UNIVERSITY OF CALIFORNIA
RIVERSIDE

Characterizing Dynamic Frequency and Thread Blocking Scaling in GPUs: Challenges and Opportunities

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Marcus Chow

March 2018

Thesis Committee:

Professor Daniel Wong, Chairperson
Professor Laxmi Bhuyan
Professor Nael Abu-Ghazaleh

The Thesis of Marcus Chow is approved:

Committee Chairperson

University of California, Riverside

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Present day data centers, or warehouse scale computers (WSC), require significant compute power to accelerate workloads such as Deep Learning (DL) and High Performance Computing (HPC). However, data center energy consumption is a first-order limiting factor. GPUs are popular accelerators for the aforementioned type of workloads due to its energy efficient properties [1], enabling order-of-magnitude improvements in operations per watt compared to traditional multi-core CPUs.

With the introduction of Elastic GPUs in Amazon EC2 instances and Azure N-series, it brings us to question "How useful are the GPUs in the Cloud?". As accelerators, GPUs are often compared against FPGAs, which consume vastly less power and can accelerate many types of regular workloads. For GPUs to be useful in data centers they need to be energy efficient. By far, the performance and energy efficiency of GPUs in a data center environment have not been well explored. This has been a challenge because the GPU are slave devices that are controlled through drivers instead of the operating system. Nvidia's

GPUs use their proprietary CUDA drivers and runtime API that allow only course grain control. Previous work treat GPUs as black box accelerators, increasing energy efficiency and utilization through Quality of Service predictions [2, 3].

In contrast to prior works which maximizes throughput and utilization, we instead aim to improve energy efficiency through exploiting the limited power management polices on GPUs. In this work, we will show that current power polices implemented in real GPUs are load agnostic and will run at the highest frequency, aiming to exploit thermal headroom for performance-only gains. This leaves out any extra power savings through workload dependent policies. Based on our observations, we propose a runtime frequency scaling technique that aims to improve the energy efficiency of GPUs that is currently lacking in existing GPUs. In addition, we will further explore opportunities to further improve energy efficiency through utilizing request parallelism as a knob to control the amount of co-location of request possible.

In this paper, we perform power management characterization of real GPUs, tail latency analyses and explore possible modifications to save power and improve throughput. In addition, we will highlight the opportunities and challenges that we encounter in regards to implementing more efficient power management policies on real GPUs. Our contributions are:

- Evaluating Power Characteristics of GPUs: Our Dynamic Frequency Scaling experiments show a non linear diminishing relationship between frequency and power, limiting potential power savings.
- Thread Block Scaling and its effect on tail latency: We do preliminary tests on Thread

Block Scaling to improve throughput and per-request energy efficiency.

- Propose simple power savings policy for GPUs: As a proof of concept, by scaling frequency with respect to current server utilization, we make theoretical calculations that show a 1.6x improvement in power savings.

Chapter 2

Background

In this chapter, we will provide the background for the GPU Execution Model and their role in Data Centers. We will also define Tail Latency and Quality of Service and their use in previous research, as well as, historical use of Dynamic Frequency and Voltage Scaling. Our policy focuses techniques to reduce power consumption and increase throughput in GPUs without violating the Quality of Service

2.1 GPU Execution Model

GPUs follow a Single Instruction Multiple Thread (SIMT) execution model running on thousands of cores. 32 threads are grouped together to execute instructions in the form of a *Warp*. Warps consist of 32 threads that operate in a lock-step manner, where every thread within the warp must execute the same instruction. Should threads within a warp need to execute different instructions, the warp will diverge, with certain threads being deactivated if a certain instruction is not in its execution path, leading to under-

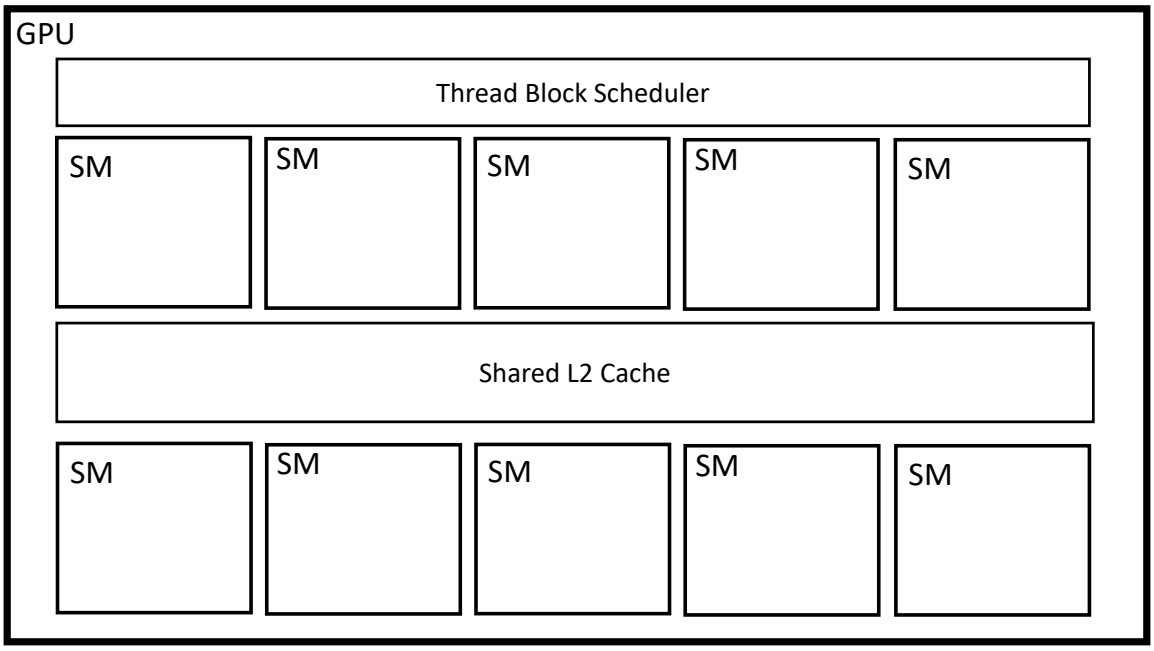

Figure 2.1: Example GPU architecture. Shows how a GPU is composed of multiple Streaming Multiprocessors. Where the Thread Block Scheduler, Schedules Thread Blocks to every SM

utilization in the hardware. As shown in Figure 2.1 Warps are then logically grouped into Thread Blocks and are mapped to hardware structures called Streaming Multiprocessors (SMs). Current GPUs contain, in the order of 20-30 SMs. Our work uses the Tesla V100, which contains 80 SMs. Kernels, which represents a program context, are then composed of multiple thread blocks.

