps on one SM, i.e. $MWP \geq R_{MtoC}$. If ($MWP < R_{MtoC}$), as shown in Fig. 6, the memory warps will be delayed due to the limited $MWP$ (only $MWP$ memory warps can be served concurrently on one SM). In this case, the number of total execution cycles is mainly determined by memory cycles:

$$TotalCycles \approx Repeat \times \frac{WarpsPerSM}{MWP} \times MemCycles \quad (4)$$

### 5.2.1 Parameters in the Model

$CompCycles$ can be easily obtained by analyzing the parallel thread execution (PTX, an assembly-like language in CUDA) code of a kernel. $MemCycles$ is calculated by:

$$MemCycles = GlobalMemLatency \times CacheMissRate + CacheLatency \times (1 - CacheMissRate) \quad (5)$$

where $GlobalMemLatency$ and $CacheLatency$ are the global memory latency and the cache latency. We measure $GlobalMemLatency$ and $CacheLatency$ on the real GPUs using micro benchmarks. In our experiments, we also set GPGPU-Sim simulator to use the same latencies. We do not build cache models to estimate $CacheMissRate$ as it may introduce substantial run-time overhead. Instead, we empirically measure the cache miss rates for all the benchmarks. The measurement is performed on the target GPU using NVIDIA Visual Profiler.

Finally, $MWP$ is constrained by the global memory bandwidth and can be calculated as follows:

$$MWP = \frac{Bandwidth \times MemCycles}{\#SM \times f \times LoadBytesPerWarp} \quad (6)$$

where $Bandwidth$ is the global memory bandwidth and $f$ is the core frequency. $LoadBytesPerWarp$ means the average memory size which is actually accessed from the global memory (excluding the memory requests served from caches) in each memory period, and it is calculated by:

$$LoadBytesPerWarp = LoadBytesPerThread \times 32 \times CacheMissRate \quad (7)$$

$LoadBytesPerThread$ is the average memory size accessed by each thread in one memory period, and it is obtained by analyzing the PTX code.

All the parameters used in the model, except those are related to the problem size and the number of SMs, can be collected offline through either kernel analysis or measurement.

### 5.2.2 Optimization Algorithm

In this section we describe the proposed online power and aging optimization algorithm. Algorithm 1 presents the details. As previously discussed, the inputs to Algorithm 1, such as $Bandwidth$, $f$, $GlobalMemLatency$, $CacheLatency$, $CacheMissRate$, etc., can be collected through offline analysis. But, the problem size, the number of threads, and aging rates of SMs are known at run-time.

Algorithm 1 first attempts to find an optimal number of SMs (denoted by $optSM$) to minimize the execution cycles. This is done by comparing the performance under different SM numbers using the performance model. If $optSM$ turns out to be the maximum SM number, we will relax the performance constraint by an allowed range $\delta$. In other words, we accept the $optSM$, where its performance is slightly worse than using the maximum number of SMs. After that, we collect the NBTI-induced $V_{th}$ shift of all SMs through the NBTI sensors. Finally, we assign the $optSM$ SMs with the lowest degradation rate to execute the kernels and power gate the rest of SMs. $\delta$ is determined empirically. In this work, we use $\delta = 0.08$.

**Algorithm 1:** Finding the optimal GPU configurations

**Input**: the kernel (PTX code), the problem size, number of threads, and GPU parameters: $Bandwidth$, $f$, $GlobalMemLatency$, $CacheLatency$, $CacheMissRate$
**Output**: optimal #SM ($optSM$), and SM assignment

```
Evaluate the execution cycles using maximum #SM, denoted by C_max;
for k=(maximum #SM) to (minimum allowed #SM) do
    Evaluate the execution cycles using k SMs, denoted by C_k;
    if C_k ≤ C_max then
        optSM = k;
if optSM ==maximum #SM then
    for k=(maximum #SM) to (minimum allowed #SM) do
        if C_k ≤ (1 + δ)C_max then
            optSM = k;
Read NBTI-induced per-SM V_th shift from the NBTI sensors;
Assign the optSM SMs with the lowest degradation rates to
execute the kernel, other SMs are power-gated;
```

# 6. SIMULATION RESULTS

## 6.1 Experimental Methodologies

The framework of Fig. 4 is emulated by a software simulator written in C++. We use GPGPU-Sim [31], which is a cycle-accurate GPGPU simulator, with an integrated power estimation tool GPUWattch [34], to execute kernels and collect statistics of kernels. GPGPU-Sim uses the configurations of NVIDIA GTX580, as shown in Table 1. We assume that the minimum allowed #SM is 4, as using too few SMs is not practical for parallel computing. HotSpot [35] is used to estimate the temperature of all SMs. An NBTI estimator using the model shown in Section 3 is used to emulate the NBTI sensors by calculating the NBTI-induced degradation rate of each SM, based on the obtained kernel execution time and temperature. We use Eq. (1) to calculate the $V_{th}$ shift when an SM is working and use Eq. (2) when an SM

is power-gated. The obtained $\Delta V_{th}$ can be regarded as the maximum possible degradation rate of all transistors in one SM. We use the following technology parameters: nominal $V_{dd0} = 1.0$V and nominal $V_{th0} = 0.3$V.

