Safe Limits on Voltage Reduction Efficiency in GPUs: a Direct Measurement Approach

Jingwen Leng[†⊤], Alper Buyuktosunoglu[⊤], Ramon Bertran[⊤], Pradip Bose[⊤], Vijay Janapa Reddi[†]
[⊤]*IBM T.J. Watson Research Center,* [†]*The University of Texas at Austin*
[⊤]{jleng, alperb, rbertra, pbose}@us.ibm.com, [†]jingwen@utexas.edu, [†]vj@ece.utexas.edu

ABSTRACT

Energy efficiency of GPU architectures has emerged as an important aspect of computer system design. In this paper, we explore the energy benefits of reducing the GPU chip's voltage to the safe limit, i.e. V_{min} point. We perform such a study on several commercial off-the-shelf GPU cards. We find that there exists about 20% voltage guardband on those GPUs spanning two architectural generations, which, if "eliminated" completely, can result in up to 25% energy savings on one of the studied GPU cards. The exact improvement magnitude depends on the program's available guardband, because our measurement results unveil a program dependent V_{min} behavior across the studied programs. We make fundamental observations about the program-dependent V_{min} behavior. We experimentally determine that the voltage noise has a larger impact on V_{min} compared to the process and temperature variation, and the activities during the kernel execution cause large voltage droops. From these findings, we show how to use a kernel's microarchitectural performance counters to predict its V_{min} value accurately. The average and maximum prediction errors are 0.5% and 3%, respectively. The accurate V_{min} prediction opens up new possibilities of a cross-layer dynamic guardbanding scheme for GPUs, in which software predicts and manages the voltage guardband, while the functional correctness is ensured by a hardware safety net mechanism.

1. INTRODUCTION

General-purpose GPU (GPGPU) architectures are already important elements of mainstream computing. Although the GPU provides enormous computation capability, it comes with the cost of consuming much more power than its counterpart CPU. A high-performance GPU card has a peak power consumption between 250 W and 300 W, whereas many commodity CPUs plateau around 130 W. In the context of latest GPU architectures, we have seen a significant emphasis on improving performance per watt metric. For example, NVIDIA claims that the Kepler architecture achieves 3× the performance per watt of their previous-generation Fermi [1], and the latest Maxwell architecture achieves 2× performance per watt improvement over the Kepler [2].

State-of-the-art GPU power-saving efforts strongly reflect and follow the CPU power optimizations trend. Typical optimizations include clock and power gating, and dynamic voltage and frequency scaling (DVFS). Although there has been increasing focus on applying these traditional techniques to GPUs [1, 3], we need to focus on new(er) opportunities for power optimization. In this paper, we demonstrate the energy-efficiency benefits of pushing the GPU architecture's voltage to its operating limit. To combat the worst-case process, temperature and voltage variation (noise), traditional design methodology relies on voltage guardbanding. The voltage guardband relative to the nominal voltage is predicted to grow due to increased variations as technology scales [4]. The industry standard practice of designing for the worst-case condition leads to energy inefficiency because the chip could have operated at a lower supply voltage in the nominal case [5, 6].

We explore the energy benefits of reducing the GPU chip's voltage to the V_{min} point at a fixed frequency, using NVIDIA's off-the-shelf GPU cards spanning two architectural generations (Fermi and Kepler). At the V_{min} point, the program executes correctly but fails if the voltage is reduced any further. Our measured results demonstrate two fundamental observations. First, the V_{min} is program dependent. Second, the guardband between the nominal voltage and V_{min} is large (9% - 18%) on a GTX 680 card. The guardband protects against the process, temperature and voltage variation. We study each variation's impact on V_{min} and observe that the voltage noise has the largest impact. Because voltage noise depends on program characteristics [4], it also matches the program-dependent V_{min} observation.

Since voltage noise is the main determinant of V_{min}, we must understand its key characteristics. Specifically, we need to determine the root cause of the largest voltage droop and when exactly that happens in the context of CPU-initiated GPU execution. We profile each

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.
MICRO-48, December 05-09, 2015, Waikiki, HI, USA
© 2015 ACM. ISBN 978-1-4503-4034-2/15/12 ...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830811

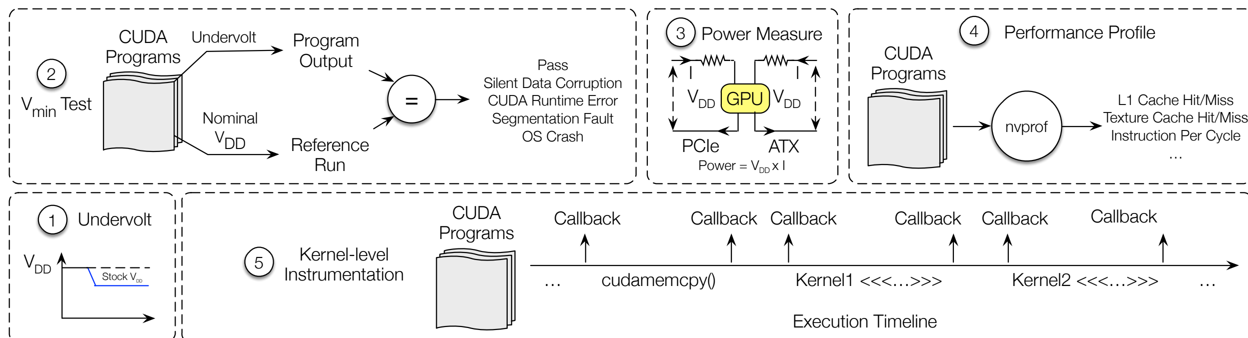


Figure 1: Overview of the experimental setup. 1. Undervolt: we use an overclocking tool to control the GPU chip's voltage. 2. V_{min} test: we measure the V_{min} point of each program by gradually undervolting the GPU and check the program output correctness. 3. Power measurement: we use custom power-sensing circuitry to measure GPU power. 4. Performance profile: we use `nvprof` to access performance counters. 5. Kernel-level instrumentation: we use the callbacks before and after each kernel invocation to measure the V_{min} and power of each kernel.

program's performance characteristics and measure its power consumption to study the interaction among performance, power and V_{min}. These measured program-driven metrics and inter-relationships form the foundation of facts from which we make observations about the key characteristics of voltage noise in GPUs. With these observations, we further show that we can predict each kernel's V_{min} value with at most 3% error, using microarchitectural events. This enables a promising opportunity for maximizing energy efficiency: the software predicts and operates at the safe voltage limit, and a lightweight "safety net" hardware mechanism ensures reliable operation for the rare case of a program that exceeds the pre-set prediction error margin. We study the potential in a conceptual exemplary design through measurement on a real GPU card.

In summary, we make the following key contributions:

- We measure the voltage guardband between V_{min} and the nominal supply voltage on several off-the-shelf GPU cards spanning two architectural generations. The results reveal that a relatively large amount of voltage guardband exists on all studied cards.
- We characterize the process, temperature and voltage variation impact on the V_{min}. We experimentally determine that the di/dt droop part of voltage noise during the kernel execution has the highest impact and causes the program-dependent V_{min} behavior.
- We perform a quantitative study of the relationship between the program's performance characteristics and V_{min}. We study how to use a kernel's performance counters to predict its V_{min} accurately, and we demonstrate the large energy-savings potential of a conceptual exemplary design using the V_{min} prediction through measurement on a real GPU card.

We organize the paper as follows. Section 2 describes our experimental setup. Section 3 presents the V_{min} measurement results and analysis. Section 4 analyzes the root cause of the large V_{min} variability. Section 5 studies the interaction between program characteristics and V_{min}, and how to predict V_{min} values accurately using microarchitectural performance counters. Section 6 demonstrates a possible optimization opportunity derived from our experimental insights. Section 7 discusses related work, and Section 8 concludes the paper.

2. EXPERIMENTAL SETUP

In this section, we describe our experimental setup shown in Figure 1. The central piece is the fine-grained voltage guardband exploration test, i.e. the V_{min} test (① and ②). We also measure the program's power consumption (③) and profile its performance characteristics (④ and ⑤) to study the interaction of a program's V_{min} and its performance and power.

2.1 Voltage Guardband Exploration

We explore the voltage guardband on several off-the-shelf GPU cards and a large set of representative programs via V_{min} measurements. We describe the details of V_{min} test, studied GPU cards and programs.

V_{min} Test We measure the voltage guardband for each individual program by measuring the V_{min} point, an operating point at which the program executes correctly but fails when the voltage is reduced any further. The V_{min} test includes two parts: ① and ② in Figure 1. We decrease the GPU's operating voltage from its stock setting. The stock setting of the GTX 680 card is 1.09 V at 1.1 GHz. We use the MSI Afterburner [7] to control the GPU chip's voltage at a fixed frequency. The granularity for controlling the voltage is 12 mV. We do not modify the memory frequency and voltage.