GPUs are slave devices, which process computation that is offloaded from the CPU. The CPU is referred to as the host side and the GPU is called the Device side. Every request sent to the GPU involves three stages: 1) Data transfer from Host to Device, 2) kernel execution, 3) and data transfer from Device to Host. All three stages must be initiated on the host side. These stages can be overlapped by stages from other requests using different *streams*. Overlapping two or more streams, can increase overall throughput

but can hinder servicing time due to hardware contention (such as cache contention, or memory bandwidth sharing), which is a critical factor in data centers. Also, unlike the CPU, NVIDIA GPUs have a hard limit to the number of concurrent kernels it can handle. On current Pascal GPUs that limit is 128 concurrent kernels [11]. This significantly restricts the number of requests a device can handle. This means that, when the GPU is at full utilization, extra requests are filled into one or more work queues. As shown in Figure 2.2, the driver maps each software stream to one of the hardware work queues [27]. A queue executes kernels sequentially, however kernels from multiple queues may be executed on the same GPU concurrently, as long as there exist enough hardware resources (SM) to allow both kernels to execute concurrently.

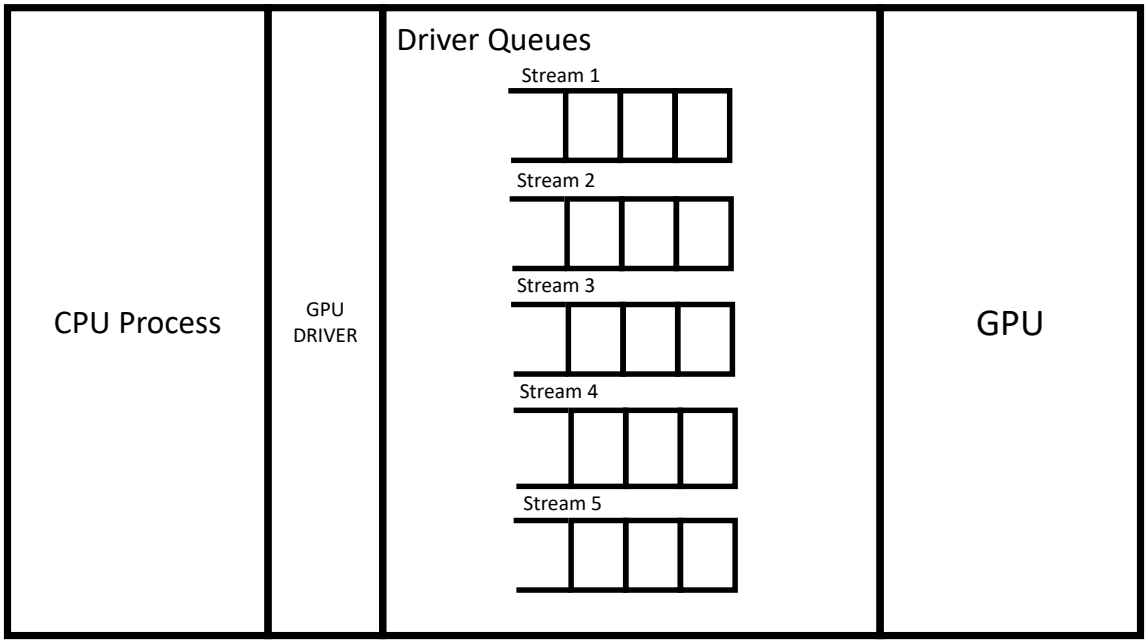

Figure 2.2: How the driver handles kernel calls from a CPU Process. Kernels from different streams are able to execute concurrently, however kernels from the same stream are pushed into Driver Queues and executed sequentially

2.1.1 Nvidia MPS

Nvidia Multi-Process Service(MPS) is designed as a way to colocate multiple processes to the same GPU, with the goal to increase overall utilization in High Performance Computing Environments [27]. The under laying service intercepts any kernel calls to the GPU from the processes and handles the scheduling of separate contexts by acting as an intermediate context. This allows the GPU to only need to operate on a single context and reduce overheads of collocating separate processes. MPS is also designed for Execution Resource Provisioning, which is the ability to limit the amount of resources a single process uses on the GPU to reduce overall contention. This contention is reduced both in the on card memory and within SMs. However, this partitioning is static and can only be assigned when launching the process. Our work aims to provide dynamic resource provisioning through thread block scaling and be able to modify resource limits based off of current server utilization and Quality of Service constraints.

2.2 Data Center Workloads

Data centers are notoriously energy inefficient, because they must be able to handle times of peak load, data centers are often provisioned for the worse case. As seen in Figure 4, the top chart shows a typical data center utilization over a period of time from Google's data center traces [30]. The utilization traces features periods of high load and low load. These times of under-utilization provide an opportunity to slow down an incoming request and save power. The main driver for increased GPU usage in data center is the exponential demand for more computational power. This growth comes from Cloud Services, such

as Amazon Web Services, and a growing reliance on Deep Learning. These Deep Neural Networks require a high number of computation for training and inference. DNN as a Service, [13] will continue to grow as more services rely on Deep Learning, and powers many applications from Intelligent Personal Assistants (such as Apple Siri, Amazon Echo, and Google Assistant) to Recommendation systems (such as movie recommendations in Netflix, or product recommendations in Amazon). For the purpose of this work, we mainly focus on inference workloads which powers much of the user-facing request workloads.

2.2.1 DjiNN and Tonic

Djinn and Tonic [13], is one such implementation of DNN as a Service.

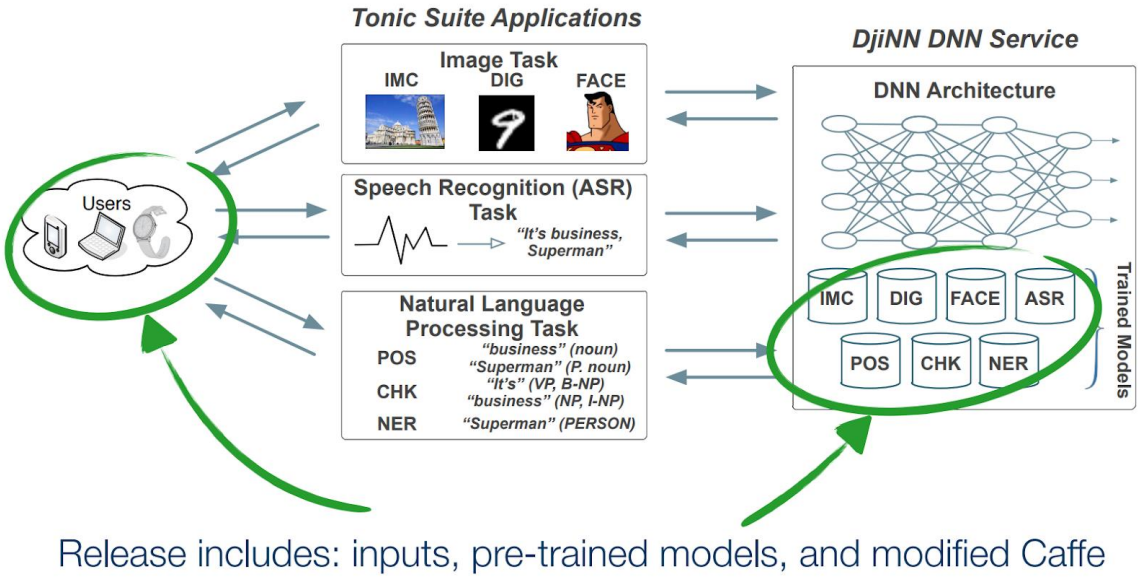

Figure 2.3: Overview othe Djinn and Tonic service. Adapted from [12]

In Figure 2.3, we show an overview of the DjiNN and Tonic suite. It is comprised

of two components, Djinn is the server who handles incoming requests and processes them on the GPU. It uses Caffe [15], a deep learning framework, for its DNN infrastructure. They also include their Tonic Suite, which comes with pretrained DNN networks for a multitude of tasks. For each service in the Tonic suite, the client performs some preprocessing on the data to a Djinn server format. After preprocessing, the client then sends the data along with the task name, which tells the server which network to run the data through. The suite consists of the following classes of tasks.