Table 1: Configurations of NVIDIA GeForce GTX580.

| | |
|---|---|
| #SM | 16 |
| #SP per SM | 32 |
| processor clock ($f_0$) | 1544 MHz |
| global memory bandwidth | 192.4 GB/s |
| DRAM clock | 2004 MHz |
| L1 cache per SM | 16 KB |
| shared memory per SM | 48 KB |
| L2 cache | 768 KB |

The following 14 kernels are used to evaluate the proposed technique: vector add (VAD), matrix transposition (MT), scalar product (SCPR), fast Walsh transform (FWT), reduction (RED), bitonic sort (BSRT), and histogram (HIST) from CUDA software development kit (SDK) samples, sparse LU factorization (LU) from [2], Gaussian elimination (GAUS), path finder (PATH), hotspot (HTSP), and nearest neighbor (NN) from the Rodinia benchmark suite [36], and two real-world kernels: uniform random number generator (RNG) and radix sort (RSRT). Each kernel is executed for 100 rounds. In the following, we compare our run-time technique (OURS) with the default setting using maximum number of SMs (DEFAULT) in terms of performance, power, energy, and NBTI-induced $V_{th}$ shift.

## 6.2 Experimental Results

Table 2 shows the optimal number of SMs reported from the optimization algorithm and the online analysis time of our technique. The online optimization algorithm runs quite fast (microsecond-level). Each kernel takes about millisecond or more to execute, so the online analysis overhead is very small.

Table 2: Results of the optimization algorithm.

| benchmark | optimal #SM | online analysis time ($\mu$s) |
|---|---|---|
| BSRT | 7 | 3.1 |
| FWT | 15 | 2.5 |
| HIST | 15 | 2.8 |
| LU | 12 | 2.2 |
| RED | 15 | 2.2 |
| RNG | 11 | 2.1 |
| RSRT | 8 | 2.5 |
| SCPR | 14 | 2.7 |
| MT | 15 | 2.6 |
| VAD | 9 | 2.3 |
| GAUS | 13 | 3.0 |
| PATH | 15 | 2.8 |
| HTSP | 15 | 2.8 |
| NN | 15 | 2.5 |

The results of the kernel execution time are shown in Fig. 7. For RNG and RSRT, the performance is improved by reducing the number of SMs. For some benchmarks, such as FWT, SCPR, GAUS, they incur slight performance degradation. This is because we allow a small performance degradation when searching the optimal number of SMs (see Algorithm 1). On average, the performance is only degraded by 0.6% as shown by the "AVG" bar in Fig. 7. Hence, our technique can maintain good performance for a wide range of kernels even though a portion of SMs are powered gated. Fig. 7 also shows that the time cost of our run-time technique is very small, which can be ignored.

The power gating overhead (wake-up time) is not included in the kernel execution time. Since our power gating is employed at the SM level, and we do not change the active/power-gated SMs during the execution of a kernel, compared with the kernel execution time, the overhead is so small that can be ignored.

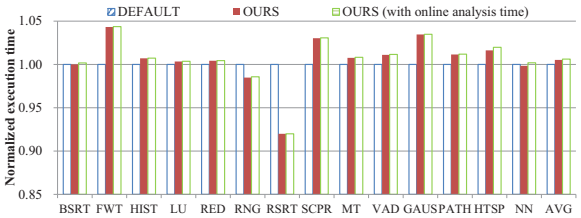



Figure 7: Results of the normalized execution time.

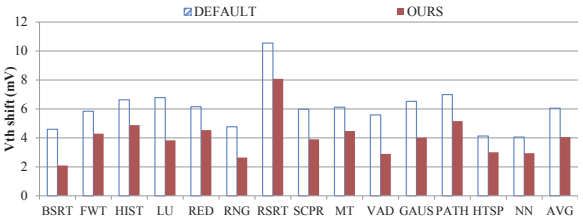
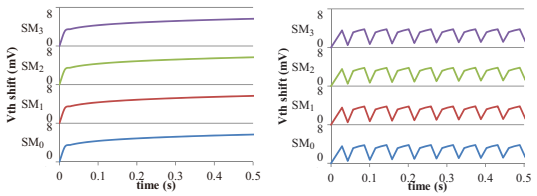



Figure 8: Results of the NBTI-induced $V_{th}$ degradation.

