UNIVERSITEIT VAN AMSTERDAM

Auto-Tuning for Energy Efficiency on GPUs

A Thesis Submitted to the Informatics Institute of the

University of Amsterdam

In Partial Fulfilment of the Requirements for the

Degree of

Master of Science

IN

Computer Science-Parallel Computing Systems

By:
Ehsan Sharifi Esfahani

Supervisors:
Prof. Dr. Rob van Nieuwpoort
Dr. Alessio Sclocco
Dr. Ana Varbanescu

August 2019

Acknowledgments

I would like to acknowledge the following people for their kindness and support during the course of my thesis and my entire master's program:

First off, I would like to take this opportunity to express my deepest gratitude to my advisors, Prof. Dr. Rob Van Nieuwpoort and Dr. Alessio Sclocco, not only for their continuous guidance, but also their valuable comments and help that I enabled to fulfil this thesis.

I would also like to express my thanks to Dr. Ana Varbanescu for her advice and all my good classmates for many things I have learned from them.

Finally, I have to thank Dr. Clemens Grelck for his good understanding of me, so I never give up.

Abstract

Because GPUs are a popular platform for accelerating applications, increasing the energy-efficiency of GPU computing is a relevant research topic for many applications and infrastructure owners.

One way to address this problem is through auto-tuning. However, existing GPU auto-tuning strategies do not consider power capping. In this research, we demonstrate how auto-tuning can be enhanced to support energy-efficiency improvement for GPUs. Specifically, we define three different auto-tuning strategies: (1) tuning core frequency, memory frequency, and power capping values, (2) tuning core and memory frequencies, and (3) tuning only power capping. We show that, by tuning core and memory frequencies and power capping values together with the application tunable parameters, we can improve the energy efficiency of GPUs for multiple benchmark applications. We also show that the power capping level is a new tunable parameter in the context of GPUs and its optimal value depends on different parameters, such as GPU architecture, application, and objective function.

As an example, our empirical analysis shows important energy-consumption reduction for NVIDIA GTX980 for three applications - vector add, stencil, and matrix multiplication - by 21%, 23%, and 17%, respectively. These figures correspond to increases in energy efficiency of 23%, 11% and 9%, respectively. We therefore conclude that auto-tuning can be successfully adapted for GPU green computing if it includes core/memory frequency and power capping as tunable parameters.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Graphic Processing Units (GPUs) are used to accelerate massively parallel computations in fields as diverse as data mining, high-quality video processing, weather modeling, machine learning, virtual reality, galaxy simulation and so on. Because of their wide adoption, GPUs have become an integral part of High-Performance Computing (HPC) systems worldwide.

Energy consumption is a key aspect of these systems, because of temperature-related problems and high running costs [6]. For example, more than half of total energy consumption of every compute node in Summit, the system leading the TOP500 list of supercomputers on November 2018, is consumed by its 6 NVIDIA Tesla V100 GPUs [1] [2]. Hence, the research community focus to reduce the power consumption of the GPUs to build more energy-efficient HPC systems.

GPU programs, have several parameters, e.g. block size and grid size, that can be fine tuned to improve performance or energy consumption. Auto-tuning is a technique to automatically find the best set of values to optimize such objective functions as performance or energy consumption [3]. Finding these optimal parameter values is a challenging and tedious task for programmers since there is no a priori knowledge on which parameters are good and which are not.

Moreover, GPUs provide the capability of altering core or memory frequencies, and these can be taken as two other tunable parameters to improve performance or energy consumption. However, this can significantly increase the search space of all the set of possible parameter values. Previous research investigated the tuning of core and memory frequency, but did not take into account the effects of power capping. GPU-level power capping is the ability to set a threshold on how much power a GPU can consume, at most, and can be thought as a new tunable parameter.

Although adding another parameter increases the search space, it does also provide a new opportunity to reduce energy consumption, especially for applications where power dissipation is a critical issue. As an example, the estimated power consumed by the computing component of the Square Kilometre Array (SKA), a large radio telescope, is in the order of 50MW [4]. This high power consumption, which increases tremendously the operational costs of such a project, is indeed one of the main challenges of this project [5].

1.1. Research Questions

In this thesis, we are going to consider the impact that GPU-level power capping has on tuning, therefore, our main research question is:

RQ: Can power capping be considered a new tunable parameter for GPUs?

If the answer to this question is positive, then more sub-questions can be asked, such as:

- What is the impact of tuning power capping on energy consumption, energy efficiency, performance, memory frequency, and core frequency?

- When adding power capping to the tunable set of GPU parameters can provide more benefits than just tuning core and memory frequency?