With each step of 12 mV undervolting, we measure each program's V_{min} point. At each step, we run the program and check program correctness by validating its output against a reference run at the nominal operating point. Each run is considered to be "pass" if i) for integer programs, the output is identical to the reference run, or ii) for floating-point programs, the output is within 10^{-2}% of the reference run. We consider a voltage level as a working voltage if the program passes 1,000 times. V_{min} is the minimal working voltage. Note that we also study the error behavior for each program

GPU	GTX 480	GTX 580	GTX 680	GTX 780
Architecture	Fermi		Kepler	
Core Counts	15	16	8	12
Core Clock (MHz)	700	875	1100	1100
Memory Clock (MHz)	1846	2004	3004	3004
Register Per Core (KB)	128	128	256	256
L1 Cache (KB)	48/16 (Configurable)			
L2 Cache (KB)	768	768	512	1536
Read-Only Data Cache (KB)	N/A	N/A	N/A	48
Memory Controllers	6	6	4	6
TDP (W)	250	250	195	250
Technology (nm)	40		28	

Table 1: GPU cards' microarchitectural specifications.

operating below its V_{min} point, but we run it 100 times for each voltage level due to long experimental time.

Measurement Noise Control We control temperature and background activities on the GPU that may impact or skew the measured V_{min} results. For temperature control, we adjust the fan speed to stabilize the temperature at 40 °C when the program starts execution. This guarantees similar measurement temperature for all studied programs. We observe only a small temperature change during program execution given its short execution time. We report all programs' V_{min} value measured at 40 °C unless it is explicit. We nullify irrelevant system activities during the experiment, specifically the graphic activities, by installing another GPU card dedicated to graphics tasks. We do not control the CPU activities, because they do not affect the V_{min} on the stand-alone GPU card (see Section 4.2).

GPU Cards We perform V_{min} measurements on several off-the-shelf GPU cards. The studied GPUs in this work span two architectural generations: Fermi (GTX 480 and 580) and Kepler (GTX 680 and 780). Table 1 lists their key microarchitectural specifications [1, 8]. Note that "core" refers to SM in Fermi, and SMX in Kepler. Five different GTX 780 cards are used to verify our experiments' reproducibility and to study if there is an observable difference related to process variation.

CUDA Programs We study a set of 57 programs from the CUDA SDK [9], Rodinia [10] and Lonestar [11] benchmark suites. These programs have both diverse performance and distinctive V_{min} characteristics, which help us make insightful observations of their interaction.

2.2 Power Measurement

We measure each program's power consumption to study the relationship between the program's power behavior and V_{min}, and quantify the energy-saving benefits of operating at the V_{min} point. The part ③ in Figure 1 shows our power measurement setup. The GPU card consumes power from the PCIe connection and the ATX power supply. We measure the power consumption of both sources and add them up to get the GPU power. We insert a 25 mOhm shunt resistor at each connection to measure the instantaneous current and voltage and calculate the power consumption. We use the data acquisition unit NI DAQ 6133 [12] to record the data at a rate of 2 million samples per second. This power measurement setup is independent of the GPU card and lets us switch cards and measure their power consumption. Note that the measured power consumption is at board-level, which includes the GPU chip, DRAM and peripheral circuits such as voltage regulator.

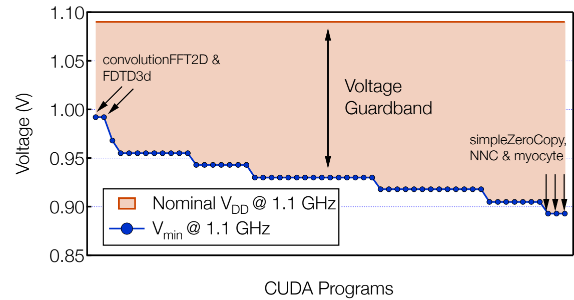


Figure 2: V_{min} measurements for 57 programs on the GTX 680.

2.3 Profiling and Instrumentation

We use the NVIDIA profiler `nvprof` [13] to access GPU's performance counters. The counters profiled include various cache misses and functional unit utilization. We collect them at the kernel level; the run-to-run variation of these counters reported by `nvprof` is within 1%.

In the V_{min} test, we rely on kernel-level instrumentation to control the voltage during each kernel's execution to measure its V_{min}. The CUPTI (CUDA profiling tools interface) library [14] provides instrumentation capability by registering the custom callbacks before and after each kernel and runtime API call. We implement our own callbacks to control each kernel's voltage.

3. VOLTAGE GUARDBAND ANALYSIS

We study how much voltage guardband different programs require to execute correctly. More specifically, we explore and quantify how far we can push the guardband for program performance and energy improvement without impacting the correctness level assumed by an application developer. We quantify the guardband opportunity between the nominal voltage and each program's safe limit (i.e. V_{min} point) on four GPU cards spanning two architectural generations. We also study each program's error behavior when it runs with the voltage beyond its safe limit. We try to understand whether more aggressive optimization is feasible by lowering the voltage further and allowing errors to happen.

3.1 V_{min} Measurement Results

We perform V_{min} measurement using the 57 representative programs on four different GPU cards spanning two architectural generations (Fermi and Kepler architecture). The comprehensive measurement helps us to study and quantify the program-specific voltage guardband behavior because the voltage guardband that exists for a program is the difference between the card's nominal voltage and the program's V_{min} value.

We first study the voltage guardband of the 57 programs on a GTX 680 with Kepler architecture. The voltage guardband for each program is the margin between the nominal voltage and its V_{min} point. The volt-

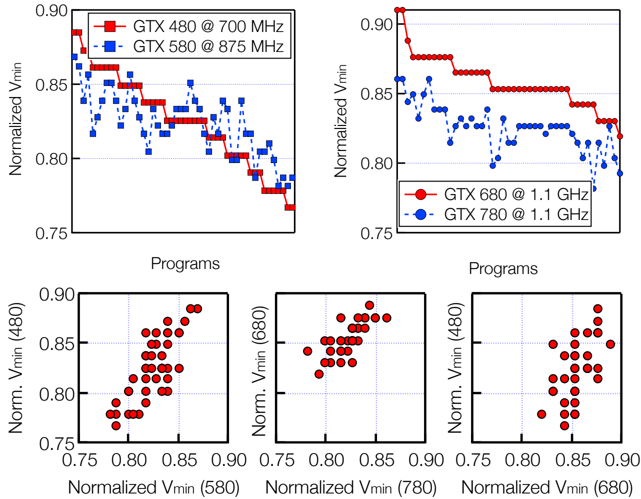


Figure 3: Comparing V_{min} across different GPU cards.

age stock setting of studied GTX 680 is 1.09 V at a frequency of 1.1 GHz. Figure 2 plots the measured V_{min} of these programs. We make two fundamental observations from the measured results as follows.

First, a relatively large amount of voltage guardband exists for all studied programs. The V_{min} value for these programs varies from 0.89 V to 0.99 V. Considering that the nominal voltage of the GTX 680 card is 1.09 V, we can calculate that a relatively large percent of the voltage (9.2% to 18.3%) can be reduced without affecting the program's functional correctness. The magnitude is similar to the measured voltage guardband on an Intel Core 2 Duo processor reported in prior work [4].

Second, Figure 2 shows a large variability in the studied programs' V_{min} values, which means that a program's V_{min} value strongly depends on its characteristics. The difference between the highest V_{min} value (0.99 V) and the lowest V_{min} value (0.89 V) is 0.1 V for the studied programs. Two programs (FDTD3d and convolutionFFT2D) have the highest V_{min} value of 0.99 V, and three programs (simpleZeroCopy, NNC and myocyte) have the lowest V_{min} value of 0.89 V, as labelled in Figure 2. Most of the programs have a V_{min} value of about 0.93 V.

We further find that these two observations, i.e. the relatively large voltage guardband and the program-dependent V_{min} behavior, exist on different GPU architectures. In total, we perform V_{min} measurements on four GPU cards: GTX 480 and GTX 580 (Fermi architecture) and GTX 680 and GTX 780 (Kepler architecture). Their specifications are described in Table 1. Because each card has a different nominal voltage, we normalize each card's V_{min} to its nominal voltage for comparison. Figure 3 plots the normalized V_{min} on four cards and their comparison. The range of voltage guardband is similar across these cards: 11.5% - 23.3% on GTX 480, 11.6% - 20.3% on GTX 580, 9.2% - 18.3% on GTX 680 and 14% - 22.5% on GTX 780.

We also observe that the program-dependent V_{min} behavior exists on all four cards: different programs have different V_{min} values. Moreover, V_{min} values on cards with same architecture are more correlated than V_{min} values on cards with different architectures. In the last three plots of Figure 3, V_{min} values on GTX 480 and GTX 580, and on GTX 680 and GTX 780, are more correlated than V_{min} values on GTX 480 and GTX 680. We can explain the lower correlation of V_{min} between two different architectures by observing program-dependent V_{min} behavior. A program's characteristics may change when it is running on a different architecture, which results in different V_{min} behavior.

