DYNAMIC VOLTAGE/FREQUENCY SCALING IN GPUs THROUGH GENETIC ALGORITHM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Pouria Hasani
September 2022

Dynamic Voltage/Frequency Scaling in GPUs Through Genetic Algorithm
By Pouria Hasani
September 2022

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Özcan Öztürk(Advisor)

Uğur Güdükbay

Süleyman Tosun

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

DYNAMIC VOLTAGE/FREQUENCY SCALING IN GPUs THROUGH GENETIC ALGORITHM

Pouria Hasani
M.S. in Computer Engineering
Advisor: Özcan Öztürk
September 2022

Dynamic Voltage/Frequency Scaling (DVFS) is the primary approach to optimizing Central Processing Units (CPUs) power consumption. A handful of approaches are conducted using this technique in General Purpose Graphics Processing Units (GPGPUs). However, due to the massively parallel execution of threads on GPUs and load imbalance on Streaming Multiprocessors (SMs), finding the best global frequency for GPU cores is not a simple task. Moreover, the proposed approaches in the literature mostly rely on an offline model, where the optimal voltage and frequency for an application is found to be used in the next execution. In this work, we use a combination of an analytical model and a genetic algorithm to adjust per SM frequency dynamically, aiming at decreasing GPU's power consumption with the least amount of performance loss without a need for offline execution. We tested our approach using 16 GPU kernels from different domains with ranging features. Our results show that we can save 9.6% of GPU's total energy on average with less than 0.95% performance loss. We also discuss further improvements and possible extensions to the proposed approach.

Keywords: Dynamic Voltage/Frequency Scaling (DVFS), General Purpose Graphics Processing Units (GPGPUs), Streaming Multiprocessors (SMs), Energy.

ÖZET

GENETİK ALGORİTMA İLE GRAFİK İŞLEMCİ ÜNİTELERİNDE DİNAMİK VOLTAJ/FREKANS ÖLÇEKLEME

Pouria Hasani
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Özcan Öztürk
Eylül 2022

Dinamik Voltaj/Frekans Ölçekleme (DVFS), Merkezi İşlemci Ünitelerinin (CPU) güç tüketimini verimli hale getirmek için kullanılan en temel yaklaşımdır. DVFS tekniğini Grafik İşlemci Ünitelerinde Genel Amaçlı Hesaplama (GPGPU) işlemlerinde kullanmak için birtakım araştırmalar yapılmıştır. Ancak, iş parçacıklarının Grafik İşlemci Ünitelerinde (GPU) büyük ölçüde paralel yürütülmesi ve Paylaşımlı Çoklu İşlemciler (SM) arasındaki yük dengesizliği nedeniyle, GPU çekirdekleri için en uygun küresel frekansı bulmak kolay bir iş değildir. Ayrıca, literatürde yer alan öneriler çoğunlukla, bir uygulamadaki optimal voltaj ve frekansın bir sonraki işletimde kullanılması için kurulan çevrimdışı bir modele dayanmaktadır. Bu çalışmada, bir analitik model ve genetik algoritma bileşimi kullanılarak, SM frekansının dinamik olarak ayarlanması, böylece çevrimdışı yürütmeye gerek kalmadan GPU'nun güç tüketiminin en az performans kaybıyla azaltılması hedeflenmektedir. Yaklaşımımızı çeşitli özelliklere sahip farklı alanlardan 16 GPU çekirdeği kullanarak test ettik. Elde ettiğimiz sonuçlar %0.95'ten daha az performans kaybıyla GPU'nun toplam enerjisinden ortalama %9,6 tasarruf sağlayabileceğimizi gösteriyor. Ayrıca önerilen yaklaşımın nasıl daha da geliştirilebileceği ve olası yeni yaklaşımlar tartışılmıştır.

Anahtar sözcükler: Dinamik Voltaj/Frekans Ölçekleme (DVFS), Grafik İşleme Birimlerinde Genel Amaçlı Hesaplama (GPGPUs), Paylaşımlı Çoklu İşlemciler (SMs), Enerji.

Acknowledgement

I want to show gratitude to my advisor Prof. Özcan Öztürk who patiently supported me even though I pushed the limitation of his tolerance to the fullest. I am grateful for his patience and guidance. I would never be able to finish my studies if it was not for his exceptional tolerance and support. I also want to thank the jury members, Prof. Uğur Güdükbay and Prof. Süleyman Tosun, for reviewing the thesis and providing valuable feedback.

I want to give a special thanks to my parents and my sister Parisa who truly filled my life with love. I also would like to thank my lovely wife Gamze, who was beside me during all the happy and sad moments and made my life beautiful.

Lastly, I want to devote my thesis to my brother Peyman who always had my back.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

General Purpose Graphics Processing Units (GPGPUs) are the leading hardware platforms for the execution of state-of-the-art parallel applications [1][2]. Applications such as machine learning models, graph processing, matrix operations, and molecular dynamics simulations [3] perform single instruction on multiple independent data. This means we can potentially achieve lots of speedup in the correct setting. Since GPGPUs have hundreds of cores, and each of these cores can execute one or more instructions in every clock cycle, they provide the perfect functionality for the above-mentioned massively parallel applications.

While GPGPUs can speed up the execution time significantly, they introduce a considerable cost due to their high power consumption [4][5]. Moreover, higher power consumption increases the device's temperature, and cooling down the device with a higher temperature requires more effort in the cooling system. One of the primary techniques for reducing power consumption is Dynamic Voltage and Frequency Scaling (DVFS). DVFS refers to the ability of the hardware to dynamically change different components' voltage and frequency. Setting the voltage and frequency to lower levels can decrease power usage substantially with the cost of performance loss. Minimizing the performance loss and maximizing the power saving using DVFS require detailed knowledge of the application's run-time characteristics [6]. Most of the state-of-the-art works in using DVFS

rely on designing power and performance models that can predict applications' execution time and hardware's power consumption to find the optimal voltage and frequency configurations.

DVFS has been fully utilized in CPUs [7][8][9][10]. However, GPGPUs, which provide extensive support for DVFS, cannot take advantage of this technique due to their massively parallel execution nature and unpredictable run-time behavior [11]. The complex interaction between different components like the memory system and Streaming Multiprocessors (SMs) causes difficulty in designing accurate power and performance models to apply DVFS optimizations on GPUs. Previous studies have proposed to collect applications' statistics after a certain number of cycles, called an *interval*. The collected statistics in each interval are then used to predict the GPU's performance in the upcoming intervals with the assumption that the upcoming intervals are computationally similar to the current ones. Since this is a safe premise [12], in our work, we use the same methodology. We have designed a simple performance model to predict the number of executed instructions. For this, we collect the applications' statistics at the end of a certain number of cycles and predict the number of executed instructions in the following intervals.

The other factor that prevents the better usage of DVFS in GPUs is the SMs load imbalance at run-time [13][14]. During the execution of an application, some SMs execute instructions without many memory accesses (compute phase), while others are waiting for memory responses (memory phase). The performance of the SMs that are in a compute phase has a linear relation with the core's frequency. On the contrary, the performance of the SMs in the memory phase is less sensitive to this frequency. Decreasing the core's global frequency will save power, but the SMs without memory requests will have significant performance loss. In this thesis, we consider per-SM DVFS adjustment to account for the different run-time execution characteristics of different SMs.

Additionally, the memory access of one SM can change the performance of other SMs, as it introduces contention in the interconnection network and memory components. This makes power optimization more complicated. In this regard,

the effect of different SMs' run-time behavior on one another is also needed to be considered while using DVFS. We also propose to use an optimization metric that can tackle this effect. Overall, in our work, we run the application for a constant number of cycles (*interval*) and collect the statistics for our performance model. In parallel, we also use this data in a power model that can predict the GPU's power usage. Then we utilize the power and performance model in a genetic algorithm-based optimization framework. The genetic algorithm-based model predicts the optimal frequencies for SMs. The prediction and optimization operations are done in parallel with the next interval. That is, during $(i+1)^{th}$ interval, we use the data collected in the i^{th} interval and find the best frequency set for $(i+2)^{th}$ interval. These operations continue to be performed for all of the intervals until the GPU program terminates.

1.1 Motivation

The GPU's power utilization accounts for a significant percentage of the system's total power cost. On different computer systems, from supercomputers to mobile devices, decreasing even a small percentage of GPU's power usage can lead to a considerable reduction in the total energy cost [15]. The GPU's power consumption can be divided into two parts [15][16][17][18]: 1. Dynamic power, 2. Static power. The dynamic power changes proportionally with $Voltage^2 * frequency$ [16], whereas static power is correlated with $Voltage^2$ [19]. On most of the devices, the *voltage* is set automatically when adjusting the *frequency*, and lower frequencies translate into lower voltages as well. Hence, frequency reduction can potentially decrease both static and dynamic power.

However, if the application's behavior during the execution is not considered, changing the core's frequency can drastically affect the GPU's performance. We run two GPU kernels from the Rodinia benchmark suite [20] to show the importance of considering applications' characteristics while applying DVFS. As shown in Table 1.1, both kernels are first executed with the default frequency of 700MHz, and the average stall cycles and execution times are listed. We also execute both

benchmarks with an average SM frequency of 500MHz, where the frequency is chosen dynamically and varies during the execution time. As seen from this table, the average stall cycle for BFS is 32% of the total busy cycles, whereas it is 8.94% for Backprop. However, the performance loss for Backprop is significantly higher than that for BFS due to lower stall cycles. Our performance model considers the stall cycles in the current interval and uses this data to predict the number of executed instructions for the upcoming intervals.

Table 1.1: Comparison of two GPU kernels' average stall time and performance loss while applying DVFS. Benchmarks are executed with an average SM frequency of 500MHz and compared against the default frequency of 700MHz. The frequencies are chosen dynamically and varies during the execution time.

Kernel name	Average stall cycles (%)	Performance loss (%)
BFS	32.30	4.12
Backprop	8.94	28.34

Using DVFS in GPUs requires a more fine-grained design to harvest the technique's benefits when compared with CPUs. Most of the problems in elevating DVFS in GPUs are due to the Single Instruction Multiple Threads (SIMT) nature. While thousands of threads run on different SMs, their execution features vary drastically. Figure 1.1 shows an example of the variation in the stalls among different SMs. More specifically, the figure depicts the number of stall cycles of the first 8 SMs of GTX 480 GPU hardware with 15 SMs. The results are collected for 100 intervals while running CFD from the Rodinia benchmark suite [20]. As can be seen from this figure, SM0 and SM7 experience a high number of stall cycles in the first ten intervals, while SM5 and SM6 are in their compute phase. Between intervals 40 to 50, SM0 and SM4 take a small share of total stall cycles across all 8 SMs. The rest of the intervals also show the imbalance in the number of stall cycles. Based on this observation, we believe it is critical to implement DVFS in individual SMs, instead of applying it globally.

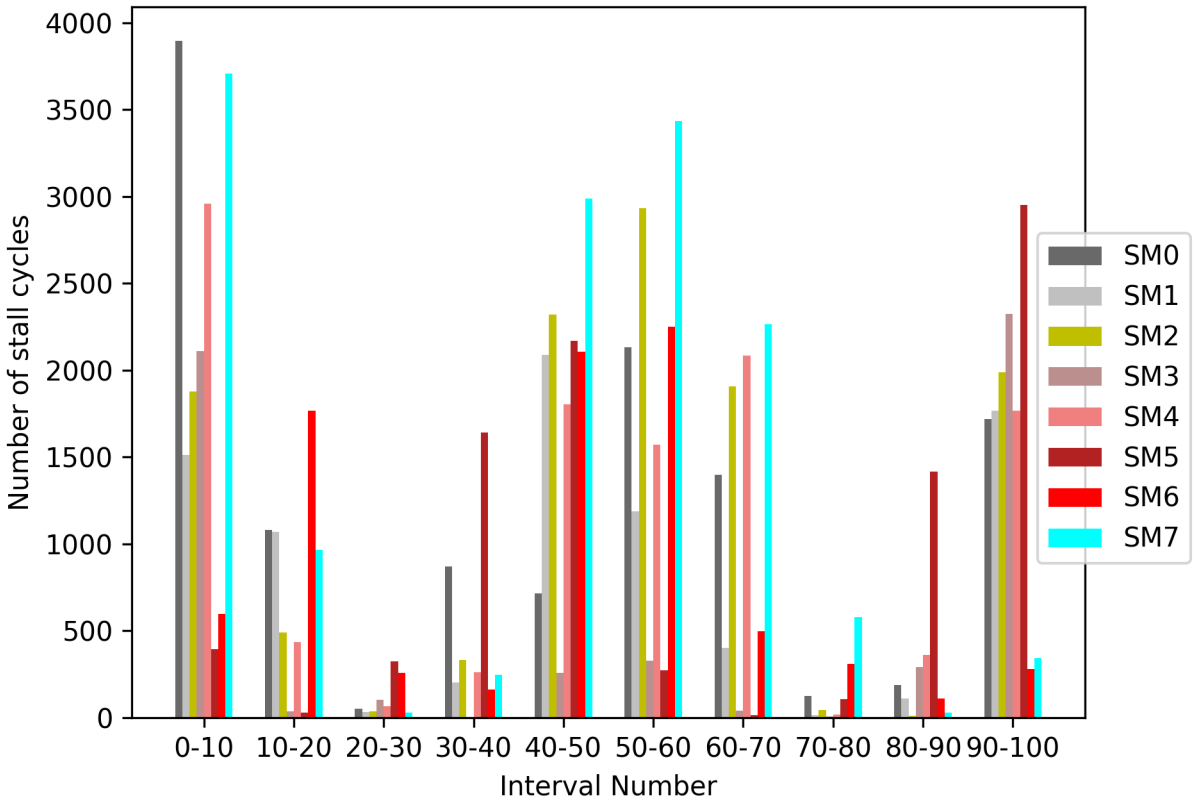

Figure 1.1: Snapshot of the number of stall cycles of the first 8 SMs of GTX 480 with 15 SMs while running CFD for the first 100 intervals.