Image Task

The Image tasks are comprised of three separate services; Image Classification, Facial Recognition, and Digit Classification. Image Classification can predicate one thousand unique classes with a high accuracy using the AlexNet network [18]. Facial Recognition, is modeled after Facebook's DeepFace Network [29] and has an accuracy close to that of humans. Digit Recognition is used to classify hand written digits and was modeled using the MNIST Data set [19].

Natural Language Processing Task

NLP tasks take a string of text as an input and are design to extract specific semantic information from the text. The tasks include; Part-of-Speech Tagging, assigns whether a word is a verb or a noun, Word Chunking, labels a segment if it either is in the beginning or inside, and Name Entity Recognition, Labels nouns as a Person, Place, or Thing.

Automatic Speech Recognition

This final task is capable of decoding an audio file to text. This task primarily uses the framework provided by Kaldi [28]. This network requires the most preprocessing out of all the tasks, because it most generate feature vectors describing the speech in the input audio file.

2.3 Tail Latency

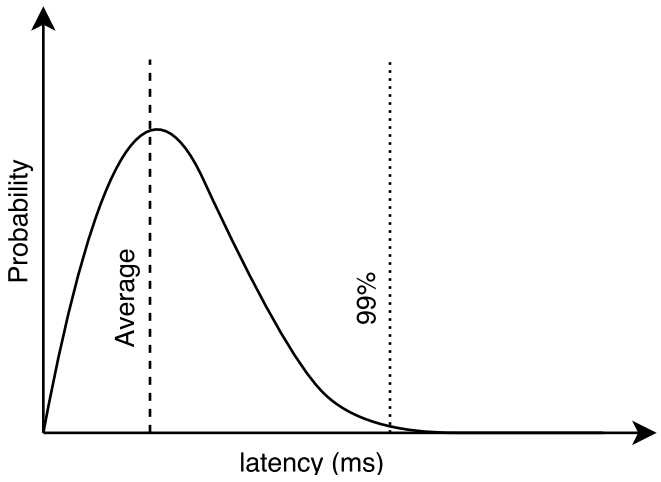

Figure 2.4: Example of Latency Distribution in Data Centers. Guaranteed latency is under the 99 percentile, but average latency is much lower than the tail. This gives some slack to slow down requests and save energy.

A common performance metric in latency-critical data center application is *tail latency*. Specifically, in this paper, we use *99th-percentile tail latency* which is defined to be the time where 99% of all incoming request completes. Tail latency is affected by many components in a data center. [5] describes why there exists such variability in data centers. Reasons include, sharing of resources, background daemons, active maintenance, queuing

overheads, power and thermal limits, garbage collection, and energy management. Prior research explore how these overheads and tail latency can be minimized [14, 21, 31]. In [21], they were able to accurately measure tail latency caused from interference in the form of background processes. A real-time scheduler is much more efficient at handling background processes than a normal priority based mechanism. Important to our work, they also find power savings techniques, applied in times of low utilization, can actually inflate tail latency. We aim to exploit this, to reduce power consumption. Figure 2.4 illustrates latency distribution common in data centers and how there is some slack between the average latency and the tail latency. To save power, previous papers employ different techniques to push the average latency closer to the tail, such as using DVFS [14, 16] , or Sleep states [4]. Now that GPUs are gaining popularity in data centers, we examine how to extend average latency and save power through frequency and thread block scaling techniques.

2.4 Related Work

2.4.1 Quality of Service Aware Dynamic Power Management

There have been many prior works addressing tail latency and Quality of Service guarantees in data center services. Adrenaline adds fine grain control over CPU voltage boosting to reduce the tail latency [14]. By shorting the tail latency, the overall efficiency in the data center is increased; in terms of both utilization and energy. They leveraged fine grain voltage control in modern CPU's which is not currently available in GPUs. To compensate, our work focuses on increasing utilization through concurrent request execution and thread block scaling.

Sleepscale, [22], offers a mechanism to control sleep sates dynamically and stay within Quality of Service constraints. Our work is also a power management runtime system, however, we focus on managing power with GPU specific attributes and characteristics.

Prophet [2] shows the main challenge of ensuring QoS when offloading computation to accelerators. Concurrent stream execution will interfere with each other and make accurate QoS guarantees difficult to keep. They use tail latency predictions to identify safe streams to execute without effecting the execution of others. To increase utilization they co-locate batch applications with latency-critical services. Our work focuses on collocation of latency-critical services. This provides a different challenge, as we must take into consideration the Quality of Service constraint with every workload.

CPU-Miser [7] describes a DVFS driven CPU runtime system. This technique incorporates a power management technique without any performance penalty. This system consists of a predictor which uses the feedback from performance monitor and scale the frequency and voltage accordingly.

While most of the work is limited to CPU-only systems, there is minimal exploration of DVFS scheduler in heterogeneous (CPU-GPU) systems. In [17] paper, Komoda et. al. propose a power capping technique for such heterogeneous systems by coordinating DVFS and task mapping on GPUs. They built an empirical model based off of workload profiling to predict the performance and power consumption when running different workloads.

One other paper [8] studies the effect of DVFS on performance and energy efficiency on a K20 GPU. Authors Ge et. al. chose matrix multiplication as the workload and ran it

with different input sizes and application clock frequencies.

The Survey and measurement study of GPU DVFS on energy conservation [24] provides in-depth analyses on dynamic voltage and frequency scaling on different workloads. Also, the experimental results exclaim the fact that frequency and voltage scaling depend on both GPU architecture and the application.

Further, all the related GPU DVFS papers we discussed here are based on desktop grade GPUs. These explorations seem futile since there has been remarkable advancement in server grade (Tesla) GPUs. Also, they motivate the need for advanced DVFS software techniques to improve energy savings without performance penalties.

2.4.2 Power Management techniques

Power management is essential to reduce the cost of maintaining a data center. [10] Multiple DVFS schemes have been proposed to reduce the cost and improve utilization and energy proportionality [4, 6, 23]. Rubik [16] controls CPU frequency by a statistical model to predict arrival distributions and without affecting tail latency. They use pre-computed target tail latency tables to allow fast predictions. Powerchief [31] uses dynamic boosting techniques to reduce bottlenecks in multi-stage services. In multi-stage service, requests get queued at each stage, requiring different boosting strategies per stage. Work in this area primarily focus on GPU power management schemes. However, none of them control power on the GPU side.