3.2 Error Distribution Below V_{min}

We experimentally measure each program's failure probability when operating below its V_{min} point. This helps us to determine whether more aggressive optimization of operating below the V_{min} point is feasible. We describe the details of error events and how we detect their occurrence. There are four main types of error events: i) silent data corruption; ii) CUDA runtime errors, GPU driver fault or segmentation fault; iii) operating system crash; and iv) infinitely long execution.

Silent data corruption (SDC) [15] refers to when a program finishes execution without any warning but produces an incorrect end result. We detect it by comparing the test output from the undervolt run against a *golden* output from a reference (fault-free) run. We compare the integer and floating-point output separately, as described earlier. CUDA runtime errors refer to the erroneous execution of a program that fails at runtime (e.g., memory and stream management). Such errors are explicitly reported by the CUDA runtime system. Driver fault occurs when the GPU driver code executed by the CPU loses communication with the GPU. Often this results in a screen freeze followed by an automatic hard reset of the GPU card. These two types of errors can be detected by inspecting the standard error output. The harshest error is the OS crash, after which a manual reboot is required. We stop the voltage reduction experiment once an OS crash happens. Some programs, such as BFS and DMR, operate on graph data structures and use iterative algorithms to converge to the final output. An error may cause it to deviate from convergence, and its execution time becomes longer or infinitely long. Due to its rare occurrence, we manually detect the error and do not study its probability.

We gradually increase the undervolt percent level. We run the program 100 times and record the outcome at each level. Figure 4 shows the undervolting experiment results for six representative programs. In each subplot, the x-axis shows the undervolt percent, i.e., percent reduction from the nominal voltage. The undervolt percent at the leftmost x-axis point corresponds to the program's V_{min} point. For example, the V_{min} of convolutionFFT2D is 0.99 V, which corresponds to 9% undervolting, marked as "V_{min} Point" in Figure 4. The rightmost x-axis point is marked as "OS Safe Point," beyond which the program can cause an OS crash. The y-axis plots the distribution of 100 runs that result in a pass, SDC, CUDA runtime error or segmentation fault. For example, at the 11.3% undervolt level, convolutionFFT2D

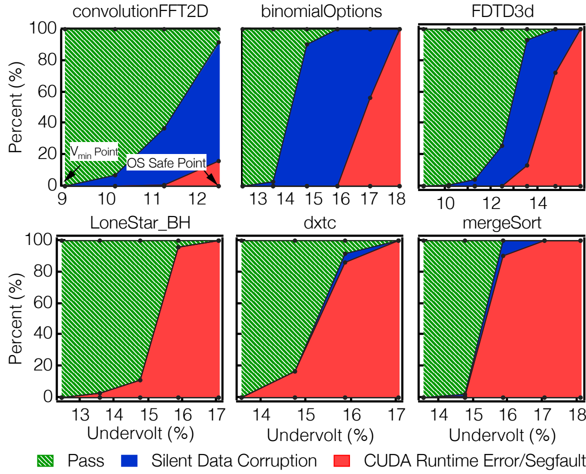


Figure 4: The distribution of runs that result in pass, silent data corruption, CUDA runtime error or segmentation fault when increasing undervolt percent.

has 63 runs that result in a pass, 36 runs that result in SDC and 1 run that results in a runtime error.

We summarize four key observations from this experiment. First, all 100 runs at the V_{min} point result in pass, validating our V_{min} measurement results. Second, an additional 4-5% undervolt percent below the V_{min} point usually causes an OS crash. In other words, the OS safe point is 4-5% lower than the V_{min} point.

Third, we observe two program categories with different failure behaviors. The top three programs in Figure 4 have significant SDC incidence during undervolting, whereas the bottom three suffer primarily from crash failures (runtime error or segmentation fault). In other words, the first category is "SDC-prone," and the second is "crash prone." In our study, there are 37 and 20 programs for each category, respectively. We inspect their source codes to diagnose the possible cause of their behavioral differences. We find that the most obvious difference between the two categories is the intra-loop control dependency (i.e., conditional branches and embedded function calls). Programs with large such dependency are prone to crash errors. Instead, programs with minimal such dependency and fixed loop counts have a larger SDC incidence before the onset of crash errors during undervolting. This observation matches the common intuition that control-intensive codes have higher crash probability, because of the higher probability of illegal memory address references.

Fourth, the program failure probability increases as the undervolt level increases, because lower voltage translates to less timing margin and higher error probability. Moreover, we observe an avalanche error effect when voltage is below a certain value. For example, the error probability of `FDTD3d` increases from 3% to 90% when the undervolt percent increases from 10% to 12%. This avalanche error effect may be caused by the common design practice that most paths are skewed toward the critical timing specification of the processor [16]. When the voltage goes below V_{min}, most paths would have a timing violation, causing an avalanche error effect.

In summary, more aggressive optimization by lowering the voltage further is possible because the program can still execute correctly at times below the V_{min} point. However, the challenge is to detect the execution error such as SDC. Moreover, the potential improvement might be marginal given the avalanche error effect. Thus, we focus only on pushing the guardband for energy improvement without impacting the correctness level assumed by an application developer.

4. ROOT CAUSE OF V_{min} VARIABILITY

In this section, we analyze the *root cause* of the large V_{min} variability so that we can perform voltage guardband optimizations. There are two fundamental questions regarding the root cause. First, which variation causes the large V_{min} variability? The voltage margin mainly protects against the process, voltage and temperature (PVT) variation and aging. Each variation has a different implication for optimization. Second, which program activity pattern causes the program-dependent V_{min} behavior? CUDA programs have complicated activity patterns, such as the CUDA runtime execution, initial kernel launch, kernel-to-kernel transition and the kernel execution. We must identify the causative pattern in order to determine the optimization efforts.

4.1 Variation Impact Analysis

We first determine which type of variation causes the program V_{min} variability, from the candidates of process, temperature and voltage variation and aging. After identifying the voltage variation (noise) as the cause of the program V_{min} variability, we further analyze its dominant component (IR drop or di/dt droop).

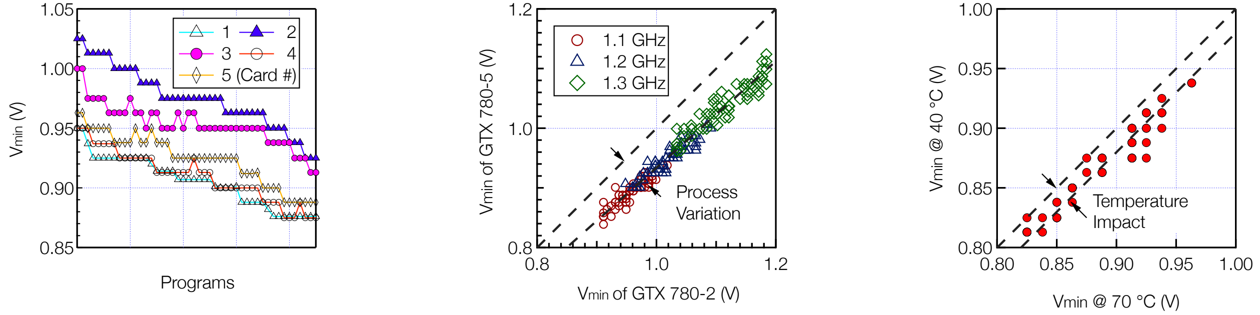
4.1.1 Process, Temperature and Aging

In our work, we directly measure the process and temperature variation impact on V_{min}, and use the method of exclusion to infer the impact of voltage noise.

Process Process variation causes variable device threshold and speed, and thus different V_{min} values. It results from imperfect lithography [17, 18] and dopants diffusion [17, 19]. Process variation can be further divided into inter-die variation, which means that the same device on a chip has different features from a different die, and intra-die variation, which means the device feature varies between locations on the same die [17, 20].

We use five GTX 780 cards to study the impact of process variation. Figure 5a plots the V_{min} of our studied programs, which are measured at 40 °C. Program names are omitted because of space constraints. They are sorted in the descending order of Card 2's V_{min}, the highest among all cards. The largest observable difference of V_{min} among the five cards is that the V_{min} values of all programs measured on one card shift up or down by a relatively constant value compared to the values on the other card. The largest V_{min} difference of the same programs between two cards is about 0.07 V.

We also measure Card 2 and Card 5's V_{min} at three frequencies: 1.1, 1.2 and 1.3 GHz. Each marker in Fig-



(a) Process variation impact on V_{min} of five GTX 780 cards.

(b) Process variation impact on V_{min} of two cards and three frequencies.

(c) Temperature variation (40 °C and 70 °C) impact on V_{min}.

Figure 5: (a) and (b) study process variation impact on V_{min}, and (c) studies temperature variation impact on V_{min}.

ure 5b plots the V_{min} of the same program running on two cards at a frequency point. If there were no variation at all, the V_{min} of the same program would be identical on both cards, which would result in markers lying on the dashed 45-degree diagonal line in Figure 5b. But, in fact, the V_{min} on the GTX 780-2 card is consistently higher than the values on the GTX 780-5 card by a relatively constant offset, which increases slightly as the frequency increases. We also find that the magnitude of the V_{min} difference between two cards is not identical for all programs in Figure 5a and Figure 5b. For example, some programs have a greater V_{min} difference between two cards than other programs.