Besides the imbalanced distribution of the stall cycles between different SMs, it is very common that some SMs stay idle while others execute instructions. This is not due to the memory access, but rather it is caused by the fact that some SMs run out of instructions to execute. This can happen when all threads assigned to some SMs are completed while other SMs still have running threads. Or, it can be due to the synchronization in cooperative groups in which a certain number of blocks are grouped, and the program may require synchronization between these thread blocks. Therefore, SMs that reach the synchronization point earlier than others in one cooperative group will stay idle until all the SMs reach that point. Figure 1.2 shows a snapshot of the execution cycles of the first 8 SMs of GTX 480 GPU hardware while running CFD from the Rodinia benchmark suite. In this figure, Y-axis indicates the idleness of the respective SM. As shown in Figure 1.2,

different SMs can have idle cycles at different intervals. Idle cycles provide a great opportunity to save power by decreasing the core's frequency. Since there are no instructions in the pipeline, performance degradation is expected to be minimal.

By interpreting Figures 1.1 and 1.2, we can conclude that each SM's run-time behavior can change as it proceeds to execute instructions. At any cycle, one SM can be in a compute phase, a memory phase, or stay idle. This indicates the necessity of having an online DVFS framework that can adjust the voltage and frequency for each SM for current run-time features. In our framework, we use per-SM data collection and frequency adjustment to address the unbalanced execution of the application on different SMs.

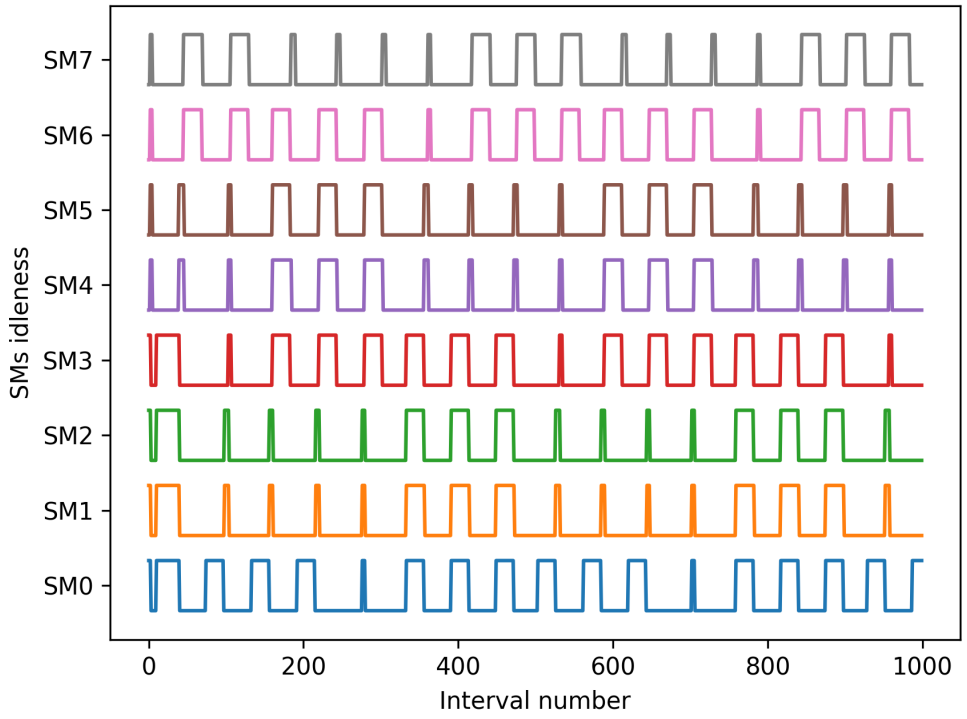

Figure 1.2: Busy interval distribution of the first 8 SMs for the first 1000 intervals while running CFD from Rodinia benchmark.

1.2 Contribution

In this study, we present an online DVFS optimization framework. Firstly, we introduce a simple performance model to predict the number of executed instructions to be used for the upcoming sampling intervals. We also use a power model adopted from the literature to predict dynamic and static power alongside the performance model. Using these two models, we design a genetic algorithm-based optimization framework to adjust the voltage and frequency of each SM separately. Our genetic algorithm aims at reducing the power per instruction by utilizing power and performance models. While we believe per-SM frequency adjustment is internally possible and accountable, we do not have a way of implementing it in the current GPU hardware and tool-set. Hence, we use a state-of-the-art GPU simulator [21] by modifying it to account for per-SM statistics collection and power calculation. This GPU simulator, called GPGPUSIM, is a functional and timing simulator for GPUs, and it is widely used in the literature. Our genetic algorithm implementation is also integrated into the tool.

The main contributions of the proposed approach can be summarized as follows:

- We use a simple performance model to predict the number of executed instructions for the following intervals along with the stall cycles.
- We provide an analytical optimization model to adjust per-core voltage and frequency aiming at minimizing power per instruction.
- We implement a genetic algorithm model to minimize total power across all SMs by adjusting voltage/frequency individually through DVFS.
- We extend a state-of-the-art GPU simulator to account for per-SM data collection and to provide DVFS optimization support.
- We test our approach on GPU kernels and compare the results with the literature.

1.3 Organization

The rest of the thesis is organized as follows. Chapter 2 gives the related work, whereas Chapter 3 discusses the GPU architecture, power and performance models used, and our optimization metric. Chapter 4 elaborates on the analytical optimization model, and in Chapter 5, we explain the details of our genetic algorithm based model. Chapter 6 gives the experimental setup and the results. Finally, in Chapter 7, we conclude the thesis.

Chapter 2

Related Work

In this chapter, we explain the state-of-the-art approaches to DVFS optimizations on GPUs. We can broadly classify GPU energy optimization and performance modeling studies into statistical and analytical approaches. In statistical methods, the GPU device is considered as a black box, and the applications' statistics are collected during their execution [22][17][23][24][16]. These statistics are then fed to trained machine learning models to predict the power and performance [25][26][27][28][29]. Statistical methods require collecting many performance counters at run-time, which introduces extra overhead to DVFS optimizations [30][31][32]. In our method, we use only two performance counters with negligible overhead compared to the statistical approach in which many hardware counters are used. Moreover, different architectures provide different hardware and performance counters, which makes it even harder to apply the same model across different GPU architectures. The analytical methods include breaking down the execution pipeline into pieces and modeling the main bottlenecks contributing to the execution time and power consumption [33][22][34].

2.1 Statistical Methods

Abe et al. applied linear models assuming that the power and performance have a linear relationship with a certain set of hardware counters [24]. They evaluated their model across three different GPU architectures, where they had 33.5% to 67.9% performance prediction error on average.

Wu et al. [17] and Ardalani et al. [23] used machine learning models to predict GPU's power and performance. Wu et al. trained a machine learning model using different kernels' performance counters collected with different GPU settings during their executions. Then for new applications, one run with the default setting is done to collect the hardware counters. These hardware counters are fed to the machine learning model to predict the power and performance on different GPU settings. The proposed method [17] requires an offline execution of the kernel, which usually is impractical because most of the time, especially in machine learning applications, the first run takes hours or even days to finish. Therefore, this can also be considered an offline model. In such offline methods, the kernels are executed once, and the necessary data is collected to feed the power and performance models. Later, the GPU is configured with the optimal voltage and frequency for the subsequent executions of the kernels.

In [16][18][26][27][28], similar offline methods have been proposed. Wang et al. designed a performance model using hardware and kernel performance counters collected during the execution of the kernels with default settings [26]. Then, they predicted the execution time with different core and memory frequencies on different architectures. Guerreiro et al. modeled per component power consumption using a target microbenchmark [27]. Later, they extended their work to add more components and use more microbenchmarks to improve the model's accuracy [26]. Their work can also predict GPU power consumption on default frequency configuration without needing the power measurement equipment. Both of these approaches require an offline application execution before making any predictions.

The other issue with the offline methods is their inaccuracy in power and performance prediction when concurrent kernels are running on the GPU. In most realistic execution scenarios, especially in cluster computers, multiple applications can run on single GPU hardware simultaneously. There can potentially be conflicts or contention on the GPU hardware resources between concurrent kernels. Hence, the performance counters in the offline execution may not match the performance counters in concurrent execution of the same kernels. This can lead to a miss prediction of the execution time and power. Furthermore, the approaches mentioned above lack an accurate prediction for the complete duration of the application. As the application behavior can significantly change throughout the execution, a fine-grained prediction mechanism can take better advantage of DVFS when compared with a one-time offline DVFS adjustment.

Besides the offline approaches, several works use assembly code analyses to predict the applications' power and execution time with different voltage/frequency levels [35][36][37]. These models predict the performance and power consumption before running the application on the hardware. However, an application's power consumption and execution time may vary in different execution settings due to the concurrent execution of multiple applications on the hardware. Therefore, they cannot capture the run-time variation of applications' execution characteristics. In [37], a performance model has been designed that predicts the execution time and power consumption using an RNN model with the GPU assembly instructions (PTX). They used this model for energy saving, tested it on different architectures, and achieved 13.12% energy saving on average.

In our work, the performance and optimization models only use two performance counters that can be easily provided by the state-of-the-art GPU hardware, as opposed to the statistical methods in the literature that need many performance counters. Collecting many parameters through different performance counters incurs extra overhead both in hardware and execution time. Additionally, our proposed method applies DVFS configurations during the applications' run-time without needing an offline execution. Therefore, we can adapt to the live characteristics of an application with minimal overheads.

2.2 Analytical Methods

In [38], a power and performance model is proposed using hardware counters. More specifically, they break down the execution and memory pipelines to capture the execution characteristics better. Even though they achieve a 93.3% accuracy, the proposed model is hardware specific and requires extensive modification to use in a different architecture.

An online power and performance prediction model and DVFS optimization tool have been proposed in [39]. Their work includes a hybrid model that uses the data collected during the application execution. Firstly, they collect GPU's power, performance, and utilization rate. Based on these data, they detect iterative intervals of the application's run-time using FFT. The iterative intervals refer to the periodic sections of the execution time in which the GPU's power, performance, and utilization rate have similar data sequences. This iterative behavior is seen in applications with loops in their code. Later at the beginning of these intervals, they predict GPU's power and performance for different core and memory frequency settings. Then, they configure GPU's core and memory with the optimal frequencies. Although this method is suitable for applications with iterative behavior, it might not be helpful in cluster computing. Since multiple clients can share one GPU, and even though these clients may have an application with iterative nature, the GPU's run-time power, performance, and utilization rate statistics for these concurrent applications can interfere with one another, and FFT cannot find an iterative section with this mixed statistics. Hence, this method will not be able to adjust the best frequency.

Moreover, detecting iterative sections of applications at run-time requires saving interval statistics in array-based structures, which can occupy extra space in the memory. In our proposed approach, we record the statistics for each interval, and we replace these data with new statistics for later intervals. Therefore, in our method, the storage overhead is negligible.