2.5 Dynamic Voltage and Frequency Scaling (DVFS)

Power consumption in a CMOS circuit is due to Static Power dissipation and Dynamic Power dissipation. Static power is the power consumed by the circuitry with no activity while Dynamic power is when the circuit is operational. Furthermore, static power consumption is almost negligible due to circuit design optimizations. From [9] Dynamic power in a CMOS transistor is expressed as:

$$P = CV^2f \tag{2.1}$$

In equation 2.1, C, V, and f represents Capacitance, Voltage, and operational frequency respectively.

Energy is the power consumed over time. Hence,

$$E = Pt \implies E = CV^2 \tag{2.2}$$

From equation 2.1 and 2.2, it can be clearly understood that changing frequency and Voltage impacts Dynamic Power consumption that further affects the energy consumed. DVFS techniques employ this relationship to derive energy savings with negligible performance penalty. Additionally, the above relationship helps in understanding that frequency scaling does not necessarily reduce energy consumption. This is because the reduced frequency corresponds to longer execution times thereby increasing energy consumption, motivating the need for more than just DVFS policy.

2.5.1 DVFS in GPUs

It has been shown previously that DVFS in GPUs can be used without affecting performance [25], however further research is needed to find optimal policies. Currently, there are limited power management policies implemented in GPUs. In our initial experiments, we explore the GPUs power characteristics and motivate the need for a better DVFS policy for GPUs. To increase performance, Nvidia employs GPU Boost to dynamically adjust the core and memory frequency in the GPU [26]. GPU Boost increases operating frequency whenever an application is running and the GPU has not hit its power or thermal limits. Figure 4.2 (Baseline) shows this policy where the frequency starts at 1800 MHz and gradually steps down it's frequency to 1746MHz due to the GPUs thermal limit. Details of experimental setup is provided in chapter 4. There are occasion dips in frequency when there are times of extremely low utilization. Clearly, this policy is purely performance orientated and does not optimize for low utilization. Frequencies are throttle when power or thermal caps are hit, or when it is completely idle.

Chapter 3

Motivation

In this section, we investigate dynamic frequency scaling effects in latency-critical services on GPUs. Specifically, our initial experiments explore the following properties; power savings and Thread Block Scaling. For our experiments, we use the deep neural network service, DjiNN [13]. DjiNN includes three separate services, image classification, natural speech processing, and face recognition. We use Djinn to vary the workload as well as create latency restrictions on the GPU. More explanation can be found in Section 4.2.

3.1 Tail Latency Calculations

To start our experiments we need to determine the Quality of Service (QoS) our system is capable of handling. To get QoS, we stress our GPU at the maximum frequency it can sustain without throttling (1809 Mhz) and run multiple tests, to get the baseline service time for a workload. We use this baseline to find the theoretical number of requests that can be serviced within a second, as seen in 3.1. We then again test our system with a

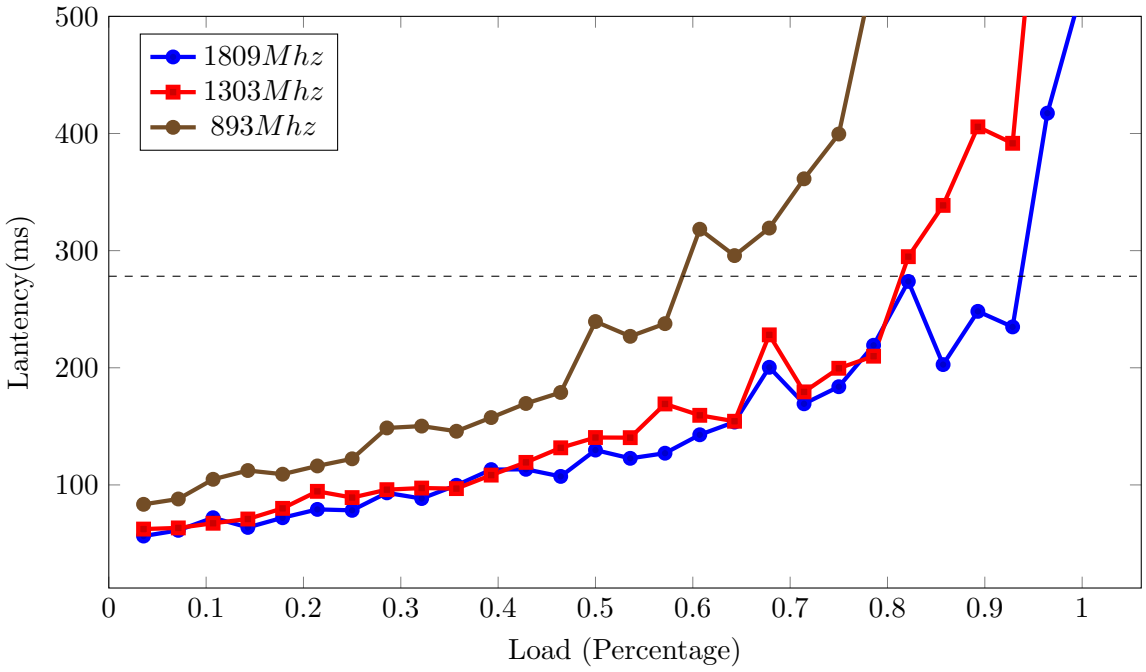

Figure 3.1: Impact of frequency and server load on tail latency. At light loads, lower frequencies are able to maintain QoS.

client sending requests at the Theoretical Full Load and record the 99% latency.

$$TheoreticalFullLoad = \frac{1}{t_{Baseline}} \tag{3.1}$$

From here we define the tail latency, similar to [4,14,16], at the point of the "knee" when running at full frequency, shown as the dotted line (278 ms) in Figure 3.1. This "knee" occurs around the 90% load of the GPU. We will use this tail latency as our Quality of Service requirement throughout our other experiments.

In Figure 3.1, we ran our experiments at three different frequency settings: 1809MHz, 1303MHz, and 893MHz. We ran our experiments at various loads (x-axis), ranging from active idle (0% load), to maximum load as determined by equation 3.1. As Figure 3.1 shows, when the server is underutilized, the GPU is able to run at much lower frequencies

without any QoS violations. For example, at a low load of about 40% load, we can achieve latency of 113ms , 119 ms, and 131.691 ms, while running at 1809MHz, 1303MHz, and 893MHz, respectively. These latency levels are well below the target tail latency of 278 ms. By slowing down requests and still meeting the latency targets, we can effectively run at a lower power. This provides an opportunity to slow down requests and move average latency closer to the tail latency, which will save overall power in the GPU.

3.2 Power savings from DVFS

Even though GPUs are considered to be more energy efficient in terms of the number of operations per watt, they have high overall power usage. Since there has been little study in DVFS in GPUs, it is important to understand how much power savings is possible. We measured the power of the GPU while varying the frequency under a constant load. Nvidia provides an API, Nvidia Management Library (NVML), that allows us to probe the power at the software level. This library's power measurement has an accuracy of $\pm 5\%$. We probe for the frequency every 10 milliseconds.

In Figure 3.2 we show the results of our preliminary study. We vary the frequency of the GPU from around 300MHz to 1800Mhz (as shown in the x-axis) at a low (blue line), medium (red line), and high (gray line) load. Here load is measured as the volume of requests received by the server at a given time.