Given the same experimental conditions for other factors, we attribute the cause of V_{min} difference on different cards to the process variation. Both inter-die and intra-die variation have the systematic component and the random component. The former can explain the constant difference of V_{min} on two cards, and the latter can explain the small variability of the difference possibly due to the critical timing path shift. Note that additional measurements on more GPU cards are required to draw a more statistically sound conclusion.

Temperature We measure V_{min} values at two temperatures (40 °C and 70 °C) to study the temperature's impact. We observe 40 °C as the nominal temperature and 70 °C as the highest temperature when running an OpenGL stress test at the highest frequency and lowest fan speed. Thus, the temperature 70 °C is a very unlikely worst-case scenario for regular CUDA programs. Figure 5c shows the results. We observe a similar impact on V_{min} as the process variation but with a smaller magnitude: V_{min} at 70 °C is consistently about 0.02 V higher than the values at 40 °C. The temperature variation impact on V_{min} is similar to the process variation.

Aging In our study, we cannot directly measure the impact of aging on V_{min}. However, it is unlikely that the aging effect caused such large V_{min} variability among the programs. The published measurement results on a recent IBM z System shows that the circuit speed degrades only 1-2% in the long term [21]. Moreover, all our experiments are done within a few months, thus the impact of aging would be even smaller. Hence, the magnitude of the observed V_{min} variability between programs and cards cannot be explained by aging effects.

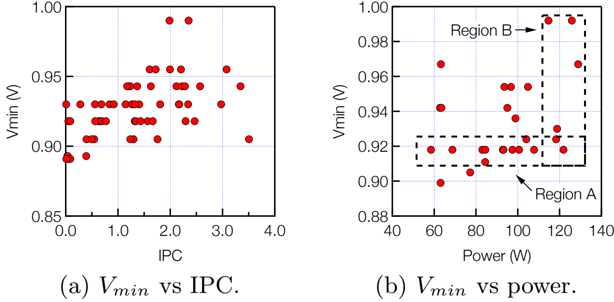
In summary, we observe that both process and temperature have a relatively uniform impact on the V_{min} across all programs. Neither can explain the large variability of measured V_{min} across programs. Because the voltage guardband protects against process, temperature and voltage variation (noise) and aging, voltage noise remains the only possible cause, per method of exclusion. In other words, the program with a higher V_{min} value is due to a greater magnitude of voltage noise. This also matches with established knowledge that voltage noise results from the interaction between program activity and the processor power delivery network [22], and thus V_{min} depends on the program characteristics. Note that measuring the voltage noise directly through the oscilloscope or on-chip sensors [4, 23] can directly prove our observation. We leave it as future work.

We also observe that voltage noise has a larger impact on the voltage guardband compared to process and temperature variation. The measured V_{min} ranges from 0.89 to 0.99 V on the same card with the same temperature, which indicates the magnitude of voltage noise of 0.1 V. This is larger than the measured process variation impact of 0.07 V and temperature impact of 0.02 V. In the rest of the paper, we focus on voltage noise analysis on the GTX 680 card, unless explicitly mentioned.

4.1.2 Voltage Noise: IR Drop vs di/dt Droop

After identifying the voltage noise as the causative variation for the program V_{min} variability, we also study its dominant component in GPU architectures. There are two components in voltage noise: IR drop and di/dt droop as shown in Equation 1. These two components have distinctive properties. The IR drop component is determined purely by the instantaneous current draw, whereas the di/dt droop component is determined by the current draw's increasing rate. Each component has a different implication for determining optimization efforts due to their distinctive properties, and as such we must understand which component is dominant.

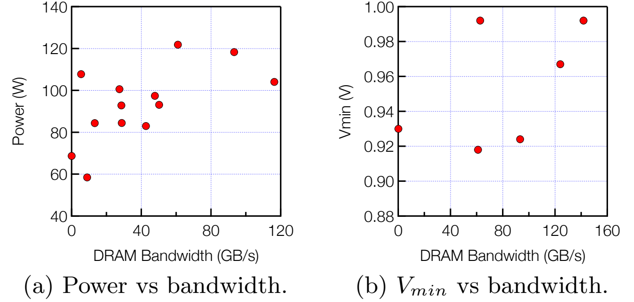
$$V_{actual} = V_{DD} - I \times R - L \times \frac{di}{dt} \tag{1}$$



(a) V_{min} vs IPC.

(b) V_{min} vs power.

Figure 6: V_{min} vs IPC and measured power.



(a) Power vs bandwidth.

(b) V_{min} vs bandwidth.

Figure 7: (a) Power vs DRAM bandwidth for programs in Region A that have similar V_{min} of 0.92 V. (b) V_{min} vs DRAM bandwidth for programs in Region B that have similar power consumption around 120 watts.

We leverage the IR drop's property to test the hypothesis that it is the dominant component. If the hypothesis holds true, a program with high V_{min} would have a high power consumption, too. We first inspect the relationship between the V_{min} and the maximum IPC of the program because the IPC is a good indicator of power consumption [24]. Figure 6a plots the results. The highest possible IPC is 4 because the Kepler architecture can issue up to four warps per cycle. Figure 6a shows that programs with higher IPC do not necessarily have higher V_{min}. Instead, the programs with an IPC of about 2 have the highest V_{min}. Thus, there is no evident correlation between V_{min} and IPC.

We also measure the GPU's power consumption to directly inspect the relationship between V_{min} and power. Figure 6b shows the result. There is no evident correlation between the V_{min} and the GPU card power consumption either. However, the measured power is at board level, which includes both GPU chip and DRAM power (Section 2.2). Only GPU chip power consumption would impact V_{min}, and DRAM power may disturb the correlation that may have existed. Thus, Figure 6b is not enough to draw the conclusion that the V_{min} has no correlation with the GPU power consumption.

To overcome this measurement limitation, we profile and collect each program's DRAM bandwidth utilization to approximate its power consumption [25]. We inspect the programs in Region A and Region B in Figure 6b separately. Programs in Region A all have a low V_{min} value of about 0.92 V but have a wide range of measured power. If the hypothesis that IR drop is the dominant component holds true, those programs would have similar chip power consumption. The high measured power would be due to high DRAM power. Figure 7a plots the relationship between the measured power and DRAM bandwidth for those programs. There are programs with very low DRAM bandwidth but with high power. This again contradicts the hypothesis.

Programs in Region B all have a high measured power consumption but a wide range of V_{min}. If the hypothesis holds true, the programs with high V_{min} would have a high chip power consumption and low DRAM power. Figure 7b plots V_{min} against DRAM bandwidth for those programs. Programs with higher V_{min} also have higher bandwidth. This is the counter example of the hypothesis. Thus, the di/dt droop instead of the IR drop is the dominant component of the voltage noise.

4.2 Program Activity Impact Analysis

Our previous analysis shows that the di/dt droop causes the large V_{min} variability among programs. In this subsection, we analyze which program activity pattern causes such di/dt droop by categorizing the program activities and measuring each category's V_{min}. We categorize the activities to four types: i) CUDA runtime activity, ii) inter-kernel activity, which causes the repeated current ramp up and down in GPU; iii) initial-kernel launch activity, which causes the current ramp up at the kernel invocation; and iv) intra-kernel activity, which causes the current fluctuation due to microarchitectural events. Identifying the relevant activity lets us mitigate the droop by suppressing the activity, or predict the V_{min} value using the activity metrics.

CUDA Runtime We find that many programs spend a significant amount of time on CUDA runtime functions such as transferring data back and forth between CPU and GPU. But it is unclear what their impact on V_{min} is. Thus, we study several CUDA runtime functions that commonly exist in CUDA programs. They are cudaMalloc, cudaMemset, cudaMemcpy, cudaSetupArgument and cudaConfigureCall. The first two allocate and set the specified size of global memory space in a GPU to a certain value, and cudaMemcpy transfers a trunk of data between the CPU and the GPU. The last two are common before a kernel starts to execute, and they basically push the kernel invocation arguments to registers [26].

To verify if any of these runtime functions is the source of large di/dt droop, we measure each function's V_{min}. We use the CUPTI library (see Section 2) to register a callback, which controls the voltage, before every runtime function invocation. We measure the five commonly seen functions' V_{min} by only performing undervolting during the tested function execution. Figure 8 plots the results. The V_{min} of these functions is 0.89 V, which is 0.1 V lower than the highest measured V_{min} of all programs (FDTD3d). Thus, the activity of runtime functions is not the source of the large di/dt droop.