Most of the efforts on optimizing GPU's power consumption using DVFS do

not consider per-SM frequency adjustment. In GPUWATTCH [40], even though they have applied per-SM frequency adjustment using stall time to decide on the frequency levels, it is unclear how they have chosen the frequency using stall time. In CRISP [12], the core's frequency is adjusted globally for all the SMs. They proposed a complete performance model considering memory-level parallelism. Although they have implemented a complex performance model, they adjust the frequency over the whole GPU, ignoring the load imbalance on different SMs. Moreover, their model needs extra adjustments in the scoreboard to have a more accurate performance model. In our approach, we record the stall cycles and the number of executed instructions that can be extracted from available hardware counters.

In this work, we take an analytical approach by detecting the main bottleneck contributing to the delay in execution of the applications using two performance counters. Using a few performance counters increases the scalability of the proposed method to be used on different architectures. Additionally, the DVFS optimizations are applied for each SM individually to account for load imbalance despite the proposed methods in the literature [38][39][12].

Chapter 3

Architecture and Background

3.1 GPU Architecture

GPUs were first designed for the execution of computer graphics computations. In such programs, thousands of instructions can be executed simultaneously on different data. Since GPU architectures can provide thousands of cores to execute these instructions, they were the optimal hardware platform for computer graphics computations. Later, other applications with similar features started to utilize the GPUs, hence the name GPGPUs. Figure 3.1 gives a high-level view of a general GPU architecture. As shown in this figure, memory blocks and SMs are the two main components of a GPU architecture. Although named differently in different architectures, SM is the major building block of the GPU. It is a collection of many cores that can run GPU's specific assembly code in a synchronized fashion. The number of cores in an SM varies between different architectures. For Nvidia's Fermi generation, this number is usually 32. Furthermore, SMs are grouped to form a cluster. The number of clusters and SMs in one cluster also differ among different architectures. In general, each cluster has one response FIFO which is used as a buffer saving the data collected from the interconnection network as a response to the memory requests of the SMs. The SMs in one cluster receive their data in a FIFO order from the response FIFO

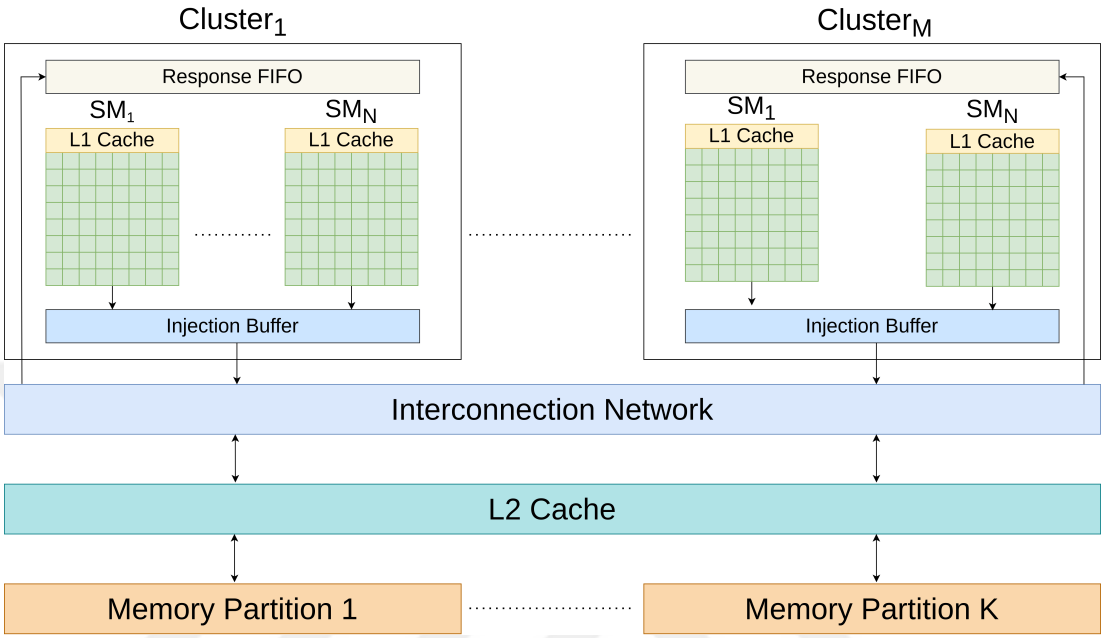

Figure 3.1: High level view of a general GPU architecture.

buffer. The memory system in GPU consists of different cache levels and per-SM shared memory alongside a global memory. After the first level cache miss, the memory requests go through the interconnection network, where the L2 cache is searched for the data. If the data is unavailable in the L2 cache, the instruction needs to wait until the data returns from the memory. The injection buffer in Figure 3.1 accepts memory accesses from the SMs and sends the requests to the interconnection network.

While running a GPU program, ideally, all the cores in the SM execute instructions at the same time. A GPU program consists of many threads. Each of these threads has its own memory space. These threads run the same instructions on different data and are grouped to form a warp. All threads in one warp have the same instructions running on different data. At each core clock cycle, a warp runs one or more instructions on each core of the corresponding SM. However, the GPU's pipeline may not be fully utilized. This can happen for different reasons, including the lack of application parallelism. When the program is sequential, concurrent threads cannot be executed, and some SMs may stay idle. Control divergence is another reason that prevents better utilization of the GPU. Control

divergence happens when different threads follow different control flows due to branch instructions executed. Since the cores on one SM strictly require executing the same instruction, threads with different instructions cannot fill all the core slots on the SM. The other reason for the lack of achieving GPU theoretical peak performance is the memory requests that stall the pipeline. When a warp encounters a cache miss, it stays inactive until the data returns from memory. At the same time, other active warps take turns executing their instructions on the SM. When multiple warps need to get data from the global memory, the warp scheduler runs out of active warps, and the pipeline has to stop until at least one warp becomes active again. During these cycles, the SM stays idle, providing an excellent opportunity for reducing voltage/frequency to decrease power consumption without any performance loss. Therefore, predicting such execution behavior is critical in GPU's online power optimizations through DVFS.

In our framework, for online DVFS optimization, we use simple power and performance models from the literature. From a high-level perspective, firstly, we run the application for a constant number of cycles, called an interval. On modern GPU hardware, 1μs is the smallest DVFS adjustment window with negligible overhead on GPU's performance [41]. Therefore, we set the interval time to 700 cycles with the default GPU's core frequency of 700MHz. This way, the interval time will be constant throughout the execution of the applications. In each interval, we record the dynamic and static power, stall time, and the total number of executed instructions for each SM separately. The collected data are then given to our genetic algorithm and analytical model. The genetic algorithm and analytical model use the power and performance models to calculate the optimal frequency set. The performance and power models are described in detail in Section 3.2 and Section 3.3, respectively.

The data collection and frequency adjustment is performed per SM. While it is technically possible to adjust SM frequencies individually, current GPU hardware and the respective tool sets do not provide a mechanism to implement this. Hence, we use a state-of-the-art GPU simulator, GPGPUSIM [21]. More specifically, we modify it to account for per-SM statistics collection and power calculation. This GPU simulator is a functional and timing simulator for GPUs and is widely used

in the literature. We extend the base simulator of GPGPUSIM to support per-SM data collection, power calculation, and dynamic frequency adjustment. We also integrate our analytical model and genetic algorithm into the tool.

GPUWATTCH [40] is an integrated tool with GPGPUSIM that calculates the power consumption of the simulated GPU. This tool uses run-time data of the application and hardware specifications to derive an accurate power model for the GPU. The default setting of GPUWATTCH combines all SMs data to calculate the overall power consumption. We extended the tool to calculate per SM power consumption.

Figure 3.2 depicts the overall mechanism to derive the cores' static and dynamic power, non-core related power, stall time, and the number of committed instructions using GPUWATTCH. Later, we use these data in our power, performance, and optimization models. In each interval, core-related data are recorded separately for each SM, and the non-core-related data are combined for the whole GPU. At the end of each interval, the collected data are given to GPUWATTCH. Modified GPUWATTCH calculates per-SM static and dynamic power and overall non-core related static and dynamic power.

The following sections describe our power and performance models used in the analytical model and the genetic algorithm.

3.2 Performance Model

For each interval, we assume that the execution time consists of two separate phases: *computation* phase and *stall* phase [42]. In the computation phase, the total execution time is just due to execution latency in the pipeline. If there is no stall, meaning if all the cache accesses were hit, computation time would be equal to the total execution time of the current interval. The stall phase is when the instructions have encountered a cache miss, waiting for their data to return from the main memory, thereby stalling the pipeline completely. The execution

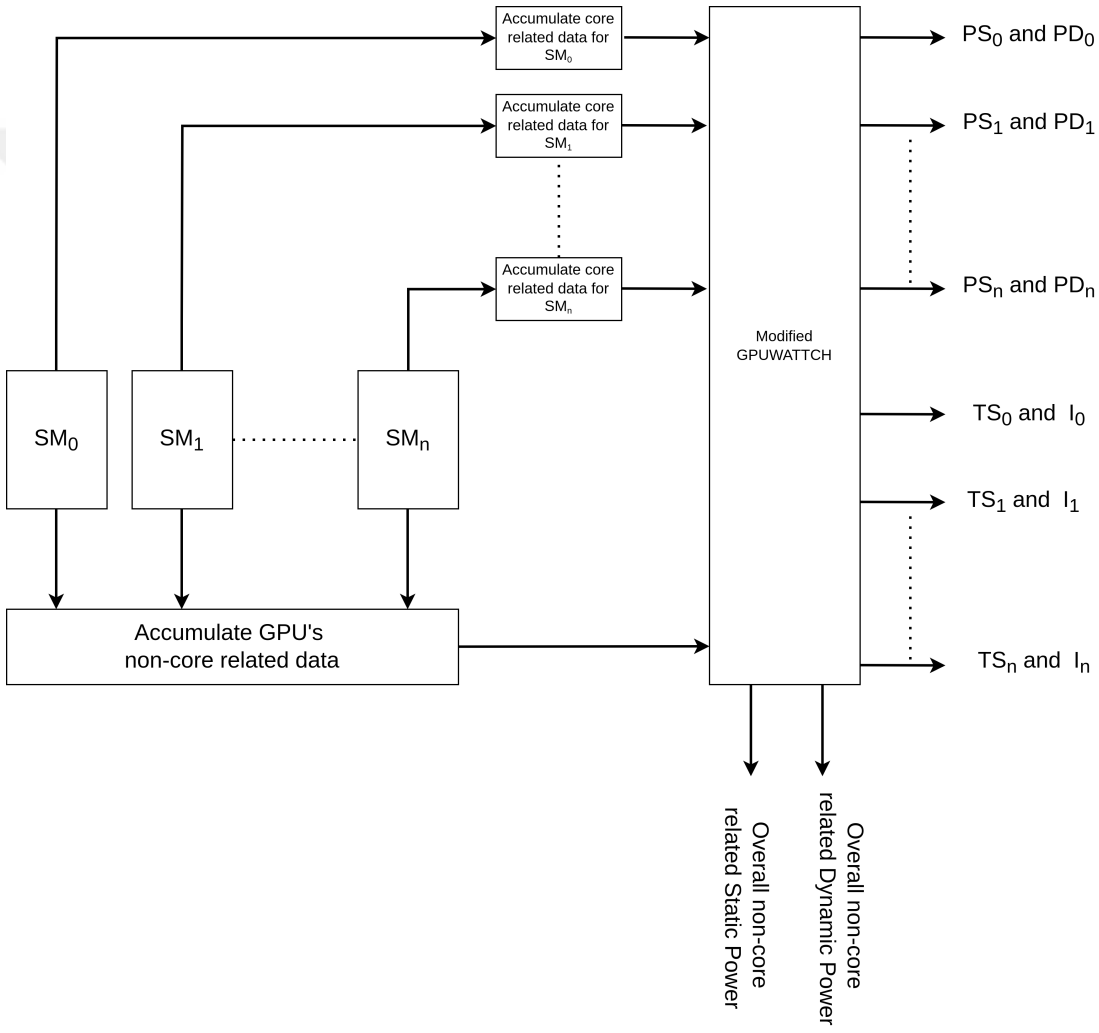

Figure 3.2: Our approach to derive the details of a specific interval. PS_i: Static Power for SM_i, PD_i: Dynamic Power for SM_i, TS_i: Stall Time for SM_i, I_i: Total number of committed instructions for SM_i.

time overlapped with memory accesses is considered as pure computation time. Equation 3.1 decomposes the interval time into stall time and computation time.