Clearly, lowering the frequency will result in less power. However, Figure 3.2 shows that the GPU quickly hits a lower bound in power consumption. Instead of the typical linear relationship, we observe that power saving start to diminish for frequencies under 1000 MHz.

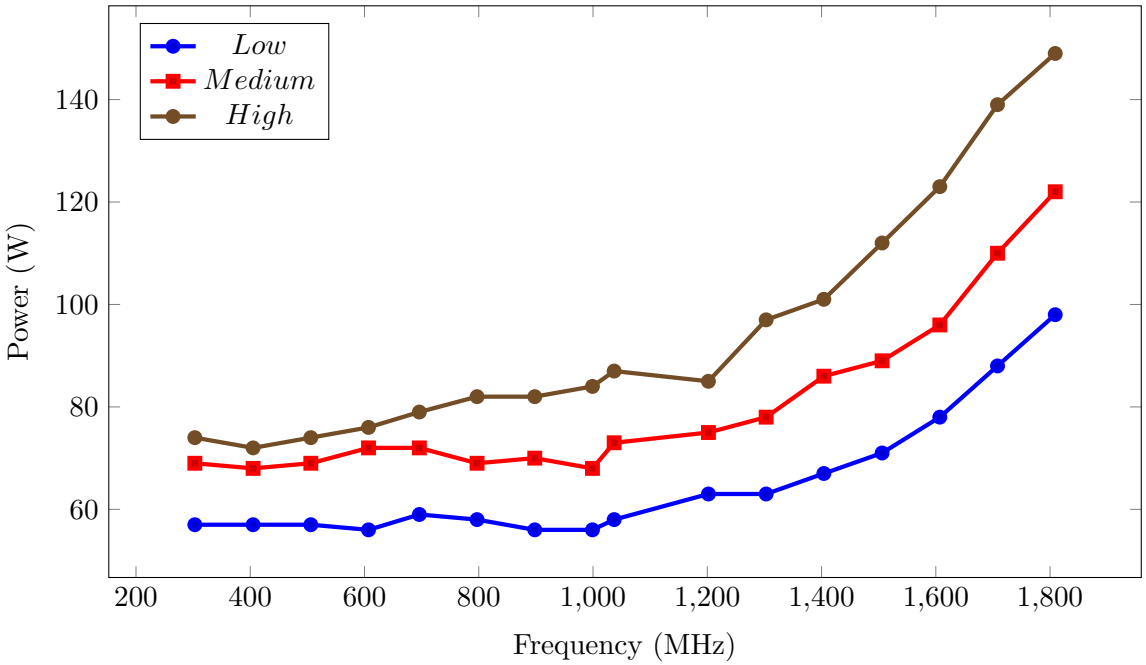

Figure 3.2: Power Usage of Low, Medium, High loads at varying frequencies. Power savings is limited at lower frequencies.

One possible explanation of this is due to on-card memory and other sources of static power consumption in the GPU card. Under any type of load, the GPU memory is used often due to there being many data transfers over the PCIe bus. Even though the GPU can be clocked at lower speeds, it does not provide any additional power savings. GPUs lack sophisticated power gating techniques and have a high static power [20]. This limits overall energy savings that we can achieve through dynamic frequency scaling and requires smarter polices that is aware of modern GPU's power consumption characteristics.

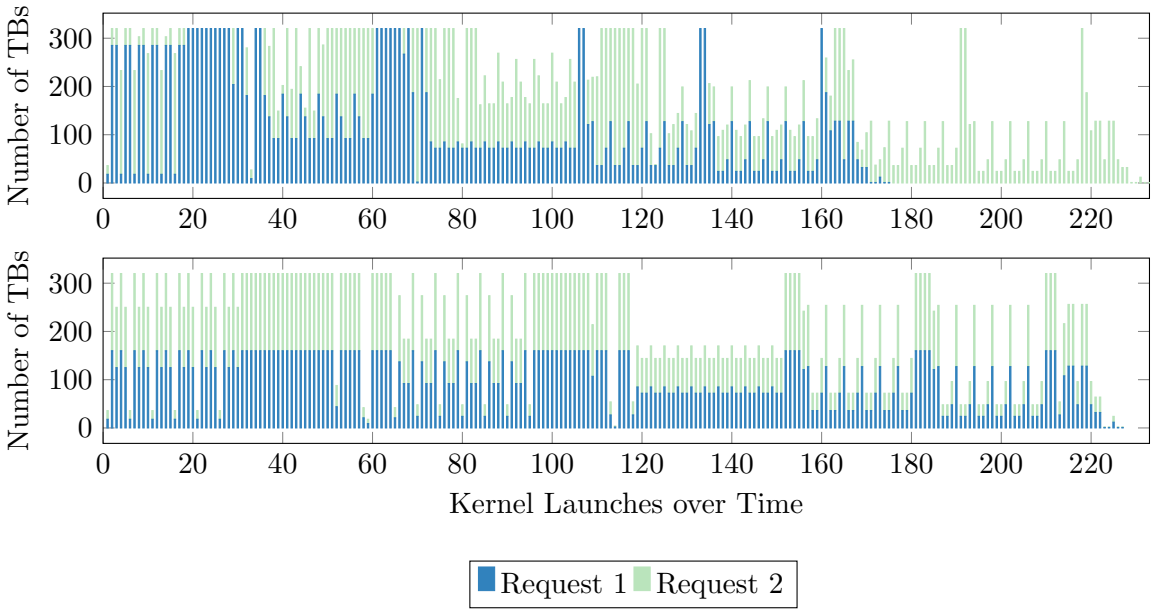

Figure 3.3: Simple example of the effect of thread block scaling with two concurrent requests of IMC workload. The top figure shows how request 1 consumes most of the resources,delaying request 2. The bottom figure shows how limiting the number of thread blocks by half can reduce the overall time of finishing both requests.

3.3 Thread Block scaling

In Figure 3.1, we showed that a latency gap exist at low loads. However, lowering the GPUs frequency may not entirely close the latency gap, and also may not necessarily save power, as observed in Figure 3.2. Therefore, we would require a different knob in order to further close this latency gap. One potential approach is through processor sleep states [4], however GPUs lack support for deep sleep states at runtime. Therefore, the needs to be a different approach in GPUs to close the remaining latency gap.

In this paper, we investigate whether scaling the concurrency of the application, by scaling the number of thread blocks a kernel has, can be utilized to cover the latency gap. By limiting the number of thread blocks within a kernel, we limit the amount of concurrency

an application can achieve on a GPU. Concurrency is also dependent on the mapping of data to threads and how much work each thread needs to complete. For example, given a matrix multiplication algorithm, a single thread may be responsible for only one element of the output matrix, or it may need to compute a block of the output matrix. In the latter, the elements in the block are complete sequentially since only a single thread is computing them. Once there are more Thread Blocks than SMs in the hardware, no more concurrency is gained.