Inter-Kernel Activity The inter-kernel activity is

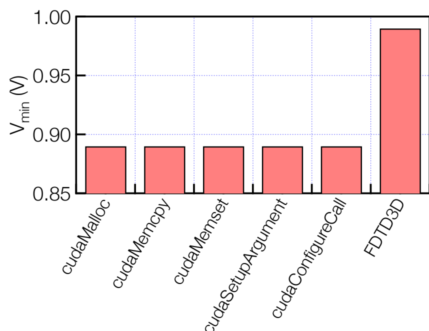


Figure 8: CUDA runtime functions V_{min} measurement results.

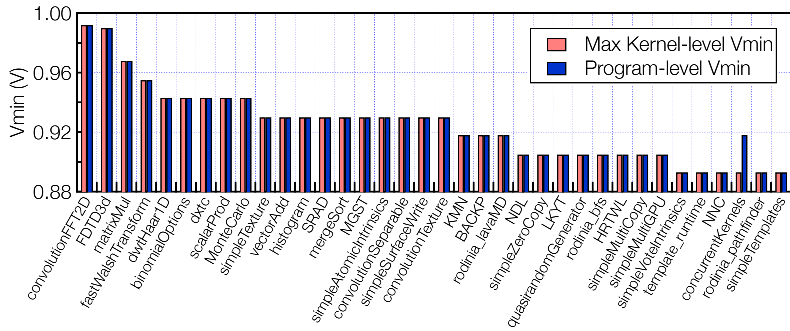


Figure 9: Kernel-level V_{min} matches the program level, meaning that voltage noise is from individual kernel execution.

another potential source of the large di/dt droop. In the GPU programming model, a CPU launches a set of kernels sequentially. The consecutive launch of kernels can result in repetitive current fluctuations (ramp-up/ramp-down) and therefore can cause large di/dt droops. Note also that the transition from CUDA runtime to kernel execution may result in similar current fluctuations.

We measure each kernel's V_{min} to verify if inter-kernel or runtime-kernel transition activity is the main source of the di/dt droop. The methodology is similar to the CUDA runtime V_{min} measurement. In addition, we insert a 1-second delay before undervolting GPU's voltage during the kernel execution. Because the voltage regulator can handle the current variation slower than millisecond [27], the 1-second delay is long enough to nullify the impact of current fluctuations caused by inter-kernel activity and runtime-kernel transitions. We define the measured V_{min} as *kernel-level* V_{min} to distinguish from the *program-level* V_{min}, which is measured by performing undervolt for the entire program execution.

Figure 9 compares the program-level V_{min} and its maximum kernel-level V_{min}. If the inter-kernel activity or runtime-kernel transition caused a large di/dt droop, the measured kernel-level V_{min} would be much smaller than the program-level. We observe that the kernel-level V_{min} of all programs except concurrentKernel match the program-level. Thus, neither of the two types of activities causes large di/dt droops. The mismatch for concurrentKernel can be attributed to the side-effect of kernel-level V_{min} measurement due to the use of CUPTI library, that is, serialized execution of all kernels. That program originally has multiple kernels which run concurrently. The serialization side effect reduces the activities on GPU, and therefore also reduces its V_{min}.

Initial-Kernel Activity The individual kernel activity remains the dominating source of the large di/dt droop because we have shown that neither the CUDA runtime function nor the inter-kernel activity is the main source. The individual kernel activity can be further divided into initial-kernel and intra-kernel activity. When a kernel is launched, all cores' states change from the idle state to the active state in a short period, similar to the effect of barrier synchronization point, which may cause a large di/dt droop [28].

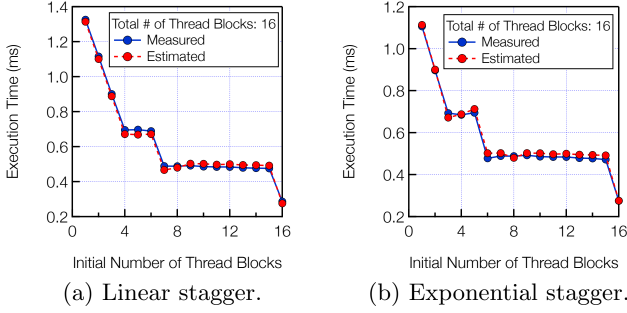
To verify if the initial-kernel activity is the source of large di/dt droop, we design an experiment to stagger the activation of cores and inspect if the V_{min} decreases after the staggering. Prior work in CPUs has shown this can mitigate synchronization-induced voltage noise [28]. If the initial-kernel activity was the source of large di/dt droop, we would observe a decrease of the kernel's V_{min}.

We develop a mechanism for thread-block execution stagger. In GPUs, a thread block is the smallest unit that is scheduled to a core [29]. The rate of core activation can be reduced by staggering the activity of issuing thread blocks. The rate of stagger can be either linear or exponential; we implement both in our experiment.

Because of space limitations, we describe briefly how the stagger execution works. We use a global variable in the CUDA global memory space to record the number of thread blocks doing the actual work. Before starting the actual computation, each thread block checks if the present number of active thread blocks exceeds the allowed number. If yes, the thread block waits until the other thread blocks are complete. Otherwise, it starts execution immediately. The control of the number of active thread blocks is the key to thread block staggering. There are two implementation-related details. First, we declare the global variable as *volatile* to bypass the non-coherent L1 cache [30] in both Fermi and Kepler [1, 8]. Second, we use a spin lock to control multiple thread blocks' concurrent accesses to this global variable, which we implement with the atomic compare-and-swap instruction atomicCAS in CUDA.

We validate our implementation of linear and exponential stagger execution using a simple program vectorAdd. Figure 10 compares the measured and estimated execution time for linear and exponential stagger with 16 thread blocks. The value of the x-axis indicates the number of thread blocks in the beginning. A value of 2 means two thread blocks can be scheduled in the beginning. After that, the number of thread blocks that can be scheduled increases linearly or exponentially. Because each thread block processes the same size of data, the execution time of each thread block is similar. The overall execution time can be easily estimated. The match between estimated time and the measured time validates our implementation of the stagger execution.

We apply the stagger execution to the top 10 pro-



(a) Linear stagger. (b) Exponential stagger.

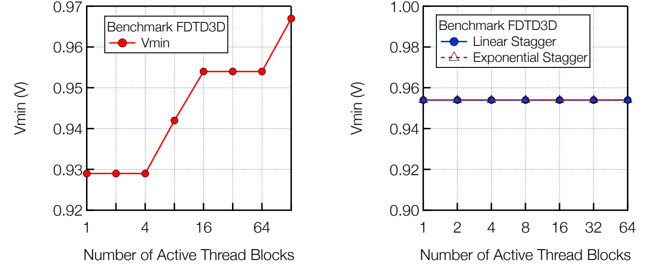
Figure 10: Correctness validation for implementation of linear and exponential stagger execution of thread blocks during the initial kernel launch.

grams with the highest V_{min} and inspect if the stagger execution can alter their V_{min} values. We measure their V_{min} under different scenarios: i) limiting the number of thread blocks throughout the execution, and ii) staggering the thread blocks' execution linearly and exponentially. We observe the same results among the programs and report only FDTD3D's results. Figure 11a shows V_{min} increases as the number of active thread blocks increases, which is expected because more active cores can build up the di/dt droop [4, 23]. However, in Figure 11b, we observe that neither linear nor exponential staggering affects the V_{min}. Thus, the initial kernel activity does not cause large di/dt droops, possibly because chip designers may already have built in some staggering mechanism, or the latency of ramping up all the cores is too long so that the overall di/dt droop effect is small.

Intra-Kernel Activity Using the method of exclusion, we infer that the intra-kernel activity causes large di/dt droops because we have excluded all other sources (CUDA runtime, inter-kernel and initial-kernel activity). The intra-kernel activities can be microarchitectural events such as cache miss in the light of prior simulation-based work [31]. They cause pipeline stalls, and the pipeline suddenly becomes active after a stall, resulting in a current surge and a large di/dt droop.

We study how a kernel's input affects its V_{min}. Specifically, a kernel can be launched multiple times with different input data in a CUDA program. We measure each launch's V_{min}. Figure 12 shows the box plot for the V_{min} of kernels with multiple invocations. Most kernels' V_{min} values only vary about 1%. Kernels in convolutionFFT2D have the largest variation, ranging from 0.92 to 0.99 V. We inspect its source code and find that invocations with much lower V_{min} do not contribute to the program output. Thus, our methodology of checking program output cannot measure these launches' V_{min} values. The V_{min} variation of other launches that actually contribute to the program output is within 0.01 V.

We also observe that the V_{min} of some kernels varies up to 0.04 V with different input. Their performance characteristics (counters) also have larger variability than other programs. In summary, most kernels have small V_{min} variation regarding the input, and some kernels' V_{min} is more sensitive to the input because of the sensitivity of their performance characteristics to the input.



(a) Limiting the number of active thread blocks. (b) Linear and exponential stagger of thread blocks.

Figure 11: V_{min} increases as the number of thread blocks increases but stays constant regardless of linear or exponential staggering. This indicates that activities within the kernel cause a large di/dt droop.