$$T = TS + K \times I \tag{3.1}$$

- T: Interval time,
- TS: Stall time,
- I: Total number of instructions executed in the interval,
- K: Average execution time of one instruction excluding the waiting time for memory response due to cache misses. It can be calculated as follows:

$$K = (T - TS)/I. \tag{3.2}$$

We record TS and I for each interval and calculate K for each SM. We use these data to predict the number of instructions executed in the following intervals with a new frequency. Let us assume the current execution is at i^{th} interval, and we want to predict the number of executed instructions at $(i+2)^{th}$ interval. Equation 3.3 models the execution time of the i^{th} recorded interval. Whereas, Equation 3.4 shows the predicted values for the $(i+2)^{th}$ interval. Note that, as shown in Figure 3.3, we collect the actual execution statistics in interval i, which

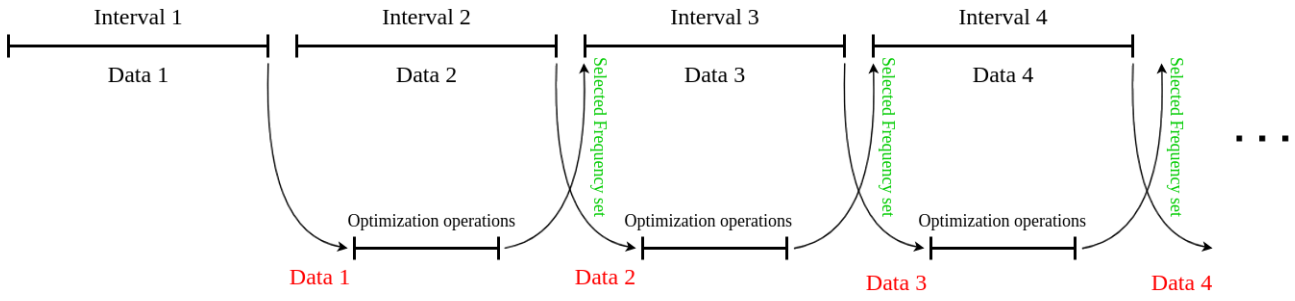

Figure 3.3: Interval-based frequency adjustment used in our approach.

are used in our model during the interval $(i+1)$. Decisions made during interval $(i+1)$ are applied during interval $(i+2)$. We choose such an approach to avoid performance overhead.

$$T_i = TS_i + K_i \times I_i \tag{3.3}$$

$$T_{i+2} = TS_{i+2} + K_{i+2} \times I_{i+2} \tag{3.4}$$

Since we have set the sampling period to 700 cycles with the default core's frequency of 700MHz, the interval time is constant throughout the execution of the application; therefore: $T_{i+2} = T_i$. Assuming that application behaves similarly, meaning that the same instructions with the same memory accesses and execution times will be executed in the $(i+2)^{th}$ interval. Based on this assumption, we can say that by changing the SM's frequency, K will change proportionally with $\frac{f_i}{f_{i+2}}$. f_i is the current frequency, and f_{i+2} is the new frequency we are trying to optimize. In this representation, we use the α term to represent the ratio of the two frequencies. More specifically,

- $\alpha = \frac{f_{i+2}}{f_i}$,
- $K_{i+2} = \frac{K_i}{\alpha}$.

Since the memory frequency is the same, the memory latency will not change. So, TS in $(i+2)^{th}$ interval will just depend on the number of instructions to be executed in the $(i+2)^{th}$ interval: $TS_{i+2} = \frac{I_{i+2}}{I_i} \times TS_i$. For the new interval, the number of executed instructions can be expressed using Equations 3.3 and 3.4 as follows:

$$TS_i + K_i \times I_i = TS_i \times \frac{I_{i+2}}{I_i} + \frac{K_i}{\alpha} \times I_{i+2} \tag{3.5}$$

The predicted number of instructions in the new interval is:

$$I_{i+2} = \frac{\alpha \times I_i \times (TS_i + K_i \times I_i)}{TS_i \times \alpha + K_i \times I_i} \tag{3.6}$$

For the simplicity of our performance and optimization models and to decrease the cost of optimization operations, we do not consider the computation time overlapped with the memory accesses in our performance model.

3.3 Power Model

Using the modeled simulator, we calculate the power consumption in each interval, for both dynamic and static power. The dynamic power changes proportionally with $f * V^2$, while static power changes with V^2 [19].

$$Power_{GPU} = Power_{Dynamic} + Power_{Static} \tag{3.7}$$

Including the voltage effects on the power in the optimization models increases the complexity of the operations. This will add extra overhead while performing the optimization decisions. Hence, for simplicity, in optimizing the new frequency, we do not consider the effect of the voltage on static and dynamic power. Instead, we use it in calculating the previous interval's power.

- $P_{Dynamic_{i+2}} = \alpha \times P_{Dynamic_i}$
- $P_{Static_{i+2}} = P_{Static_i}$

Based on above expressions, power consumption for the GPU can be predicted as:

$$Power_{GPU_{i+2}} = Power_{Static_i} + Power_{Dynamic_i} \times \alpha \tag{3.8}$$

3.4 Power per Instruction Metric

We use power per instruction ($\frac{P}{I}$) as our metric, M, to minimize as shown in Expression 3.9. This metric accounts for the performance loss during DVFS optimizations. We use this metric instead of EDP and ED^2P; in our model, the memory access overlapped with the computation is considered as computation time. Therefore, the optimization model will be more restricted to decrease the frequency for power saving. Moreover, since the saved power using DVFS is significantly high, it justifies the minimal performance loss. Hence, using only energy as optimization metric ignores the performance loss in favor of power saving.

$$M = \frac{P}{I} = \frac{Power_{Static} + Power_{Dynamic}}{I} \tag{3.9}$$

Chapter 4

Analytical Model

We implement two optimization techniques using the power and performance models mentioned earlier. Firstly, we try to minimize power per instruction for each SM using an analytical approach. In parallel, we use a genetic algorithm model to reduce the overall power per instruction. Then, we choose the new frequency set according to whichever provides the lowest overall power per instruction. We use the genetic algorithm to account for the SMs conflict over the memory components and interconnection network using the fitness function. Since the genetic algorithm is a search based-method, it may be stuck in a local minimum. Therefore, we use the analytical model to ensure that the new frequency set at least considers the power per instruction for each SM. Our experiments show that the optimization models used in our framework add an energy cost of less than 1%. This energy overhead is negligible compared to the GPUs' total energy consumption and savings provided by our approach.

Using the Equations 3.6, 3.8, and 3.9, we obtain the power per instruction value M for the next interval as follows:

$$M_{i+2} = \frac{P_{i+2}}{I_{i+2}} = \frac{Power_{Static_i} + Power_{Dynamic_i} \times \alpha}{\frac{\alpha \times I_i \times (TS_i + K_i \times I_i)}{TS_i \times \alpha + K_i * I_i}} \tag{4.1}$$

Minimizing power per instruction for each SM:

$$\frac{\partial M_{i+2}}{\partial \alpha} = 0 \rightarrow \alpha = \sqrt{\frac{P_{Static_i} \times K_i \times I_i}{P_{Dynamic_i} \times TSi}} \tag{4.2}$$

α is ratio of the chosen best frequency to current frequency for the SM in hand.

$$f_{i+2}^{SM_k} = \alpha^{SM_k} \times \quad f_i^{SM_k} \tag{4.3}$$

For many intervals, $f_{i+2}^{SM_k}$ is predicted to be greater than the default frequency or less than the lowest possible frequency. Since these frequency levels are not available on the hardware, for these cases, we set the SM's frequency to the highest and lowest levels, respectively. For the other values of α, we set the new frequency to the closest available frequency.

Chapter 5

Genetic Algorithm Model

Alongside the analytical model, we have used a genetic algorithm to account for the effect of multiple SMs' requests for the memory and contention over the interconnection network. This is achieved by implementing a fitness function that calculates the total power per instruction over all the SMs. The high-level view of the proposed genetic algorithm model is shown in Figure 5.1.

In the first step shown in Figure 5.1, the initial population is created, and the chromosomes' respective fitness values are calculated. Then, the mutation and crossover are applied, and the next generation is created using the new children

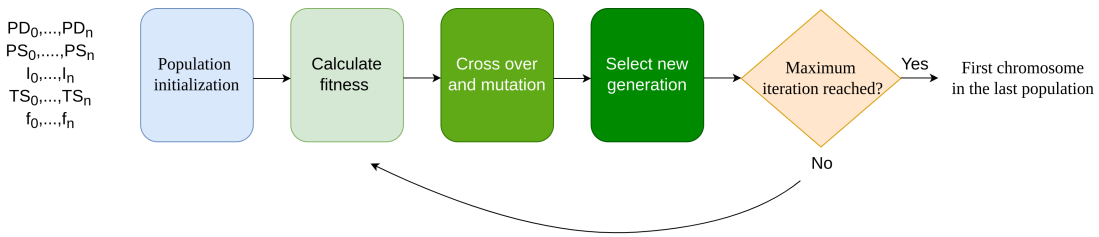

Figure 5.1: High-level view of the genetic algorithm model.

and a fraction of the fittest parents. We repeat the same process except for population initialization until the number of iterations is reached. In the end, the fittest chromosome is returned as the optimal frequency set. While it is possible to increase the number of iterations, in our experiments, we found it reasonable to select this value as 10. This is mainly due to the fact that we do not see a significant improvement after ten iterations in our tests, and increasing this number can potentially increase the execution time and energy consumption. The details of each step in our genetic algorithm based implementation are given below.

5.1 Population Initialization

The initial population is generated using the static and dynamic power, stall time, and the number of committed instructions per SM. The chromosomes used in our genetic algorithm are represented with the frequency set selected per SM. Each gene represents a potential new frequency for a specific SM. Figure 5.2 shows a chromosome used in our implementation.

The first chromosome is the frequency set of the previous interval. Other chromosomes' genes are initialized using a random function that chooses the frequency from the available frequencies in the hardware. We calculate the fitness function for each chromosome, which is expressed by the overall power per instruction. Without any changes, we directly transfer the top 20% of the fittest chromosomes to the next generation.

SM_0	SM_1	SM_2	SM_3	SM_4	SM_5	SM_6	SM_7	SM_8	SM_9	SM_{10}	SM_{11}	SM_{12}	SM_{13}	SM_{14}
frequency	frequency	frequency	frequency	frequency	frequency	frequency	frequency	frequency	frequency	frequency	frequency	frequency	frequency	frequency

Figure 5.2: A chromosome used in our genetic algorithm.

5.2 Fitness Function

Equation 5.1 shows the total number of executed instructions over all SMs for the next interval:

$$I_{i+2}^{TOTAL} = \Sigma_{k=0}^{N-1} I_{i+2}^{SM_k} \tag{5.1}$$

Note that, in the above equation, N is the number of available SMs. Using Equation 3.6 and 5.1, the predicted committed instructions given in Equation 5.1 now becomes:

$$I_{i+2}^{TOTAL} = \Sigma_{k=0}^{N-1} \frac{\alpha^{SM_k} \times I_i^{SM_k} \times (TS_i^{SM_k} + K_i^{SM_k} \times I_i^{SM_k})}{TS_i^{SM_k} \times \alpha^{SM_k} + K_i^{SM_k} \times I_i^{SM_k}} \tag{5.2}$$

Furthermore, using Equation 3.8 the total power for the next interval is calculated as follows:

$$Power_{i+2}^{TOTAL} = \Sigma_{k=0}^{N-1} (Power_{Static_i}^{SM_k} + \alpha^{SM_k} \times Power_{Dynamic_i}^{SM_k}) \tag{5.3}$$

Combining Equations 5.2 and 5.3, we can calculate the overall power per instruction for all the SMs, which is used as our fitness function:

$$M_{i+2}^{TOTAL} = \frac{\Sigma_{k=0}^{N-1} (Power_{Static_i}^{SM_k} + \alpha^{SM_k} \times Power_{Dynamic_i}^{SM_k})}{\Sigma_{k=0}^{N-1} \frac{\alpha^{SM_k} \times I_i^{SM_k} \times (TS_i^{SM_k} + K_i^{SM_k} \times I_i^{SM_k})}{TS_i^{SM_k} \times \alpha^{SM_k} + K_i^{SM_k} \times I_i^{SM_k}}} \tag{5.4}$$

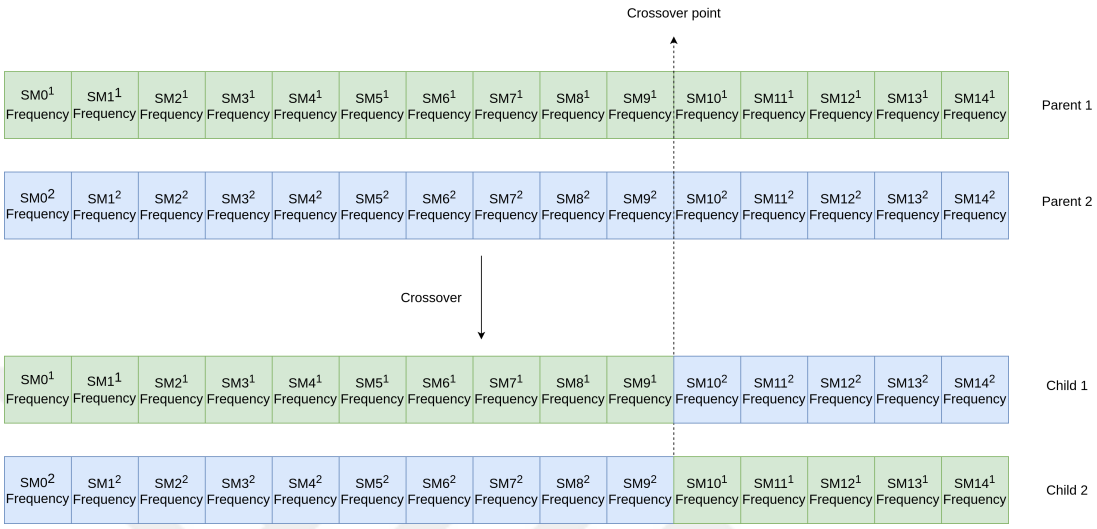

Figure 5.3: Crossover operations used in the genetic algorithm.