Fig. 8 shows the results of NBTI-induced $V_{th}$ degradation (the maximum $V_{th}$ shift of all SMs). For all the benchmarks, the NBTI-induced $V_{th}$ degradation is reduced by on average 34% when applying power gating. The reduction rate in $\Delta V_{th}$ ranges from 26% to 55%, for different benchmarks. NBTI-induced degradation can be greatly mitigated by power gating, as shown in Fig. 9, which is evaluated on benchmark LU. The $V_{th}$ shift over time monotonously increases to about 6.78mV due to the default approach. When applying power gating, each SM undergoes periodic stress-recovery phases when the kernel is repeatedly executed, so the NBTI-induced degradation is mitigated by about 44%, leading to a final $V_{th}$ shift of only 3.83mV.




(a) DEFAULT. (b) OURS.

Figure 9: NBTI-induced $V_{th}$ degradation of 4 SMs, for LU.

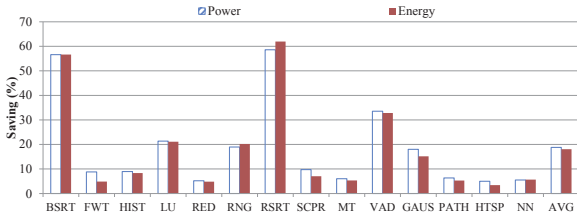



Figure 10: Results of power and energy reduction.

The power and energy saving rates for all SMs are shown in Fig. 10. These results are collected using GPUWattch [34]. Our technique achieves up to 57% (average 19%) power saving compared with the default approach. Similar trend has

been observed for energy saving. In addition to the power saving of SMs, our technique also helps to save the power of other components in the GPU, because the workloads of these components are reduced due to fewer SMs. Because of space limit, we do not list the detailed power savings of other components.

Our run-time technique optimizes power and aging according to different problem sizes. Here, we demonstrate this point using benchmarks PATH and VAD. The details of the benchmarks and the dependence of invoked threads with the problem size can be found in their C code. The results are shown in Tables 3 and 4, respectively. Clearly, different problem sizes lead to different optimal number of SMs and power/energy savings. Pure static or compiler techniques may result in sub-optimal SM number and power/energy degradation.

Table 3: Results of PATH, under different input sizes.

| input size | optimal #SM | normalized execution time | NBTI mitigation | power saving | energy saving |
|---|---|---|---|---|---|
| 1000 | 4 | 0.804 | 56.9% | 81.9% | 85.4% |
| 2000 | 7 | 1.016 | 54.9% | 63.8% | 63.2% |
| 3000 | 10 | 0.988 | 49.1% | 39.2% | 40.0% |
| 4000 | 12 | 1.046 | 44.7% | 28.5% | 25.2% |
| 5000 | 15 | 1.003 | 27.3% | 4.8% | 4.5% |

Table 4: Results of VAD, under different input sizes.

| input size | optimal #SM | normalized execution time | NBTI mitigation | power saving | energy saving |
|---|---|---|---|---|---|
| 5000 | 5 | 0.992 | 54.0% | 64.2% | 64.4% |
| 10000 | 7 | 1.027 | 53.7% | 44.6% | 43.1% |
| 30000 | 9 | 1.015 | 48.2% | 35.0% | 34.0% |
| 50000 | 9 | 1.016 | 48.3% | 36.5% | 35.5% |

# 7. CONCLUSIONS

Although modern GPGPUs provide attractive computing capability in many fields, the high power consumption has become a key challenge. On the other hand, NBTI severely affects the reliability of nano-scale integrated circuits. This paper proposes a run-time technique for simultaneous NBTI and power optimization for GPGPUs. Based on the observation that kernels which are memory-intensive or bandwidth-bound do not need all SMs due to either bandwidth saturation or shared resource contention, some SMs can be power-gated for NBTI mitigation and power saving. Our technique obtains the optimal number of SMs at run-time based on the dynamically tracked NBTI-induced threshold voltage shift and problem sizes of GPGPU applications. Experiments show that our technique reduces the power consumption of SMs by 19% and mitigate NBTI-induced degradation by 34%, while the performance is almost not affected.

# 8. ACKNOWLEDGEMENTS

This work was supported by 973 project 2013CB329000, National Science and Technology Major Project (2013ZX03003013-003), National Natural Science Foundation of China (61373026, 61261160501, 61028006), National Science Foundation (NSF 0905365/1218867), and Tsinghua University Initiative Scientific Research Program.