5. PROGRAM-CENTRIC V_{min} ANALYSIS AND V_{min} PREDICTION

Our analysis indicates that the di/dt droop caused by the microarchitectural events during the kernel execution causes the large V_{min} variability. This finding motivates us to explore the feasibility of using microarchitectural performance counters to predict the programs' V_{min} value. We study several different approaches of using microarchitectural performance counters to predict the V_{min} value, which enables the possibility of software-driven voltage margin reduction scheme.

5.1 Program Category and V_{min}

We first study whether a set of programs with similar performance characteristics have similar V_{min} values, and whether they have distinctive V_{min} values compared to another set of programs with different characteristics. We categorize CUDA programs into four types by their performance characteristics: memory bound, whose execution time is bound by the memory bandwidth; compute bound, whose execution time is bound by the core's computational capabilities; latency bound, which do not have enough threads and thus have very low utilization of both compute units and main memory bandwidth; and balanced, which achieves high utilization on both the compute units and memory bandwidth.

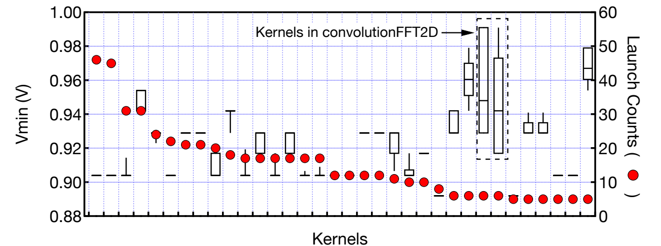


Figure 12: The V_{min} variation of each kernel with multiple launches. The kernels in convolutionFFT2D have large variation because the outputs of some launches do not contribute to the final program output.

Figure 13: V_{min} for the different program types.

Figure 13 shows the V_{min} for the different program types. Compute- and latency-bound programs tend to have lower V_{min} (i.e., smaller droop). Both memory-bound and balanced programs have diverse V_{min} values, and two of the memory-bound programs have the highest V_{min} among all programs (i.e., larger droop). Prior work for CPUs has shown that two conditions are needed for large voltage droops to occur: pipeline stalls, and synchronized stalls among multiple cores. They can explain why the memory-bound programs can have large droops: they have memory-related pipeline stalls, and these kinds of stalls tend to synchronize because of memory subsystem contention. Although pipeline stalls also exist in latency-bound programs, they are not likely aligned because of the lack of shared resources contention. Compute-bound programs either have stable power draw or unsynchronized stalls. In summary, we observe that the program's performance characteristics correlate well with its V_{min} value. This encourages us to study how to predict the V_{min} accurately.

5.2 Top-Down Approach for V_{min} Prediction

In this subsection, we study two top-down approaches for V_{min} prediction that let us to quickly evaluate the feasibility and the accuracy of the performance counter based V_{min} prediction model. The top-down approach does not require any prior knowledge about V_{min}. Instead, it uses all possible performance counters from a large number of programs and automatically constructs the prediction model, allowing a fast implementation and evaluation. We consider two methods: linear regression and neural network. The former produces a linear model, and the latter produces a nonlinear model.

We collect 28 performance counters to construct the V_{min} prediction model. These are IPC, utilization level of different functional units (arithmetic logic, floating-point and load-store units), single- and double-precision floating-point operations per second (FLOPS) and hit and miss rates of various caches (instruction, data and texture cache). These counters are input features for both the linear regression and neural network approaches.

We calculate three error metrics for determining the prediction model's efficacy. We calculate the root mean square error (RMSE) as the metric to indicate average prediction accuracy. We also calculate the maximum overprediction error, which means the predicted undervolting level is higher than the actual value. The actual

(a) Linear regression

(b) Neural network.

Figure 14: The predicted undervolt vs measured undervolt using microarchitectural events. (a) uses linear regression, and (b) uses neural network.

value corresponds to the program's V_{min} point, and undervolting more than that level can cause the program to fail. The third error metric we calculate is the maximum underprediction error, which means the predicted undervolting level is lower than the actual value. In this case, extra guardband is wasted because the program could have run at a lower voltage level.

For the linear regression, we collect the 28 performance counters from 557 kernel launches. Thus, these data form a 557×28 input matrix for the linear regression. Figure 14a shows the predicted value against the actual undervolt level. Although linear regression has an RMSE value of only 2.2%, it overpredicts many kernels' undervolting levels by 7%, where the maximum is 10%. This requires additional guardband to tolerate. Moreover, the maximum underprediction error is 10%: the predicted undervolt level is 5% but the actual level is 15%. This results in a 10% guardband wastage.

We use the same 557×28 matrix to train a neural network to predict the undervolting level. The trained neural network has one hidden layer. Its input layer has 28 neurons, and the hidden layer has 10 neurons. Figure 14b shows prediction accuracy, with an RMSE value of 0.5%. Not only does the neural network outperform the linear regression approach in the average error, but the maximum overprediction error is only 3%. Moreover, the maximum underprediction error is only 2%, which guarantees the least guardband wastage. Thus, the neural network produces a more accurate V_{min} prediction model than the model constructed using the linear regression method, which indicates a nonlinear relationship between V_{min} and performance counters.

5.3 Bottom-Up Approach for V_{min} Prediction

The use of all performance counters yields an accurate V_{min} prediction model. In this subsection, we evaluate a bottom-up approach for V_{min} prediction. The bottom-up approach identifies a small set of performance counters that strongly correlate with V_{min}, and constructs a simple model with comparable accuracy. Because these events cause different di/dt droop, it matches with the previous observation in Section 4.2 that the di/dt droop during kernel execution determines the V_{min}.

We first identify the most relevant performance coun-

(a) Piecewise linear model using the IPC, DRAM read throughtput and single precision FLOPS. (b) Prediction accuracy.

Figure 15: The details of using the bottom-up approach for V_{min} prediction, and the prediction model accuracy.

ters (i.e. features). We rank the 28 features using the Boruta package [32], which iteratively sorts each feature's relevance and ranks them by their relevance to the dependent variable V_{min}. We find that DRAM read throughput, IPC and single-precision floating-point operations per second rank highest among all counters. The reason for them being the most relevant to V_{min} might be that they represent the general pipeline activity, which determines the voltage noise profile.

We construct the V_{min} prediction model with the three most relevant counters using the following methodology. Because the peak achievable IPC in the Kepler architecture is four, we first categorize the kernels to four types by their IPC values: *[0, 1)*, *[1, 2)*, *[2, 3)* and *[3, 4]*. The heuristic we use is that the IPC is a good indicator of the pipeline stall degree. For example, kernels in the IPC *[3, 4]* region have V_{min} from 0.93 to 0.95 V. In this region, stalls are rare, and therefore the di/dt droop is small. In contrast, kernels in region *[0, 1)* have a larger V_{min} variability, from 0.9 to 0.97 V. In this region, pipeline stalls are frequent, and thus the di/dt droop may have a larger magnitude. IPC alone is not enough to capture the V_{min} variability. In each IPC region, we perform a piecewise linear regression with one breakpoint against the other two counters (DRAM read throughput and single precision FLOPS) separately. We show the results with better accuracy in Figure 15a. We summarize the key findings below.

IPC *[0, 1)* The undervolt level decreases as DRAM read throughput increases in this region. When both the read throughput and IPC are low, the pipeline stalls are most likely due to the lack of threads (i.e. latency bound in Section 5.1). In this case, the di/dt droop is small and the undervolt level is large. Kernels with high read throughput and low IPC are memory bound. They have enough threads utilizing the memory bandwidth, which is the prerequisite for generating a large current surge after the stall and thus large di/dt droop.

IPC *[1, 2)* Single-precision FLOPS correlates better with V_{min} than DRAM read throughput in this region. The undervolt level decreases rapidly as the FLOPS increases until 10 GFLOPS and plateaus. This can be attributed to the fact that higher FLOPS can cause a larger current surge and therefore a larger di/dt droop.

IPC *[2, 3)* The undervolt level first increases but then decreases as the read throughput increases. Kernels with low read throughput and medium IPC are computing bound. They have mostly dependence-induced stalls. The increased read throughput means more memory stalls, but the two types of stalls are unlikely to align with each other, and thus the noise begins to decrease [23, 31]. However, the kernel becomes memory bound after the read throughput increases above 20 GB/s. Then, the di/dt droop starts to increase similar to the kernels in the IPC region of [0, 1).