5.3 Crossover

For applying crossover, we pick two parents from the initial population using weighted probabilities. The chromosomes with lower fitness values have a higher chance of being selected. We use single-point crossover in our model, which refers to choosing a random gene in both parents and swapping the genes between them to generate new chromosomes. We repeat the crossover to generate 40% of the new generation's chromosomes. Figure 5.3 is an illustration of the crossover technique. The purpose of crossover is to use the combination of the fittest parents to explore their neighboring space, hoping that the new chromosome has a lower fitness value.

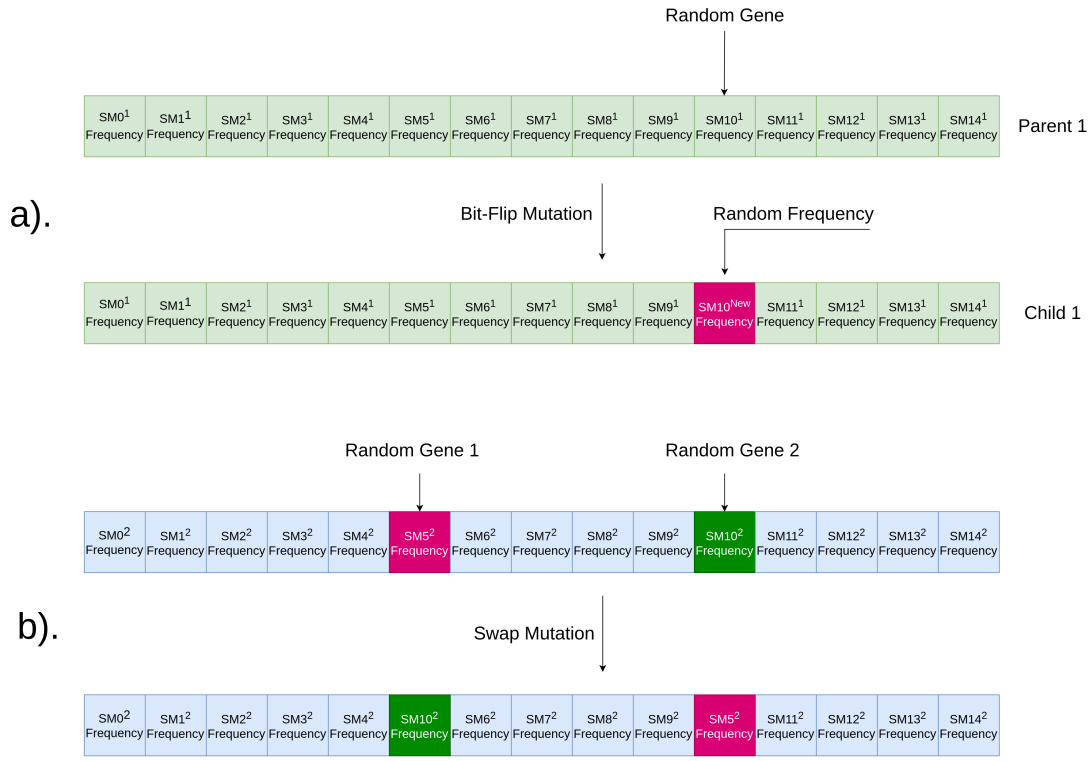

Figure 5.4: Mutation operations used in the genetic algorithm. (a) Bit Flip mutation, and (b) Swap mutation.

5.4 Mutation

After generating the new chromosomes using crossover, we apply the mutation technique. Firstly, two parents are chosen based on weighted probabilities. Recall that this way, chromosomes with lower fitness values have a higher chance of generating new chromosomes. Then we apply Bit-Flip and Swap mutations on these chromosomes. In Bit-Flip mutation, one gene is randomly selected, and the value for that gene is updated using a random function. The new value is chosen from the frequencies that the hardware supports. In Swap mutation, we randomly select two genes and swap their values. We apply these two mutation techniques to the chosen parents to generate 40% of the new populations' chromosomes. Figure 5.4 shows Bit-Flip and Swap mutation operations. The mutation technique is usually done to escape the local minimum using the fittest chromosomes and randomly changing some of their genes.

5.5 Hyper-parameters

The hyper-parameters used in our genetic algorithm model are shown in Table 5.1. We have fine-tuned these values through experimental evaluations. The mutation rate is selected as 0.4. We observed that the genetic algorithm converges to a local minimum for lower mutation rate values without proper exploration of the search space. Moreover, higher mutation rate values decrease the probability of convergence. The crossover rate is also set to 0.4 to converge the search to an optimal solution. The remaining 20% of the new population's chromosomes are the 20% of elite chromosomes of the previous population. The number of iterations is set to 10, and we generate ten chromosomes in each iteration. As explained before, increasing the number of iterations is possible, but we fine-tuned these values based on the experiments. Our experiments show that further increasing the number of iterations does not significantly improve. Furthermore, this increase can potentially cause additional execution time and energy overheads.

Table 5.1: Fine-tuned hyper-parameters of the genetic algorithm.

Hyper-parameter	Value and Description
Population size	10
Number of iterations	10
Crossover method	Single-point crossover
Crossover rate	40%
Mutation method	Bit-Flip and Swap
Mutation rate	40%
Elite count rate	20%

5.6 Hybrid Model

The new frequency sets given by the analytical model and genetic algorithm are evaluated to check which one has less value for overall power per instruction. Accordingly, interval $(i+2)^{th}$ will run with the selected frequency set. Figure 5.5 shows the hybrid model used to find the best frequency set for GPUs' SMs for each interval.

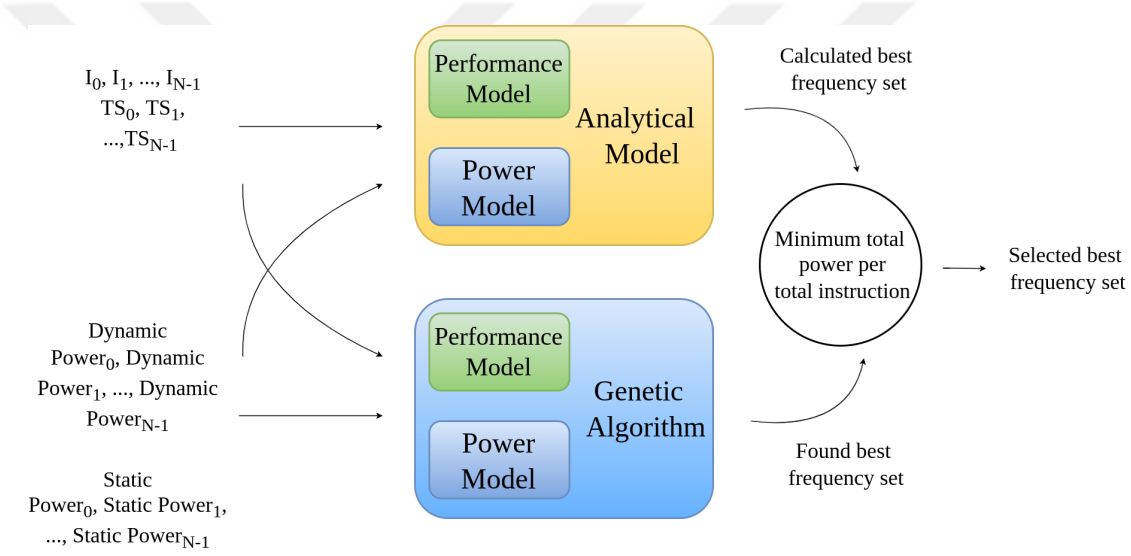

Figure 5.5: Hybrid model to find the best frequency set.

While executing instructions on the GPU, some of the SMs may require higher number of memory accesses, creating congestion in the interconnection network. Therefore, the memory accesses for other SMs will also have longer latency. Using the total power per instruction in the genetic algorithm's fitness function ensures that the new frequency set considers this effect. However, since the genetic algorithm is a search-based optimization technique, it can be stuck in the local minimum and propose a non-optimal frequency set. Using an analytical model alongside the genetic model guarantees that the power per instruction of each SM is minimized.

Chapter 6

Experimental Evaluation

6.1 Setup

As explained before, we use GPGPUSIM as our test platform since it is impossible to adjust frequencies for SMs individually in the hardware. Although our approach can work with any GPU architecture, we tested with NVIDIA's GTX 480 GPU in order to be able to compare our results with previous efforts. NVIDIA GeForce GTX 480 has 15 SMs, and seven different frequency levels, ranging from 100MHz to 700MHz. The default frequency is set to the highest value. The voltage of the hardware is set to $1\,V$ by default. The lowest voltage is $0.55\,V$ which corresponds to the lowest frequency value. It is well-known that voltage changes linearly with frequency [15][43]. Therefore, we consider a linear relation between the SM's frequency and the voltage.

For the evaluation of our model, we use the Rodinia benchmark [20]. Rodinia is one of the most commonly used benchmarks in GPU performance evaluation. It includes several kernels, where each benchmark stresses a different part of GPU architecture. We use 16 kernels from the Rodinia benchmark suite to test our approach. The kernels are selected to have a good representation of possible GPU applications with different execution properties.

Kernel	Summary	No. executed instructions	No. intervals
BFS	Breadth-First Search	2.30*1e6	1081
CFD	Computational Fluid Dynamics	5.37*1e7	171295
Kmeans	k-means Clustering	1.49*1e6	2467
LCY	Leukocyte Tracking	6.18*1e8	151308
LUD	LU Decomposition	5.24*1e8	64316
NN	Neural Network	7.60*1e8	138934
SR	Speckle Reducing Anisotropic Diffusion	2.14*1e8	35472
SC	Stream Cluster	3.49*1e8	145558
HW	Heartwall	6.16*1e8	72098
HS	Hotspot	9.77*1e8	77781
MUM	MUMmerGPU	1.68*1e7	1872
B+tree	b+tree	1.29*1e7	2142
Backprop	Backpropagation	2.94*1e6	294
Gaussian	Gaussian Elimination	2.73*1e7	9118
LavaMD	LavaMD	5.09*1e8	103710
Path	Pathfinder	3.48*1e8	145558

Table 6.1: Characteristics of the tested benchmarks. The first column gives the name, the second column gives a brief summary of the benchmark, the third column shows the total number of executed instructions, and the last column is the total number of sampling intervals.

Table 6.1 lists the characteristics of the tested benchmarks. More specifically, the first column gives the name; the second column summarizes the benchmark. The third column shows the total number of executed instructions, and the last column shows the total number of sampling intervals.