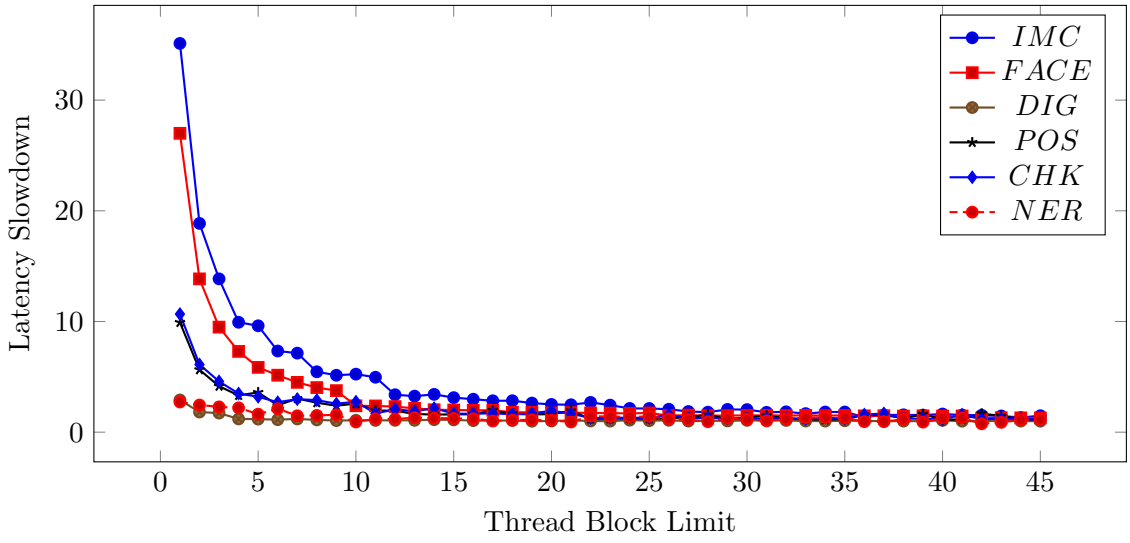

Figure 3.4: Latency Slowdown due to Thread Block Scaling.

This technique can be seen in Figure 3.3, which shows a simple example of GPU utilization with two concurrent requests. For this experiment, we ran the IMC workload and measured the number of thread blocks launched during the processing of a single request. A single requests calls multiple kernel calls (as shown by each bar in the figure). The height of the bar shows the number of thread blocks launched by that kernel. In this experiment,

we limit the number of thread blocks to 320, which is the physical thread block limit of the GPU hardware. In this figure, the x-axis the number of kernel launches over the course of a single request. The top chart shows how the GPU is utilized without any thread block scaling. In this case, Request 1 would take up all available resources, while request 2 can only use what is left over. This delays the execution of kernels in request 2, leading to a much longer runtime of request 2 compared to request 1. However, if we limit the amount of resources each request gets, then the time to complete both requests is less. This can be seen in the bottom chart. In this case, we limit each request to exactly half of the hardware thread blocks available. In this scenario, both request 1 and request 2 can process at the same time. During high thread block scenarios, the thread blocks are limited in the hardware, allowing us to to stretch the time of execution. At times of low thread block counts, both the lower thread block counts of request 1 and request 2 can fit comfortably into the hardware, and thus run concurrently without any slowdown. As we will see later, this property allows us to aggressively scale the parallelism of the GPU kernel without greatly affecting the latency of the request.

Our goal is to limit the number of TB a request is allocated in order to service multiple requests at the same time. Figure 3.4 shows the potential gains with Thread Block Scaling. At a medium load, we vary the number of thread blocks a single request uses. For the purpose of clarity, we only show the results of thread block limits from 1 to 45. We ran experiments up to the maximum hardware limit, but results from 45 to 320 is similar with minimal thread block scaling effect. The chart's y-axis is normalized to the baseline service time of each workload. We observe that at around 15 thread blocks, our service

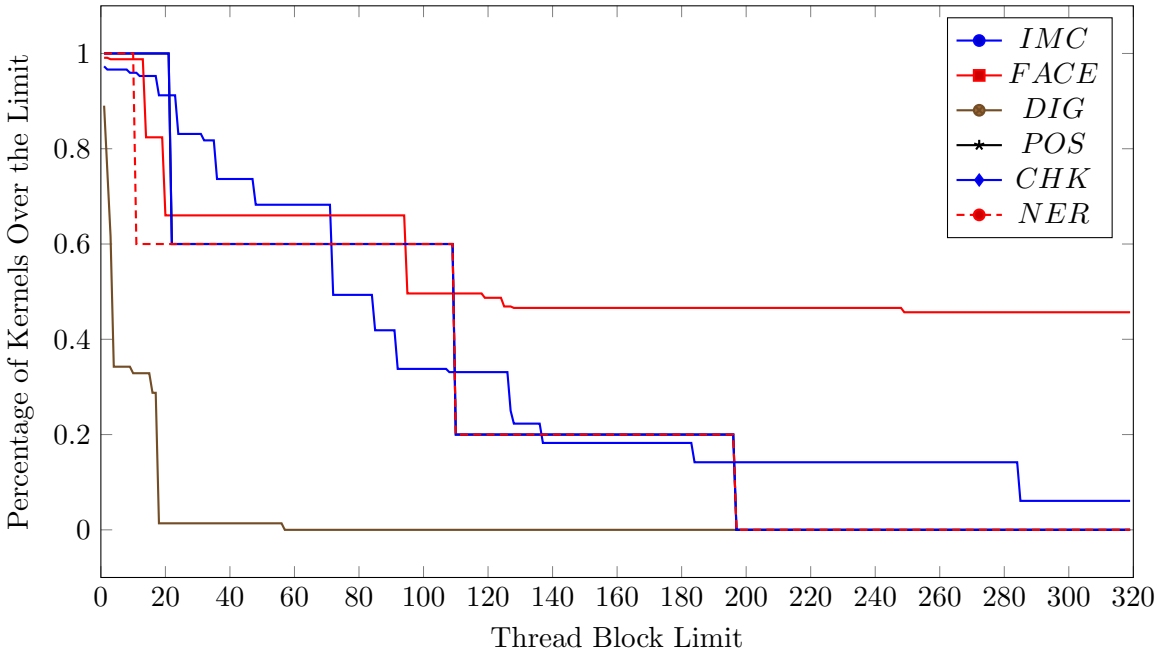

Figure 3.5: Percentage of Kernels that are greater than the limit. This chart shows that a large percentage of kernels do not utilize all GPU resources

time is still under our system's tail latency. This is true for all workloads. Below 15 thread blocks, the amount of latency slowdown increases significantly, depending on how sensitive an application is. In certain cases, even with a single thread block, some applications, such as NER and DIG, only slowdown by 3x.

With 15 thread blocks, a single GPU will be able to handle two or more requests concurrently without violating tail latency. This is possible because every request contains multiple kernel launches and every kernel launches a different number of thread blocks. Therefore, limiting the number of thread blocks actually only affects a small percentage of kernels. This can be seen in Figure 3.5. For every workload we counted the number of kernels that exceed the thread block limit as a percentage of the total kernels launched. The figure shows that most kernels are below the threshold and the limit only starts to

affect tail latency when around 75% of the kernels are restricted. At the point where the percentage of kernels over the limit hits around 75%, we observe the spike in latency shown in Figure 3.4. These results indicate that workloads then to tolerate thread block scaling well, indicating that we have many opportunities in squeezing more throughput out of the system.