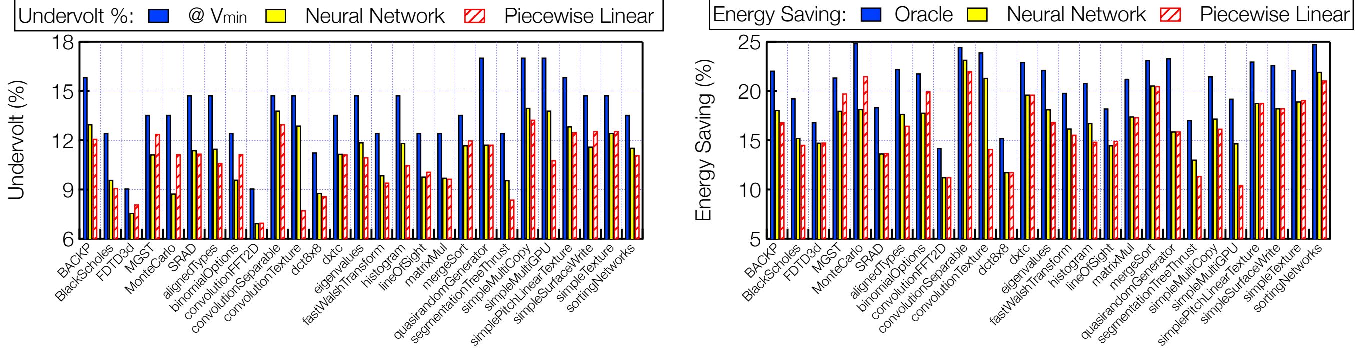
IPC *[3, 4]* As mentioned earlier, kernels in this region have a small V_{min} variability because stalls are rare, and therefore the di/dt droop is small. The undervolt level increases slightly as the read throughput increases.

Figure 15b shows the accuracy of the final prediction model derived using the above method. It has the RMSE of 0.9%, and the maximum overprediction and underprediction error is 2.7% and 3.3%, respectively. The accuracy of this model using three key counters is comparable with the model produced by the neural network approach using all performance counters.

6. ENERGY EFFICIENCY OPTIMIZATION

In this section, we discuss the energy efficiency improvement potential of reducing the voltage operating margin to the safe limits. We first present the energy-efficiency benefit analysis of the oracle case when operating the GPU at the V_{min} point. We then discuss the energy benefits in the context of an exemplary design concept that reduces the operating margin by deploy-

Figure 16: Energy savings for each program operating at the V_{min} for GTX 480 and 680 card.



(a) Undervolting level at V_{min} point and predicted V_{min} point. (b) Energy savings with oracle scheme versus predictive scheme.

Figure 17: Comparison of (a) undervolting level and (b) energy savings at the measured and predicted V_{min} point.

ing the V_{min} prediction model and ensures functional correctness with a sensor-based safety mechanism.

6.1 Oracle Analysis

We first show the measured energy savings by running each program at its V_{min} point. Initially, we experimentally measure a program's V_{min} as shown in previous sections. We then measure the overall GPU's energy consumption under two scenarios: i) with nominal frequency and voltage, and ii) with nominal frequency and V_{min}. We measure a subset of programs shown in Section 2 that have long-running kernels. We measure GPU power at the card level, which includes the power consumption of the GPU chip, DRAM and peripherals.

Figure 16 shows the energy savings on two different GPU cards: GTX 480 and GTX 680. By lowering the core voltage without changing the frequency, we can improve energy efficiency. On (geometric) average, the energy saving is about 21% for GTX 680 and 15.8% for GTX 480. The energy savings range from 14% to 25% for GTX 680, and from 8% to 22% for GTX 480.

The energy savings depend on two factors: the percentage of voltage that can be reduced, and the ratio of GPU chip power consumption at the card level. Note that the undervolting impacts only the GPU chip power consumption but not the DRAM power consumption. For GTX 680, the smallest improvement is seen with convolutionFFT2D. We can reduce its energy consumption by 14%. It has the highest V_{min} value of 0.99 V, which results in the smallest gain in energy savings. The benchmark MonteCarlo has the largest saving of 25%, although it has only a medium V_{min} value of 0.94 V. This is because the card-level power is dominated by the GPU chip power but not the DRAM power. We observe that the energy savings with GTX 480 are generally lower than GTX 680. This can be explained by the higher DRAM power consumption for GTX 480 card.

6.2 Exemplary Design and Benefit Analysis

Having established the (oracular) opportunity of energy savings, we now consider an exemplary design to claim the potential energy benefits shown previously. Note again that our goal is to improve the GPU's energy efficiency while ensuring absolute correctness.

In this conceptual design, the system firmware (or runtime software) reduces the voltage operating margin to save energy. It uses specially architected hardware performance counters and a V_{min} prediction model to predict and operate with the reduced margin. Although our model predicts V_{min} within 3% error margin for training programs, the prediction error for unseen programs, whose characteristics are very different from the training programs, could exceed the 3% error margin. This kind of corner case can result in a system failure. Thus, our design must be augmented with an overall "safety net" to ensure functional correctness.

Our scheme requires only a lightweight safety net in the hardware because the software takes the role of margin detection (via prediction) from the hardware. The existing lightweight voltage droop sensor, called the skitter circuit [33, 34], suits our system well. The control firmware can use this sensor to detect the corner case, i.e., the larger-than-expected voltage droop that causes a violation of the 3% error margin established from model training. Upon detecting such a case, the control firmware would restore the voltage to the nominal level and roll back to the last checkpoint to guard against any timing-error-related corruption that could have occurred. Then, this program can be added to the offline training set to recalibrate the prediction model.

Our exemplary software-hardware (cross-layer) design for reducing voltage margin is distinct from the hardware-only feedback control loop. The latter relies on complex and high-calibration-overhead sensors like critical path monitors [35]. This exemplary design relies only on a lightweight skitter circuit. Other lower-complexity error detectors, such as parity or ECC sensors [36, 37] available in large SRAM macros (e.g. caches or register files), can also provide an alert about error-prone voltage levels and can help in the exemplary design. Because these hardware detectors are limited to memory arrays, we also need the skitter circuits for the logic

paths that are engaged in computation and control.

We demonstrate that we can achieve energy savings close to the oracle case using the prediction models presented in Section 5.2. We evaluate two cases of executing each program at the undervolting level predicted by the neural network and the piecewise linear prediction model separately. Because both models have a maximum overprediction of less than 3%, we add another 3% margin to guardband this model prediction error. Figure 17a compares the undervolting level at the V_{min} point and the level predicted by the two models plus the additional 3% margin. The predicted undervolting level of both models is always lower than the level at the V_{min} point, which means that all those programs execute correctly without any faults. However, the safety net mechanism is still required for the functional correctness of other programs. The average gap between the actual and predicted undervolting level is only 2.7% for both models, which minimizes the guardband wastage.

Figure 17b compares the energy savings running at the undervolting level predicted by the two models with the savings in the oracle case. The energy savings range from 11.3% to 23.2% using the neural network model and 10.5% to 22% using the piecewise linear model, as opposed to 14% to 25% in the oracle case. The average savings are 16.9% and 16.3% using the two models and 21% in the oracle case. Using both models achieves over 80% of the energy-savings benefits of the oracle case.

7. RELATED WORK

To the best of our knowledge, our work is the first to perform a comprehensive measurement-based study of voltage guardband in GPUs. We compare and contrast our work with prior work on voltage guardband both in CPU and GPU domains. In our work, we conduct the V_{min} test on a class of off-the-shelf GPU cards to characterize their voltage guardband. Prior work on reducing the CPU voltage guardband is categorized into one of two types. The first type reduces the voltage guardband while the processors continue to function correctly [5], whereas the second type tolerates timing speculation errors with the aid of an error detection and recovery mechanism [38]. Similar to the former scenario, the V_{min} test assumes that no error occurs at the V_{min} point. However, our work's emphasis is to characterize the voltage guardband and build a fundamental understanding of its important characteristics on a GPU.

Our measurement results reveal that the di/dt droop during the kernel execution is the largest contributor to voltage guardband in the multiple studied GPUs. Many prior works adopt the simulation approach to study the di/dt droop in the single-core [27, 39, 40] and multi-core [28, 41] CPUs. There are also prior efforts that conduct a measurement-based study of voltage noise in CPUs [4, 23, 42]. Prior work on voltage noise in GPUs focus on modeling [43] and characterization [31] via simulation. Our work is the first to perform a comprehensive measurement study of voltage noise using multiple off-the-shelf GPU cards. We measure each program's V_{min} and characterize its error behavior and probability when the GPU chip's voltage goes below V_{min}.

Prior work in CPUs relies on hardware sensors such as the critical path monitor [5], shadow latches [38] or ECC feedback [36, 37] to reduce the operating margin for energy saving. To the best of our knowledge, our work is the first to use performance counter measurement for each kernel to predict its V_{min} value accurately and achieve near-optimal energy savings by operating the GPU with a program-specific undervolting level.

8. CONCLUSION

We demonstrated that we can achieve energy-reduction benefits as high as 25% by lowering the GPU's core voltage without inducing errors. The challenge for leveraging this opportunity lies in understanding what influences the choice of the safe limit. We find that the di/dt droop is the largest determinant of V_{min}. We find that GPU V_{min} is program specific. We characterize the impact of program characteristics on the V_{min}, and show that we can predict the V_{min} of each individual kernel using performance counter information. We demonstrate that we can achieve a large energy savings by operating the GPU at the predicted V_{min} point.