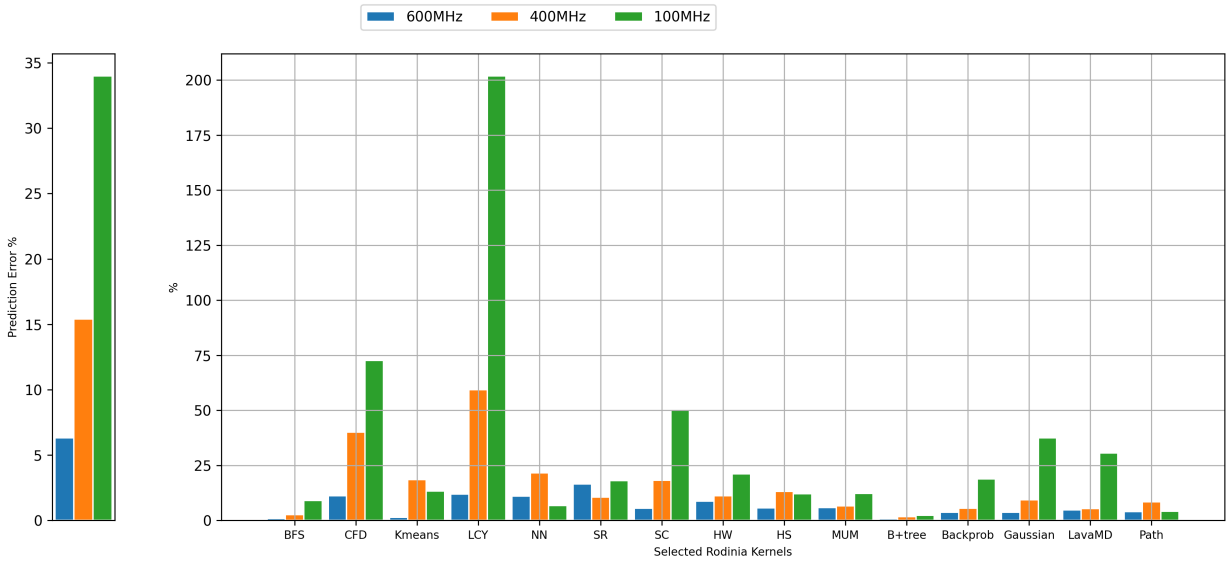

Figure 6.1: Mean absolute percentage error prediction of stall time for 15 Rodinia kernels across all GPU SMs. There are three tested frequencies (600MHz, 400MHz, 100MHz) for each of which the kernels are executed for 20μs. The results are compared to the baseline with 700MHz for the core's frequency.

6.2 Performance Prediction Error

We run our kernels for 20µs with four different core frequency settings. The data of the default frequency is used to predict the stall time and the number of executed instructions in other frequency levels. Figure 6.1 shows the mean absolute percentage error of the predicted stall time, whereas Figure 6.2 shows the mean absolute percentage error of the number of executed instructions using the recorded values across 15 SMs for all 15 kernels. Among the 16 kernels, LUD has one SM working during the 20µs period, which is why it is not included in the error results. The prediction error of the stall time and the number of executed instructions vary with the target frequency. For 600MHz as the core frequency, the average prediction error is 6.34% for stall time and 6.25% for the number of executed instructions. With lower frequencies, the prediction error is increasing, which is understandable as the number of executed instructions in the defined execution period is decreasing, and the similarity between instructions is declining. Among all the kernels, LCY shows a very high prediction error for the

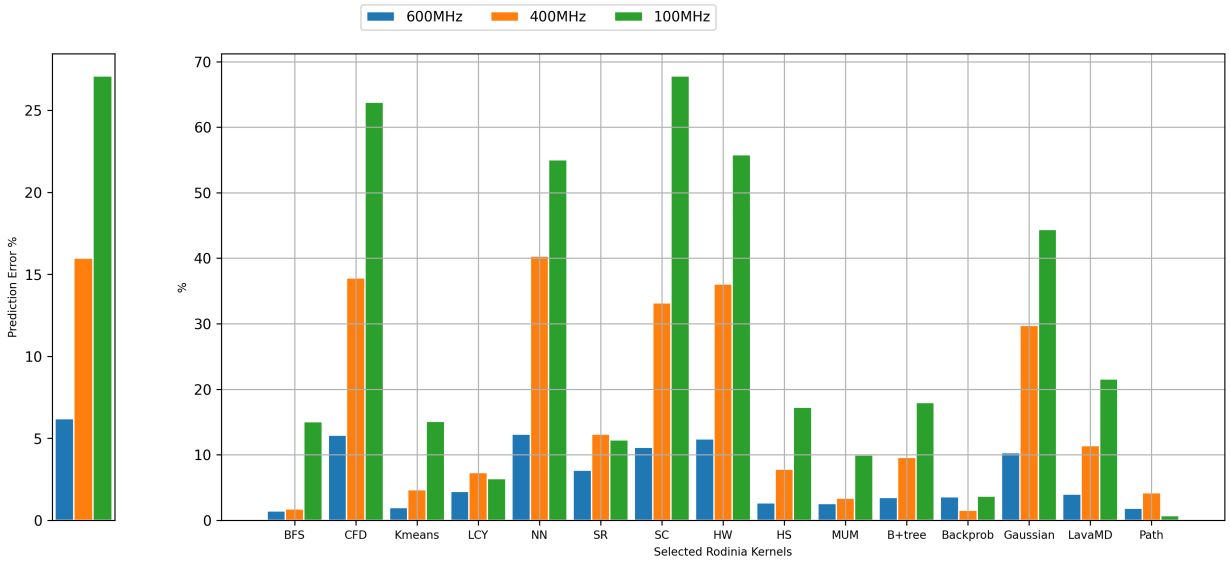

Figure 6.2: Mean absolute percentage error prediction of executed instructions for 15 Rodinia kernels across all GPU SMs. There are three tested frequencies (600MHz, 400MHz, 100MHz) for each of which the kernels are executed for 20μs. The results are compared to the baseline with 700MHz for the core's frequency.

stall time. We believe this is due to the inaccuracy of our model in capturing the computation overlapped with memory latency. The same applies to the high prediction error of the number of executed instructions for CFD, NN, HW, and Gaussian.

On average, our model has 18.58% and 16.48% mean absolute percentage error across all kernels on stall time and the number of executed instructions, respectively. Note that, our baseline frequency setting is 700MHz and we test with three different frequency settings over the default core frequency. More specifically, the frequencies are set to 1/7 (100MHz), 4/7 (400MHz), and 6/7 (600MHz) of the default frequency.

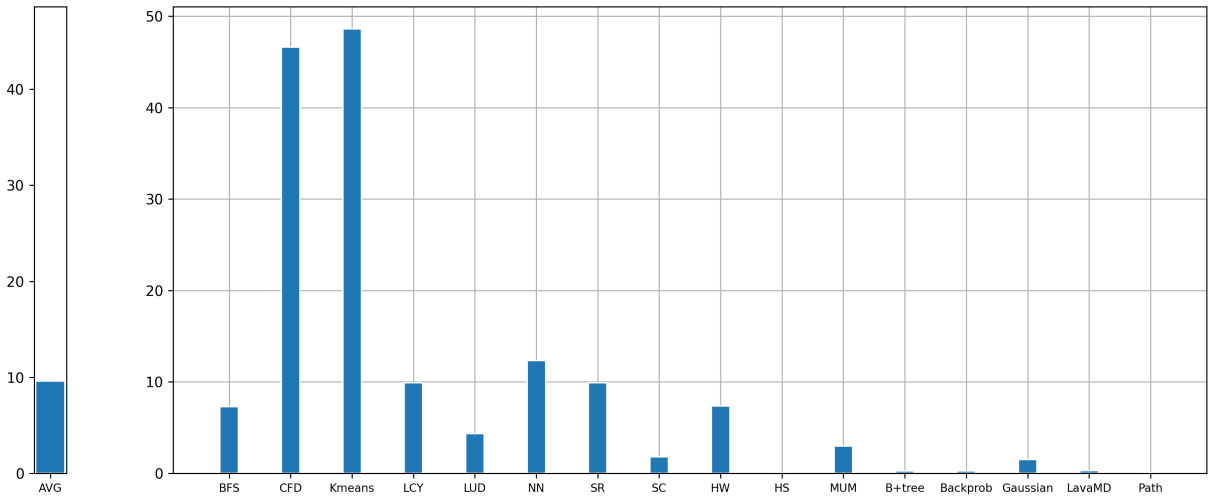

Figure 6.3: Percentage energy savings using the combination of analytical model and genetic algorithm for the tested benchmarks.

6.3 Experimental Results

Figure 6.3 shows the total energy savings with our approach. The average energy savings obtained using our approach is around 10%, which is successful when the conservative nature of the technique is considered. More specifically, as can be seen, Kmeans and CFD provide a very high percentage of energy savings. This is not due to the high stall time, but it is caused by the fact that Kmeans and CFD suffer from high load imbalance among SMs. While some SMs are still executing instructions, other SMs are done with their computations. In these cases, our approach sets the idle SMs' frequency to the lowest frequency level until they resume their execution. Some benchmarks, such as HS, B+Tree, Backprob, LavaMD, and Path, do not take advantage of the proposed scheme since the ratio of stall cycles to busy cycles is very small and, thereby, does not provide opportunities for energy savings.

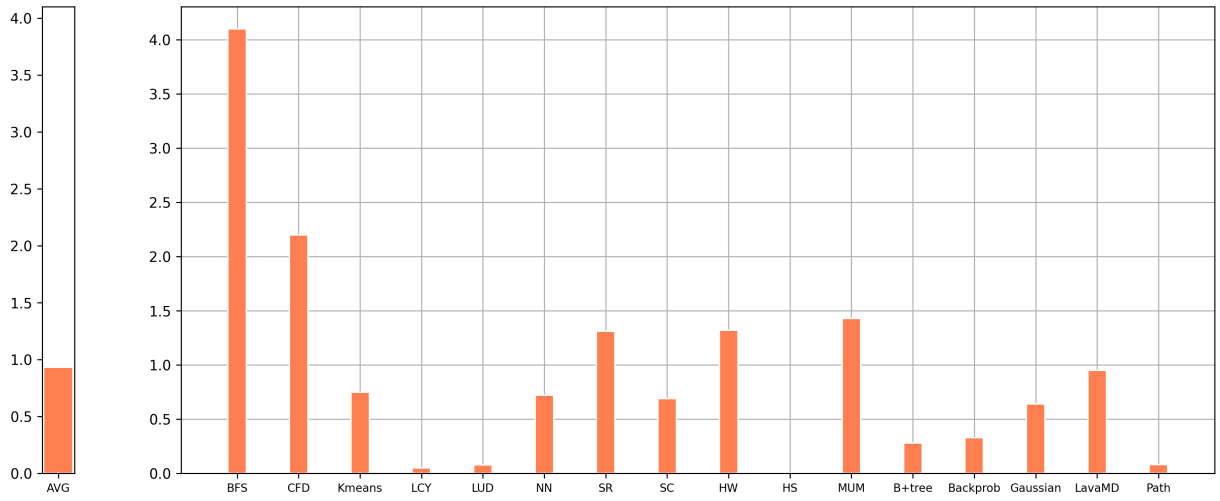

Figure 6.4: Execution time increase in percentage using the combination of analytical model and genetic algorithm for the tested benchmarks.

Figure 6.4 shows the performance loss using our online DVFS adjustment tool. The results are reported as the percentage increase in execution time based on the default frequency settings. As can be seen from this figure, the average performance overhead is less than 1%, which is acceptable when potential energy savings are considered. Based on the results given in Figure 6.4, the performance loss for BFS and CFD is relatively high compared to other kernels. For BFS and CFD, we observe that the performance overhead using the statistics collected at $(i)^{th}$ interval for frequency adjustment at $(i+2)^{th}$ interval, is higher than the performance loss when we use statistics of interval $(i)^{th}$ for frequency adjustment of interval $(i+1)^{th}$. Therefore, the high percentage of performance loss in these two kernels is due to the lack of similarity between non-consequent intervals. Moreover, the kernels with a high percentage of stall cycles provide better opportunities for decreasing the SMs' frequency for saving energy, thus slowing down the execution pipeline and having relatively higher performance overhead. In general, the difference between the performance loss of different kernels is possibly caused by either the lack of similarity between non-consequent intervals (i.e., $(i)^{th}$ and $(i+2)^{th}$ intervals) or it is because our optimization framework finds better

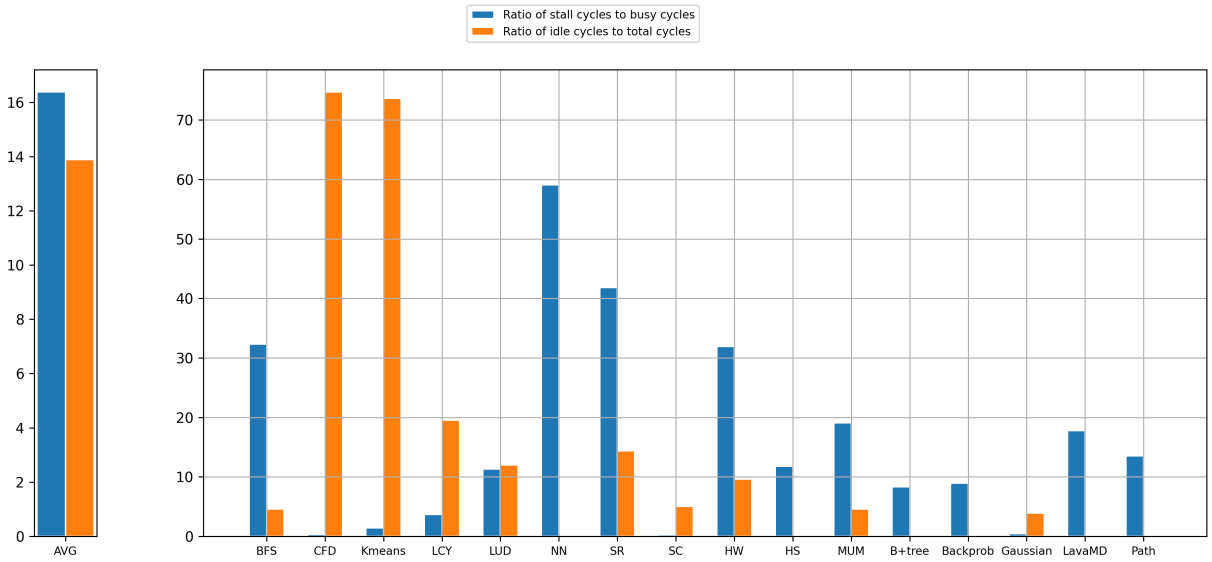

Figure 6.5: The ratio of stall cycles to total busy cycles and the ratio of idle cycles to the total cycles for the tested benchmarks (in percentage).

opportunities for decreasing the SMs' frequency for saving energy in kernels with a high percentage of stall cycles, than the kernels that run with maximum SMs' frequency.