# 9. REFERENCES

[1] J.D. Owens and et al. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[2] L. Ren and et al. Sparse LU factorization for parallel circuit simulation on GPU. In *DAC*, pages 1125–1130, 2012.

[3] V. Bertacco and et al. On the use of GP-GPUs for accelerating compute-intensive EDA applications. In *DATE*, pages 1357–1366, 2013.

[4] Y. Liang and et al. Real-time implementation and performance optimization of 3D sound localization on GPUs. In *DATE*, pages 832–835, 2012.

[5] NVIDIA Tegra. http://www.nvidia.com/object/tegra.html.

[6] Qualcomm Inc. http://www.qualcomm.com/snapdragon.

[7] S.W. Keckler and et al. GPUs and the Future of Parallel Computing. *Micro, IEEE*, 31(5):7–17, 2011.

[8] S. P. Park and et al. Reliability Implications of Bias-Temperature Instability in Digital ICs. *Design Test, IEEE*, 26(6):8–17, 2009.

[9] O. Kayiran and et al. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *PACT*, pages 157–166, 2013.

[10] J.T. Adriaens and et al. The case for GPGPU spatial multitasking. In *HPCA*, pages 1–12, 2012.

[11] X. Chen and et al. Assessment of Circuit Optimization Techniques under NBTI. *Design Test, IEEE*, PP(99):1–1, 2013.

[12] F. Firouzi and et al. NBTI mitigation by optimized NOP assignment and insertion. In *DATE*, pages 218 –223, 2012.

[13] J. Abella and et al. Penelope: The NBTI-Aware Processor. In *Micro*, pages 85 –96, 2007.

[14] A. Calimera and et al. NBTI-aware power gating for concurrent leakage and aging optimization. In *ISLPED*, pages 127–132, 2009.

[15] X. Fu and et al. NBTI tolerant microarchitecture design in the presence of process variation. In *MICRO*, pages 399–410, 2008.

[16] A. Rahimi and et al. Aging-aware compiler-directed VLIW assignment for GPGPU architectures. In *DAC*, pages 1–6, 2013.

[17] M. Namaki-Shoushtari and et al. ARGO: Aging-aware GPGPU Register File Allocation. In *CODES+ISSS*, 2013.

[18] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*, pages 152–163, 2009.

[19] S. Hong and H. Kim. An integrated GPU power and performance model. In *ISCA*, pages 280–289, 2010.

[20] S. S. Baghsorkhi and et al. An adaptive performance modeling tool for GPU architectures. In *PPoPP*, pages 105–114, 2010.

[21] Y. Zhang and J.D. Owens. A quantitative performance analysis model for GPU architectures. In *HPCA*, pages 382 –393, 2011.

[22] Y. Kim and A. Shrivastava. CuMAPz: A Tool to Analyze Memory Access Patterns in CUDA. In *DAC*, pages 128–133, 2011.

[23] A. Sethia and et al. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In *PACT*, pages 73–82, 2013.

[24] S.Z. Gilani and et al. Power-efficient computing for compute-intensive GPGPU applications. In *HPCA*, pages 330–341, 2013.

[25] M. Abdel-Majeed and M. Annavaram. Warped register file: A power efficient register file for GPGPUs. In *HPCA*, pages 412–423, 2013.

[26] P.-H. Wang and et al. Power gating strategies on GPUs. *ACM Trans. Archit. Code Optim.*, 8(3):13:1–13:25, October 2011.

[27] Y. Wang and et al. Run-time power-gating in caches of GPUs for leakage energy savings. In *DATE*, pages 300–303, 2012.

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

[29] S. Bhardwaj and et al. Predictive Modeling of the NBTI Effect for Reliable Design. In *CICC*, pages 189 –192, 2006.

[30] M.A. Alam. A critical examination of the mechanics of dynamic NBTI for PMOSFETs. In *IEDM*, pages 14.4.1 – 14.4.4, 2003.

[31] A. Bakhoda and et al. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, pages 163–174, 2009.

[32] H. Kaul and et al. A 300 mV 494GOPS/W Reconfigurable Dual-Supply 4-Way SIMD Vector Processing Accelerator in 45 nm CMOS. *JSSC*, 45(1):95–102, Jan 2010.

[33] P Singh and et al. Dynamic NBTI management using a 45nm multi-degradation sensor. In *CICC*, pages 1–4, 2010.

[34] J. Leng and et al. GPUWattch: enabling energy optimizations in GPGPUs. In *ISCA*, pages 487–498, 2013.

[35] W. Huang and et al. Differentiating the roles of IR measurement and simulation for power and temperature-aware design. In *ISPASS*, pages 1–10, 2009.

[36] S. Che and et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.