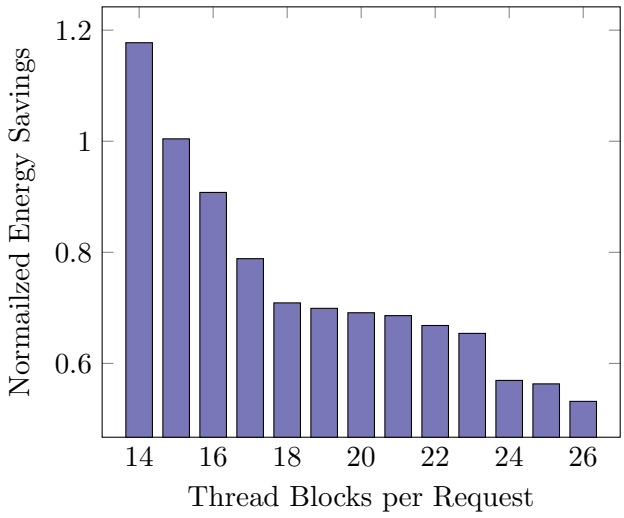

Figure 3.6: Expected Energy Consumption

By using streams we can assume two requests are processing on the GPU simultaneously and effectively double the energy efficiency per request. Figure 3.6 demonstrates the potential energy savings. This chart is normalized to the amount of energy used by a single request with no restriction on the number of thread blocks. Due to the number of thread blocks greatly exceeding the number of SMs on the hardware, the thread blocks eventually becomes serialized, incurring significant thread block scheduling overheads. Energy savings only comes if both requests take a shorter time than running sequential. This is seen when

the number of thread blocks is below 14, which actually use more energy because the service time is more than double a single request. Even though there may be less SM than current running thread blocks — e.g. two requests using 20 thread blocks each on the Titan X with 28 SM — there are many opportunities for the two requests to overlap computation. Therefore, the service time for both requests will be less than double and we expect to see the type of energy savings similar to our calculations. These initial results show that the concurrency of thread blocks processing requests can have significant impact on the energy efficiency of GPUs.

Chapter 4

Evaluation

We propose a workload-aware frequency selection policy. The purpose of this initial work is to demonstrate the potential for energy savings in real GPUs utilizing latency-aware frequency scaling. For brevity, exploration of Thread Block Scaling is reserved for future work.

4.1 Example Power Management Policy

We propose our own GPU Runtime in Djinn that takes into account both DVFS and Thread Block (TB) scaling. Figure 4.1 describes the operation of this runtime system. All the incoming requests arriving at the network socket is pushed into the Request Queue (RQ). Caffe further pops the requests out of the queue and processes it. Caffe creates the thread blocks as per the TB scheduler and launches them on the GPU. Caffe batches requests based on the limits obtained from the TB scheduler.

The DVFS scheduler obtains GPU performance metrics related to voltage, power

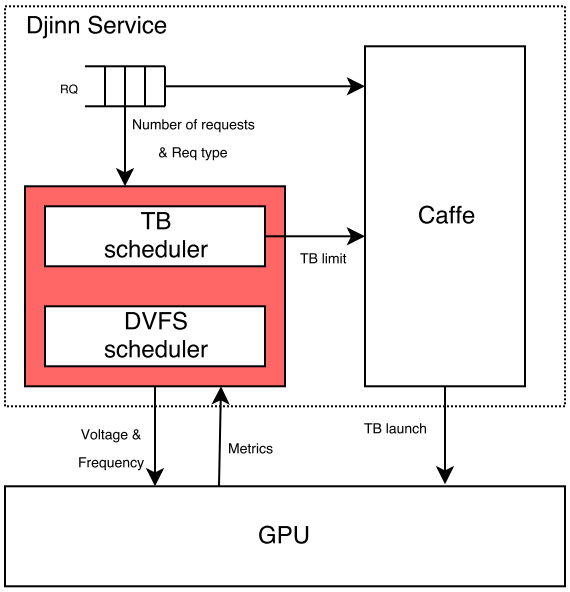

Figure 4.1: Runtime Framework in Djinn

and frequency to determine the required frequency and voltage scaling. As the utilization of the server fluctuates, it will select the minimum frequency the GPU can operate at while supporting the QoS-level, as determined in previous sections. Utilization is calculated by the number of incoming requests over a short time frame. The policy selections are based off of precomputed tail latency tables similar to Rubik [16]. Our tables incorporate queuing time plus GPU service time. GPU service time is the time for Host to Device transfers, GPU Execution time, and Device to host transfers. These tables are computed through empirical testing. We run a test across varying loads, through varying the Requests per Second at every frequency step on the GPU. Frequency steps are 100 Mhz apart starting at 1800 Mhz down to 300 Mhz. Table 4.1 shows the precomputed frequencies for a given load. In this table, we also compute the average power the GPU consumes while running at the specified frequency and load.

Table 4.1: Precomputed Frequencies to Meet QoS

Load	Frequency	Power	Load	Frequency	Power
5	303	53	55	898	70
10	303	53	60	898	75
15	303	55	65	1202	81
20	303	57	70	1202	80
25	303	61	75	1404	96
30	506	59	80	1404	96
35	607	67	85	1404	101
40	797	66	90	1809	155
45	797	67	95	1809	150
50	898	70	100	1809	155

4.2 Experimental Setup

We evaluate our policy on our GPU server with an Intel Xeon E5-1620 v4 processor, with 32GB of DDR4 RAM. We use the Pascal-based Nvidia Titan X with 28 SMs. We modified the Djinn and Tonic suite [13] to be used as a workload for our GPU. For our tests, we use Djinn's image recognition service. For these initial experiments we use a static request size. Each request is batch of 6 face images that are sent to the server and processed on the GPU using the facial recognition DNN.

We use Google's 2011 cluster data traces [30] as a simulated data center workload over a single day. The frequency of requests where then scaled to our hardware's maximum

capacity. As a baseline, we tested the GPUs default Auto boosting policy.