The kernels with a high percentage of idle time benefit significantly from DVFS. Figure 6.5 shows the ratio of the total idle cycles of all the SMs to the total number of cycles. It also depicts the number of stall cycles to the total cycles in which SMs have instructions to execute. The ratio of idle cycles to the total number of cycles is very high for CFD and Kmeans. As explained earlier, the SMs' frequency and voltage are set to the lowest value in these cycles. Since there is no instruction to be executed in these cycles, decreasing the SMs' frequency does not affect the total execution time. Also, the ratio of the stall cycles to the total number of busy cycles is very low. Therefore, during busy cycles, the SMs' frequency is set to the highest level, thereby reducing the energy benefit of DVFS. So, for CFD and Kmeans, the saved energy is just due to setting the frequency and voltage to the lowest in idle cycles.

For BFS, NN, SR, and HW kernels, the ratio of the total number of stall cycles to the total number of busy cycles is high, and most of the saved energy is due to changing frequency while having a high percentage of stall cycles. In these cases, our online DVFS framework sets the frequency of the SMs to lower levels, which slows down the execution pipeline. However, since most of the time, the execution pipelines of different SMs are stalled, the performance loss is negligible when compared to the saved energy. The rest of the kernels experience less amount of stall cycles. In Figure 6.5, Backprop and B+tree show a very small percentage of stall cycles to the busy cycles. These kernels are always in a compute phase, and the execution pipelines of different SMs do not stall. Therefore, there is no opportunity to save energy using DVFS without a significant performance loss. LavaMD, Path, and HS show a very high percentage of stall cycles to overall busy cycles. However, the saved energy is not significant. We observe that the stall cycles must be at least 60% of the total busy cycles for the least performance loss in each interval. Although LavaMD, Path, and HS have a high value for memory operations on average, in each interval, the stall cycles do not exceed 60% of total busy cycles. In these cases decreasing the SMs' frequency causes a lot of performance loss. Since our model's optimization goal is to minimize power per instruction, it chooses not to reduce the frequency in such cases.

In Figure 6.6, we give the heatmap of the kernels' intervals. More specifically, we classify the sampling intervals into ten classes. In each interval, we calculate the ratio of stall cycles to the total busy cycles. If this ratio is less than 10%, that interval belongs to class one. If it is between 10% and 20%, that interval belongs to class two, and so on. Then, we sum the number of intervals that fall into each class and find the ratio of the intervals of each class to the total number of intervals. The kernels that experience a higher percentage of intervals that fall to the classes with stall ratio to total busy cycles of more than 60% benefit from DVFS with the least amount of performance loss.

Although the total stall cycles to total busy cycles are high for LavaMD, Path, and HS, they do not have a high percentage of intervals with stall cycles to busy cycles with a ratio greater than 60%. That is why our optimization model does not decrease the frequency to save power.

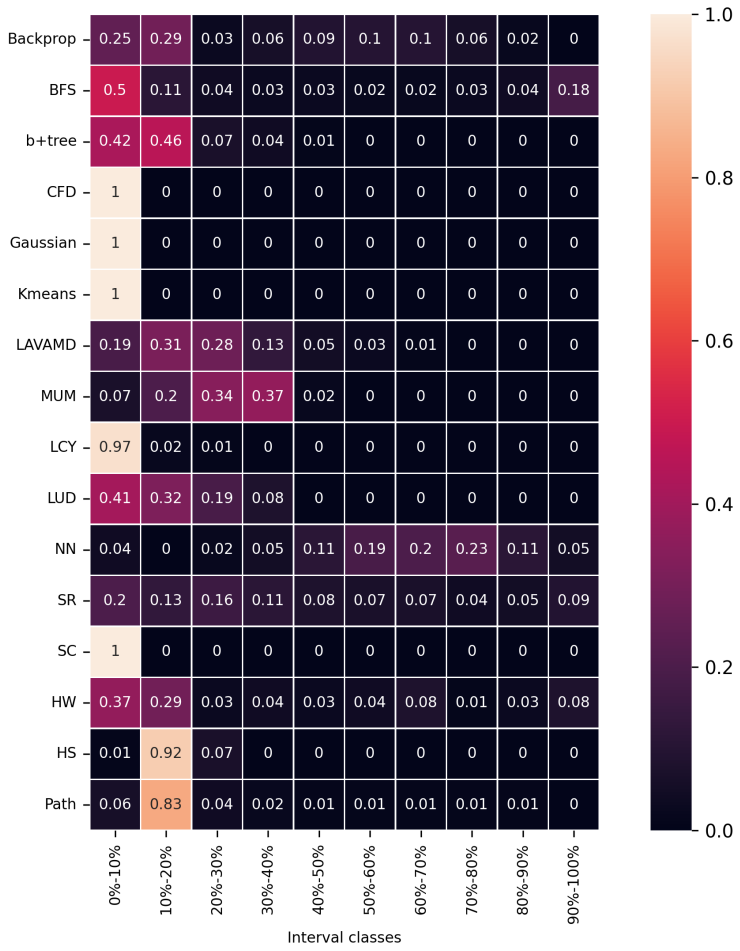

Figure 6.6: Heatmap of the kernels' intervals classification based on the ratio of stall cycles to busy cycles in each interval.

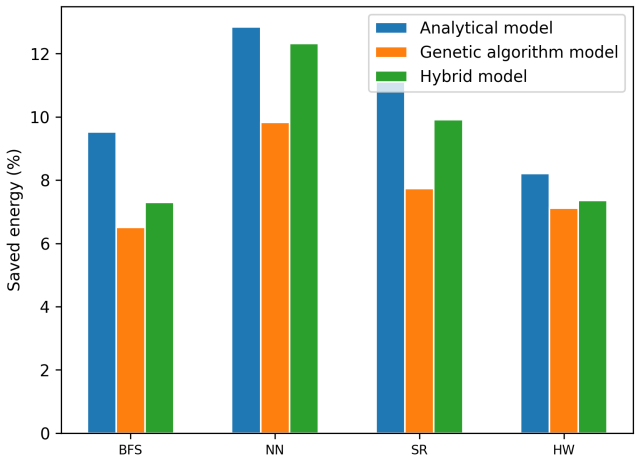

Figure 6.7: Average energy savings in percentage for the analytical, genetic algorithm, and hybrid models for BFS, NN, SR, and HW.

6.4 Sensitivity Analysis

We first compare the analytical model with the genetic algorithm model as part of the sensitivity analysis. Specifically, we test both models with selected benchmarks, namely, BFS, NN, SR, and HW. Energy and performance results are shown in Figure 6.7 and Figure 6.8, respectively. As shown in Figure 6.7, for all the kernels tested, using only the analytical model provides better energy results when compared to the genetic algorithm and the hybrid model. However, as depicted in Figure 6.8, the performance loss is also higher for the analytical model. More specifically, the performance loss for BFS is significantly high using only the analytical model. This is due to ignoring the effect of different SMs' memory access on the interconnection network and memory system. For the rest of the kernels, the performance loss for the analytical model is relatively low, and it is reasonable just to use the analytical model.

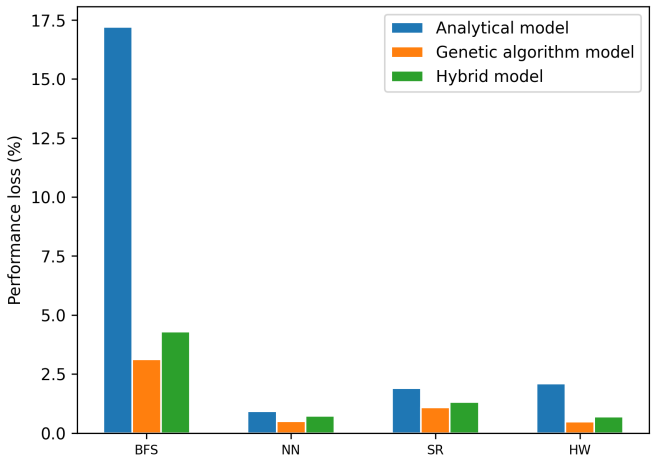

Figure 6.8: Average performance overhead in percentage for the analytical, genetic algorithm, and hybrid models for BFS, NN, SR, and HW.

On the other hand, as can be seen in Figure 6.8, the genetic algorithm causes less performance overhead. But, this comes with a price, where the energy savings provided by the genetic algorithm model is also lower when compared with the hybrid and the analytical models. For example, using only the genetic algorithm in NN has an energy saving of 9.83%, whereas it is 12.85% with the analytical model. We observe that the genetic algorithm takes a more conservative path while decreasing the SMs frequency in order to minimize the performance loss. This can possibly be due to considering different SMs' effects on each other's latency in the fitness function or caused by being stuck in the local minimum in the genetic algorithm. Overall, we believe that the hybrid model provides a reasonable combination of the genetic algorithm and the analytical model. This way, we can both decrease the performance overhead experienced in the analytical model and achieve better energy results when compared to the genetic algorithm model.

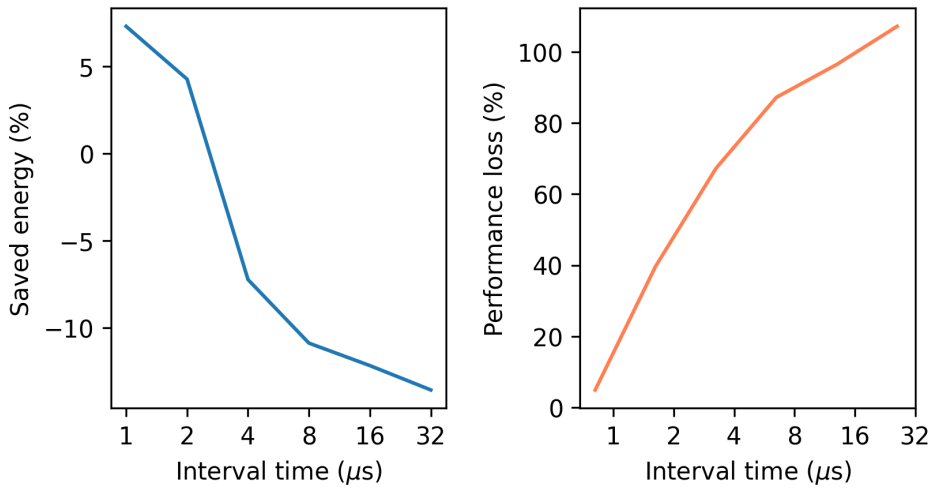

Figure 6.9: Sensitivity of the energy savings and the performance overheads to the interval time (in μs) for BFS.

We test BFS with six different interval times (in μs) to evaluate the effect of interval time on energy savings and performance overheads. Figure 6.9 shows the energy savings and the performance loss for the aforementioned sensitivity analysis. As seen in this figure, increasing the interval time decreases the potential for energy savings. This is mainly due to the fact that, with larger interval times, the similarity of the instructions between subsequent intervals diminishes. Therefore, our power and performance models' accuracy decreases, choosing non-optimal frequency sets in the optimization models.