Acknowledgments

We thank Yazhou Zu for important contributions to this project. We are also grateful to the reviewers for comments that helped us to improve the quality of the paper. This work is sponsored in part by Defense Advanced Research Projects Agency (DARPA), Microsystems Technology Office (MTO), under contract number HR0011-13-C-0022, National Science Foundation (NSF), under grant CCF-1218474, and Semiconductor Research Corporation (SRC). The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense, the NSF, the SRC or the U.S. Government. This document is: Approved for Public Release, Distribution Unlimited.

9. REFERENCES

[1] NVIDIA, "GTX 680 Kepler Whitepaper - GeForce." http://goo.gl/fyg2z1, 2012.

[2] NVIDIA, "NVIDIA GeForce GTX 980 Whitepaper." http://goo.gl/btRXed, 2014.

[3] J. Leng, T. H. Hetherington, A. ElTantawy, S. Z. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: enabling energy optimizations in GPGPUs," in *International Symposium on Computer Architecture*, pp. 487–498, 2013.

[4] V. J. Reddi, S. Kanev, W. Kim, S. Campanoni, M. D. Smith, G. Wei, and D. M. Brooks, "Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling," in *International Symposium on Microarchitecture*, pp. 77–88, 2010.

[5] C. Lefurgy, A. J. Drake, M. S. Floyd, M. Allen-Ware, B. Brock, J. A. Tierno, and J. B. Carter, "Active management of timing guardband to save energy in POWER7," in *International Symposium on Microarchitecture*, pp. 1–11, 2011.

[6] V. J. Reddi, M. S. Gupta, G. H. Holloway, M. D. Smith, G. Wei, and D. M. Brooks, "Predicting voltage droops

using recurring program and microarchitectural event activity," *IEEE Micro*, vol. 30, no. 1, p. 110, 2010.

[7] "MSI Afterburner." http://goo.gl/fs2pti.

[8] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi." http://goo.gl/zmoJkZ, 2009.

[9] NVIDIA Corporation, "CUDA C/C++ SDK CODE Samples," 2011.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *International Symposium on Workload Characterization*, pp. 44–54, 2009.

[11] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *International Symposium on Workload Characterization*, pp. 141–151, 2012.

[12] National Instruments, "NI DAQ 6133." http://goo.gl/ez2mof, 2015.

[13] NVIDIA, "NVIDIA Visual Profiler User Guide." http://goo.gl/gefn6p, 2015.

[14] NVIDIA, "CUDA Profiling Tools Interface." http://goo.gl/nbAVCf, 2015.

[15] C. Constantinescu, I. Parulkar, R. Harper, and S. Michalak, "Silent data corruption - myth or reality?," in *International Conference on Dependable Systems and Networks*, pp. 108–109, 2008.

[16] Janak Patel, "CMOS process variations: A critical operation point hypothesis.." http://goo.gl/K0yWkf, 2008.

[17] S. Bhardwaj, S. B. K. Vrudhula, P. Ghanta, and Y. Cao, "Modeling of intra-die process variations for accurate analysis and optimization of nano-scale circuits," in *Design Automation Conference*, pp. 791–796, 2006.

[18] M. Orshansky, L. Milor, and C. Hu, "Characterization of spatial intrafield gate CD variability, its impact on circuit performance, and spatial mask-level correction," *IEEE Transactions on Semiconductor Manufacturing*, vol. 17, no. 1, pp. 2 – 11, 2004.

[19] S. Roy and A. Asenov, "Where Do the Dopants Go?," *Science*, vol. 309, no. 5733, pp. 388 – 390, 2005.

[20] K. Okada and H. Onodera, "Statistical Parameter Extraction for Intra- and Inter-Chip Variabilites of MetalOxide-Semiconductor Field-Effect Transistor Characteristics," *Japanese Journal of Applied Physics*, vol. 44(A), pp. 131–134, 2005.

[21] P. Lu, K. A. Jenkins, T. Webel, O. Marquardt, and B. Schubert, "Long-term NBTI degradation under real-use conditions in IBM microprocessors," *Microelectronics Reliability*, vol. 54, no. 11, pp. 2371–2377, 2014.

[22] E. Grochowski, D. Ayers, and V. Tiwari, "Microarchitectural simulation and control of di/dt-induced power supply voltage variation," in *International Symposium on High-Performance Computer Architecture*, pp. 7–16, 2002.

[23] R. Bertran, A. Buyuktosunoglu, P. Bose, T. J. Slegel, G. Salem, S. M. Carey, R. F. Rizzolo, and T. Strach, "Voltage noise in multi-core processors: Empirical characterization and optimization opportunities," in *International Symposium on Microarchitecture*, pp. 368–380, 2014.

[24] H. M. Jacobson, A. Buyuktosunoglu, P. Bose, E. Acar, and R. J. Eickemeyer, "Abstraction and microarchitecture scaling in early-stage power modeling," in *International Conference on High-Performance Computer Architecture*, pp. 394–405, 2011.

[25] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient GPU architectures," in *International Symposium on Microarchitecture*, pp. 86–98, 2013.

[26] NVIDIA, "CUDA Runtime API." http://goo.gl/G27upA, 2015.

[27] M. D. Powell and T. N. Vijaykumar, "Pipeline damping: A microarchitectural technique to reduce inductive noise in supply voltage," in *International Symposium on Computer Architecture*, pp. 72–83, 2003.

[28] T. N. Miller, R. Thomas, X. Pan, and R. Teodorescu, "VRSync: Characterizing and eliminating synchronization-induced voltage emergencies in many-core processors," in *International Symposium on Computer Architecture*, pp. 249–260, 2012.

[29] NVIDIA, "CUDA C Programming Guide." http://goo.gl/22Cac8, 2015.

[30] NVIDIA, "Parallel Thread Execution ISA Version 4.2." http://goo.gl/KIQcAY, 2015.

[31] J. Leng, Y. Zu, and V. J. Reddi, "GPU voltage noise: Characterization and hierarchical smoothing of spatial and temporal voltage noise interference in GPU architectures," in *International Symposium on High Performance Computer Architecture*, pp. 161–173, 2015.

[32] M. Kursa and W. Rudnicki, "Feature Selection with the Boruta Package," *Journal of Statistical Software*, vol. 36, no. 11, 2010.

[33] R. L. Franch, P. Restle, J. K. Norman, W. V. Huott, J. Friedrich, R. Dixon, S. Weitzel, K. van Goor, and G. Salem, "On-chip timing uncertainty measurements on IBM microprocessors," in *International Test Conference*, pp. 1–7, 2008.

[34] P. Restle, R. Franch, N. James, W. Huott, T. Skergan, S. Wilson, N. Schwartz, and J. Clabes, "Timing uncertainty measurements on the power5 microprocessor," in *International Solid-State Circuits Conference*, pp. 354–355 Vol.1, Feb 2004.

[35] A. J. Jrake, R. M. Senger, H. Deogun, G. D. Carpenter, S. Ghiasi, T. Nguyen, N. K. James, M. S. Floyd, and V. Pokala, "A distributed critical-path timing monitor for a 65nm high-performance microprocessor," in *International Solid-State Circuits Conference*, pp. 398–399, 2007.

[36] A. Bacha and R. Teodorescu, "Dynamic reduction of voltage margins by leveraging on-chip ECC in itanium II processors," in *International Symposium on Computer Architecture*, pp. 297–307, 2013.

[37] A. Bacha and R. Teodorescu, "Using ECC feedback to guide voltage speculation in low-voltage processors," in *International Symposium on Microarchitecture*, pp. 306–318, 2014.

[38] D. Ernst, N. S. Kim, S. Das, S. Pant, R. R. Rao, T. Pham, C. H. Ziesler, D. Blaauw, T. M. Austin, K. Flautner, and T. N. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *International Symposium on Microarchitecture*, pp. 7–18, 2003.

[39] M. D. Powell and T. N. Vijaykumar, "Pipeline muffling and a priori current ramping: architectural techniques to reduce high-frequency inductive noise," in *International Symposium on Low Power Electronics and Design*, pp. 223–228, 2003.

[40] R. Joseph, D. M. Brooks, and M. Martonosi, "Control techniques to eliminate voltage emergencies in high performance processors," in *International Symposium on High-Performance Computer Architecture*, pp. 79–90, 2003.

[41] M. S. Gupta, J. L. Oatley, R. Joseph, G. Wei, and D. M. Brooks, "Understanding voltage variations in chip multiprocessors using a distributed power-delivery network," in *Design, Automation Test in Europe Conference*, pp. 624–629, 2007.

[42] Y. Kim, L. K. John, S. Pant, S. Manne, M. J. Schulte, W. L. Bircher, and M. S. S. Govindan, "AUDIT: stress testing the automatic way," in *International Symposium on Microarchitecture*, pp. 212–223, 2012.

[43] J. Leng, Y. Zu, M. Rhu, M. S. Gupta, and V. J. Reddi, "GPUVolt: modeling and characterizing voltage noise in GPU architectures," in *International Symposium on Low Power Electronics and Design*, pp. 141–146, 2014.