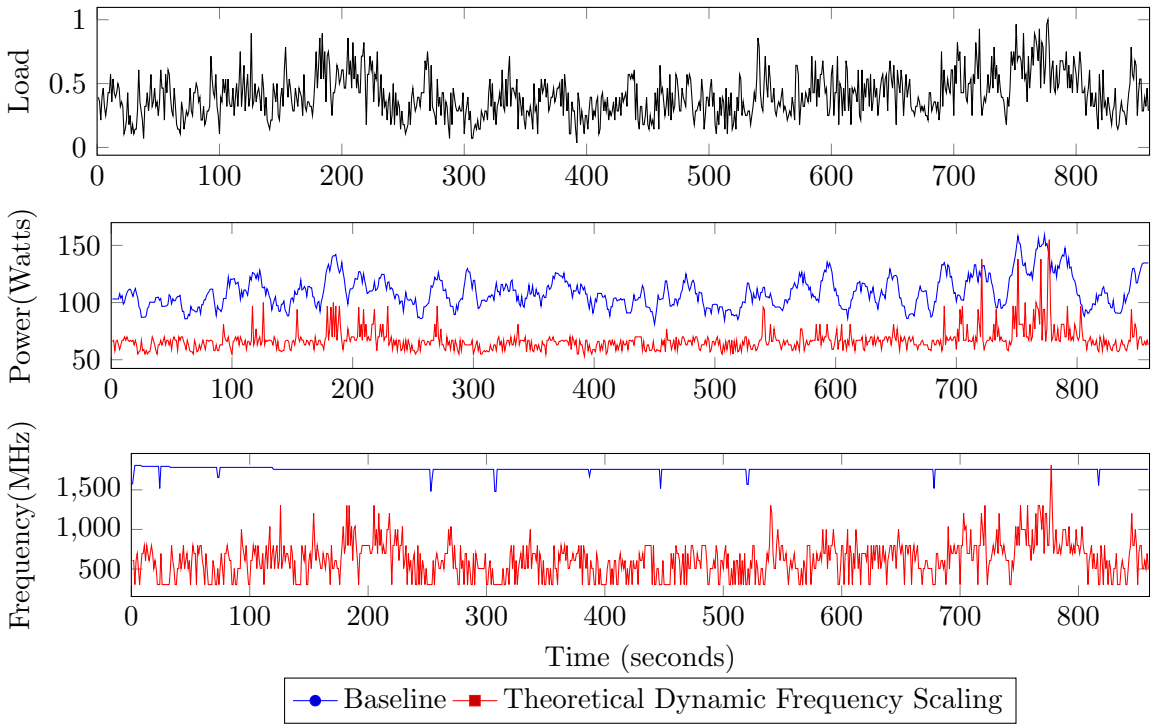

Figure 4.2: Power and Frequency Results of Titan X while running a Google data center utilization trace. Baseline (Blue) shows the behavior of existing power management in modern GPUs. The red line shows the potential for power savings by latency-aware dynamic frequency scaling.

4.3 Experimental Results

Our baseline evaluation is composed in Figure 4.2. The top chart is the load overtime. Typical to normal data center workloads, there are times of low load and high peaks. The baseline is not suited for this type of workload, as the frequency is near constant throughout the entire test, even during lower load periods. At the beginning it boosts to the max frequency, however, it cannot sustain this for long, due to thermal overheads,

Table 4.2: Average Power Usage and Frequency

Average	Baseline	DFS
Power	108.98	66.85
Frequency	1757	616

and steps the frequency down slightly. This is seen in the first 120 seconds. In red, is our theoretical dynamic frequency scaling policy. This was built by using the trace of server load and for every second took the power and frequency that matched the current load in our precomputed tail latency table. This process is similar to our proposed policy, which calculates the current load in real time, based off of the number of requests. Table 4.2, shows the average power and frequency between the two policies. Dynamic Scaling Frequency shows a significant improvement over the baseline. On average it offers a power savings of 1.6x over the default. The power savings also come with no loss of performance with regards to tail latency, because our frequency table was built with the top concern that at all levels, service time must always be under the tail latency.

As demonstrated in Section 3, significant energy savings still exist as latency slack cannot be closed entirely by frequency scaling. Thus, these results show conservative energy savings. Through Thread Block Scaling, we expect the energy efficiency of GPUs to be improved significantly.

Chapter 5

Conclusion

As faster Neural Networks and Deep Learning will continue to become a more important working in Data Centers, GPUs will be the standard accelerator used because they have been tightly focused on their performance benefits. This creates a problem as Data Centers needs to be more energy efficient, because GPUs are high power processors with limited power management. As demonstrated in our experimental results, existing GPU power management policies are aimed at exploiting thermal headroom to maximizing performance. Modern GPU power management are not designed to save power during low utilization periods, and are not aware of application's latency requirements.

However, in the case with Data Center, the current load is constantly fluctuating, which makes high performance not always a necessity. In this work, we examine possible power management techniques for GPUs. By exploiting discrepancies between the average and tail service times, we show that there is room for the GPU to save power by scaling frequency based on the current load. We also perform preliminary experiments with

Thread Block Scaling and demonstrated it as a potential solution to increase throughput and increase energy efficiency per request.

As our initial results suggest, there are still significant problems to address and solutions to explore to maximize energy efficiency in modern GPUs.

Bibliography

[1] J. M. Cebrín, G. D. Guerrero, and J. M. Garcia, "Energy efficiency analysis of gpus," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, ser. IPDPSW '12, 2012.

[2] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[3] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGARCH Computer Architecture News*, 2016.

[4] C.-H. Chou, D. Wong, and L. N. Bhuyan, "Dynsleep: Fine-grained power management for a latency-critical data center application," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016.

[5] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013. [Online]. Available: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[6] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "Coscale: Coordinating cpu and memory system dvfs in server systems," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45, 2012.

[7] R. Ge, X. Feng, W. c. Feng, and K. W. Cameron, "Cpu miser: A performance-directed, run-time system for power-aware clusters," in *2007 International Conference on Parallel Processing (ICPP 2007)*, Sept 2007, pp. 18–18.

[8] R. Ge, R. Vogt, J. Majumder, A. Alam, M. Burtscher, and Z. Zong, "Effects of dynamic voltage and frequency scaling on a k20 gpu," in *2013 42nd International Conference on Parallel Processing*, Oct 2013, pp. 826–833.

[9] R. Gonzalez, B. M. Gordon, and M. A. Horowitz, "Supply and threshold voltage scaling for low power cmos," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 8, pp. 1210–1216, Aug 1997.

[10] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM computer communication review*, 2008.

[11] M. Harris, "Gpu pro tip: Cuda 7 streams simplify concurrency@ONLINE," January 2015. [Online]. Available: https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

[12] J. Hauswald, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," Djinn Talk, 2015.

[13] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015.

[14] C. H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[16] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[17] T. Komoda, S. Hayashi, T. Nakada, S. Miwa, and H. Nakamura, "Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, Oct 2013, pp. 349–356.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1097–1105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257

[19] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[20] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: enabling energy optimizations in gpgpus," in *ACM SIGARCH Computer Architecture News*, 2013.

[21] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14, 2014.

[22] Y. Liu, S. C. Draper, and N. S. Kim, "Sleepscale: Runtime joint speed scaling and sleep states management for power efficient data centers," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, 2014.

[23] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, 2014.

[24] 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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352864816300736

[25] X. Mei, L. S. Yung, K. Zhao, and X. Chu, "A measurement study of gpu dvfs on energy conservation," in *Proceedings of the Workshop on Power-Aware Computing and Systems*, ser. HotPower '13, 2013.

[26] NVIDIA, "Nvidia gpu boost for tesla," NVIDIA, Tech. Rep., January 2014.

[27] ——, "Multi-process service," NVIDIA, Tech. Rep., October 2017. [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[28] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz *et al.*, "The kaldi speech recognition toolkit," in *IEEE 2011 workshop on automatic speech recognition and understanding*, no. EPFL-CONF-192584. IEEE Signal Processing Society, 2011.

[29] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, June 2014, pp. 1701–1708.

[30] J. Wilkes and C. Reiss, "Google cluster data 2011_2," 2016.

[31] H. Yang, Q. Chen, M. Riaz, Z. Luan, L. Tang, and J. Mars, "Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained cmp," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.