In the literature [41], performing DVFS with a 1 μs interval time causes a 4ns performance overhead. We believe this 0.4% overhead is negligible. However, further reducing the interval time will potentially hurt the overall performance as DVFS overhead exceeds 1%. Therefore we used 1 μs as the minimum interval time in our experiments. As seen in Figure 6.9, the smallest value for the interval time (i.e., 1 μs) gives better results for both performance and energy. Since this period is equivalent to 700 cycles, it is well enough for the optimization operations required by our approach, thereby allowing a clear overlap with the actual execution in the current interval.

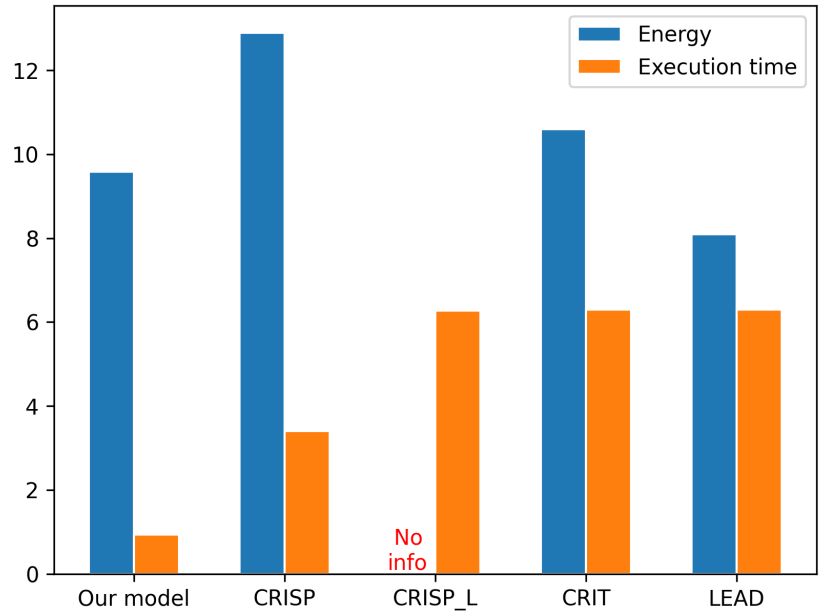

Figure 6.10: Comparison of our model with respect to CRISP and its respective models.

6.5 Comparison with Literature

We compare our results with CRISP [12], which has shown impressive accuracy in modeling GPUs' performance using an analytical model. In their model, they utilize the computation that overlaps with memory latency. They extended their approach using a more straightforward model named CRISP_L, which requires fewer changes in the hardware architecture. Figure 6.10 compares our approach and CRISP for average energy savings and performance overheads. Note that there is no information available in the CRISP paper about the energy savings of CRISP_L's model. All other models except ours use a single frequency level for all the SMs. On the other hand, our approach considers the load imbalance and different phase behavior in SMs execution time and adjusts the frequency for each SM individually. Overall, our model saves 9.59% energy on average with less than 1% performance loss, while CRISP's complicated model saves 12.9% energy with 3.4% performance loss on the selected kernels tested. We believe

that our approach provides a reasonable solution since it significantly reduces the performance overheads compared to other methods. Among the approaches tested, the least performance overhead is achieved by CRISP with a 3.4% value as opposed to our 1% performance overhead. Additionally, our energy savings are slightly less than the best approach (9.59% vs. 12.9%).

6.6 Discussion

This section discusses two possible extensions and improvements to our DVFS optimization framework. In our approach, we use a fixed time interval based approach in which the application's statistics are recorded during the execution. At the end of each interval, we predict the number of executed instructions and stall time for the next interval using these statistics, assuming that the subsequent intervals are computationally similar. Although this is a safe assumption, we propose applying DVFS optimizations with a deeper look using program analysis at the assembly code level for higher power and performance prediction accuracy.

We perform the predictions at the beginning of iterative sections of the assembly code to utilize the program characteristics better. More specifically, we use the loops inside the code, where the same instructions are executed multiple times. These code regions with the exact instructions are marked as iterative sections and detected as the first step in the proposed scheme. We can use a loop detection technique [44] to detect the iterative sections at run-time, where the executed instructions are evaluated to check if there is a branch instruction whose target program counter (PC) is less than the branch's PC. In such a case, potentially, the execution of a loop has finished, and a new iteration of the loop has started. Since the same instructions are highly likely to be executed in each iteration, the iterations' statistics are expected to be similar. Therefore, the power and performance model's accuracy will be potentially higher. We consider this approach a potential improvement to our online optimization model and plan to extend the framework to support such a feature.

The second improvement that can be applied to our optimization framework is task-based mapping of the threads to different SMs. As discussed in Chapter 1, different SMs can experience different run-time behavior due to load imbalance and diverse phase behaviors. Dynamic warp formation (DWF) is a technique to rearrange and reschedule threads assigned to a single SM in order to improve thread-level parallelism (TLP) inside the warps. This method reduces the performance loss caused by branch divergence as opposed to static warp formation in which, after a divergence point, the threads are serialized until they reach the convergence point. We propose to apply the same idea at the GPU level by dynamically scheduling the thread across different SMs based on their phase behavior and load characteristics. Therefore, the independent threads in one SM waiting for other threads to release the pipeline resources can be assigned to other idle SMs. We consider applying this method beside our DVFS optimization model to increase performance alongside energy savings.

Chapter 7

Conclusion

This work presents an online DVFS optimization framework that adjusts each SMs' frequency individually. The goal of per SM frequency adjustment is to remove the effect of load imbalance among different SMs on the DVFS energy saving. For this, we use an interval-based approach in which the application's statistics are collected for each SM separately after a certain number of cycles. The collected data is used to find the optimal frequency set for the next interval. We use a simple performance model to predict the number of executed instructions and stall cycles using the intervals' statistics. Besides, we use a power model adopted from the literature to predict the following intervals' power consumption. Using these models, we design an optimization model that can dynamically adjust SMs' frequency to reduce the power per instruction.

Our optimization model uses an analytical model and a genetic algorithm-based model in tandem. The analytical model tries to minimize the power per instructions for each SM. The genetic algorithm-based model accounts for the SMs' competition over the interconnection network and memory system by adopting total power per instruction as the fitness function. Since the genetic algorithm is a search-based model, it may be stuck in the local minimum. Using the hybrid model ensures that the proposed frequency set can minimize the power per instruction for each SM individually.

In our implementation, the prediction and optimization operations are performed in parallel with the execution of the next interval. We use this method to reduce the potential performance overhead of the proposed method. The selected frequency set is applied for the execution of the next interval. We believe that the extra power required by our optimization framework is negligible compared to GPU's power consumption. We tested our approach with a state-of-the-art GPU architecture and GPU benchmarks. Our results show that we can achieve 9.59% energy saving with a 0.95% performance loss on average.

Bibliography

[1] E. BUBER and B. DIRI, "Performance analysis and cpu vs gpu comparison for deep learning," in *2018 6th International Conference on Control Engineering Information Technology (CEIT)*, pp. 1–6, 2018.

[2] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, jul 2015.

[3] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1-2, pp. 19–25, 2015.

[4] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, and S. Kato, "Power and performance analysis of gpu-accelerated systems," in *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, (USA), p. 10, USENIX Association, 2012.

[5] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, and S. Kato, "Power and performance analysis of gpu-accelerated systems," in *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, (USA), p. 10, USENIX Association, 2012.

[6] F. Mendes, P. Tomás, and N. Roma, "Exploiting non-conventional dvfs on gpus: Application to deep learning," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 1–9, IEEE, 2020.

[7] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 18–28, 2005.

[8] S. Hajiamini, B. Shirazi, A. Crandall, and H. Ghasemzadeh, "A dynamic programming framework for dvfs-based energy-efficiency in multicore systems," *IEEE Transactions on Sustainable Computing*, vol. 5, no. 1, pp. 1–12, 2019.

[9] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "Coscale: Coordinating cpu and memory system dvfs in server systems," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 143–154, 2012.

[10] B. Acun, K. Chandrasekar, and L. V. Kale, "Fine-grained energy efficiency using per-core dvfs with an adaptive runtime system," in *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*, pp. 1–8, 2019.

[11] A. Mishra and N. Khare, "Analysis of dvfs techniques for improving the gpu energy efficiency," *Open Journal of Energy Efficiency*, vol. 04, pp. 77–86, 01 2015.

[12] R. Nath and D. Tullsen, "The crisp performance model for dynamic voltage and frequency scaling in a gpgpu," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 281–293, 2015.

[13] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao, "Dynamic load balancing on single- and multi-gpu systems," pp. 1–12, 01 2010.

[14] B. Gallet and M. Gowanlock, "Load imbalance mitigation optimizations for gpu-accelerated similarity joins," pp. 396–405, 05 2019.

[15] X. Mei, Q. Wang, and X. Chu, "A survey and measurement study of gpu dvfs on energy conservation," *Digital Communications and Networks*, vol. 3, no. 2, pp. 89–100, 2017.

[16] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás, "Dvfs-aware application classification to improve gpgpus energy efficiency," *Parallel Computing*, vol. 83, pp. 93–117, 2019.

[17] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 564–576, 2015.

[18] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, "Gpgpu power modeling for multi-domain voltage-frequency scaling," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 789–800, 2018.

[19] J. Nunez-Yanez, K. Nikov, K. Eder, and M. Hosseinabady, "Run-time power modelling in embedded gpus with dynamic voltage and frequency scaling," in *Proceedings of the 11th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures / 9th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM'2020, (New York, NY, USA), Association for Computing Machinery, 2020.

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

[21] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, 2009.

[22] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: methodology and empirical data," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pp. 93–104, 2003.

[23] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance," 12 2015.

[24] Y. Abe, H. Sasaki, S. Kato, K. Inoue, M. Edahiro, and M. Peres, "Power and performance characterization and modeling of gpu-accelerated systems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 113–122, 2014.

[25] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of gpu kernels using performance counters," in *International Conference on Green Computing*, pp. 115–122, 2010.

[26] Q. Wang and X. Chu, "Gpgpu performance estimation with core and memory frequency scaling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 12, pp. 2865–2881, 2020.

[27] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás, "Modeling and decoupling the gpu power consumption for cross-domain dvfs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 11, pp. 2494–2506, 2019.

[28] L. Wang, M. Jahre, A. Adileho, and L. Eeckhout, "Mdm: The gpu memory divergence model," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1009–1021, 2020.

[29] J. Chen, B. Li, Y. Zhang, L. Peng, and J.-k. Peir, "Statistical gpu power analysis using tree-based methods," in *2011 International Green Computing Conference and Workshops*, pp. 1–6, IEEE, 2011.

[30] A. Sethia and S. Mahlke, "Equalizer: Dynamic tuning of gpu resources for efficient execution," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 647–658, 2014.

[31] m. Xiaohan and M. Dong, "Statistical power consumption analysis and modeling for gpu-based computing," 05 2012.

[32] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ati gpu: A statistical approach," in *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*, pp. 149–158, 2011.

[33] R. Sen and D. A. Wood, "Gpgpu footprint models to estimate per-core power," *IEEE Computer Architecture Letters*, vol. 15, no. 2, pp. 97–100, 2016.

[34] S. Hong, *Modeling performance and power for energy-efficient GPGPU computing*. PhD thesis, Georgia Institute of Technology, 2012.

[35] S. Hong and H. Kim, "An integrated gpu power and performance model," vol. 38, pp. 280–289, 06 2010.

[36] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang, "Run-time technique for simultaneous aging and power optimization in gpgpus," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2014.

[37] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás, "Gpu static modeling using ptx and deep structured learning," *IEEE Access*, vol. 7, pp. 159150–159161, 2019.

[38] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 673–686, 2013.

[39] P. Zou, A. Li, K. Barker, and R. Ge, "Indicator-directed dynamic power management for iterative workloads on gpu-accelerated systems," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 559–568, 2020.

[40] 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," *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[41] S. Bharadwaj, S. Das, K. Mazumdar, B. Beckmann, and S. Kosonocky, "Predict; do not react for enabling efficient fine grain dvfs in gpus," 2022.

[42] G. Keramidas, V. Spiliopoulos, and S. Kaxiras, "Interval-based models for run-time dvfs orchestration in superscalar processors," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, (New York, NY, USA), p. 287–296, Association for Computing Machinery, 2010.

[43] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, "Gpgpu power modeling for multi-domain voltage-frequency scaling," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 789–800, 2018.

[44] J. Tubella and A. Gonzalez, "Control speculation in multithreaded processors through dynamic loop detection," in *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*, pp. 14–23, 1998.