1.2. Contribution and Outline

The major contributions of this thesis is that, to the best of our knowledge, we are the first to consider the tuning of GPU-level power capping in addition to other tunable parameters. The remaining of the thesis is organized as follow: Chapter 2 provides the background knowledge necessary for a clear understanding of this thesis. It starts by presenting different GPU architectures and their key features, the programming model of GPUs, Dynamic Voltage Frequency Scaling (DVFS), and auto-tuning. Chapter 3 presents our methodology and experiments. It describes the experimental setup, impact of tuning power capping on energy consumption, energy efficiency, performance, core frequency, and memory frequency for different GPU platforms. At the end of this chapter, we discuss the related works. Finally, chapter 4 summarizes our conclusions and highlights some future work.

Chapter 2

Background

In this chapter, we provide an overview of the preliminary concepts and background to understand the rest of the thesis. The first section begins with an explanation of several key aspects of different Nvidia GPU microarchitectures, followed by the CUDA programming model. The remaining sections discuss the DVFS technology and auto-tuning, respectively.

2.1. GPU Microarchitectures

It is nowadays common to use Graphics Processing Units (GPUs) to solve complex scientific problems, since they can provide high-performance while being more energy-efficient than traditional CPUs. Unlike CPU, a GPU consists of numerous small and simple cores working together according to the Single Instruction Multiple Data (SIMD) paradigm. In SIMD, the same instructions are executed on different data simultaneously and thus providing data-level parallelisms.

GPU Family	Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing
Release year	2006	2009	2012	2014	2016	2017	2018
CA	1.x	2.x	3.x	5.x	6.x	7.x	7.5
Feature size (nm)	>55	40	28	28	16	12	12
Approximate maximum number of transistors ($x10^9$)	1.4	3.2	7.08	8.0	15.3	21.1	18.6
Total CUDA cores min/ max	128/960	448/1792	2496/4992	772/1033	2560/3584	5120	2560
Base clock (MHz) min/max	1296/1440	1150/1300	560/745	722/1033	810/1328	-	585
Maximum supported clock min/max	Not Supported	Not Supported	758/875	1051/1178	1063/1531	1370/1455	1590
Memory Interface	DDR3	DDR5	DDR5	DDR5	DDR5/ HMB2	HBM2	GDDR6
Device Memory (GB) min/max	1.5/16	3/24	2/24	4/32	8/24	16 or 32	16
Bandwidth (GB/s) min/max	76/408	144/592	160/480	88/320	192/732	900	320
TDP (watts) min/max	170/1200	225/900	225/300	50/300	50/250	250/300	70

Table 2.1: Nvidia GPU architectures since 2006, '-' indicates unknown.

Although there are multiple GPU vendors on the market (e.g. Nvidia, AMD, Intel, ARM), in this thesis we are only going to focus on Nvidia as it is the leading provider for GPUs in supercomputers and HPC environments. In 2006, Nvidia introduced the Tesla GPU which architecture is the underlying model of current GPUs [13]. Three years later, a more advanced microarchitecture was announced by Nvidia: Fermi. The main difference between Tesla and Fermi was the addition of a cache system [9]. The data in the Fermi family is cached in the L1 and L2 cache hierarchy while there was no cache system in Tesla GPUs. Table 2.1 summarizes different GPU generations with their several characteristics since 2006.

Two terms in this table may need further explanation: Compute cApability (CA) and Thermal Design Power (TDP). Nvidia defines CA as [10]:

> "*The compute capability of a device is represented by a version number, also sometimes called its "SM version". This version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.*"

TDP, falsely believed as being the maximum possible power consumption, is defined by Jeffers et al [18] as:

> "*The TDP rating is refers to the maximum amount of heat generated by the device which the cooling system is required to dissipate in typical operations.*"

Therefore, a proper cooling systems should be selected based on TDP so that it can deal with the amount of heat dissipated to keep the temperature of the processor below the allowable maximum temperature. The authors in [18] and [19] have stated that the TDP can be roughly 50% [18] or 80% [19] of the maximum possible power consumption. The main point here is that the TDP is different than the maximum possible power consumption and the power consumption of a processor can exceed the TDP. For example, the TDP and the maximum possible power consumption for Nvidia TitanX (Pascal) GPU used in DAS-5 are 250 and 300 watts, respectively.

2.1.1. Fermi Microarchitecture

The first released Fermi GPU contained 512 CUDA cores and 16 Streaming Multiprocessors (SMs) along with different components such as L2 cache system and a scheduler. A CUDA core, or simply core, is the most basic execution unit and can run a floating-point per clock. A group of CUDA cores working together organized in a SM. Figure 2.1 shows a high-level diagram of the Fermi microarchitecture.

Fermi support a two-level distributed thread scheduler. There is a scheduler unit in Fermi named GigaThread responsible to schedule a thread block (see Section 2) to one SM, and then a thread scheduler responsible to select a warp that is ready for execution. The thread scheduler is significantly improved in the Fermi microarchitecture through 10x faster context switching and by supporting concurrent running of a different kernel in the same program, thereby increasing GPU utilization and performance.

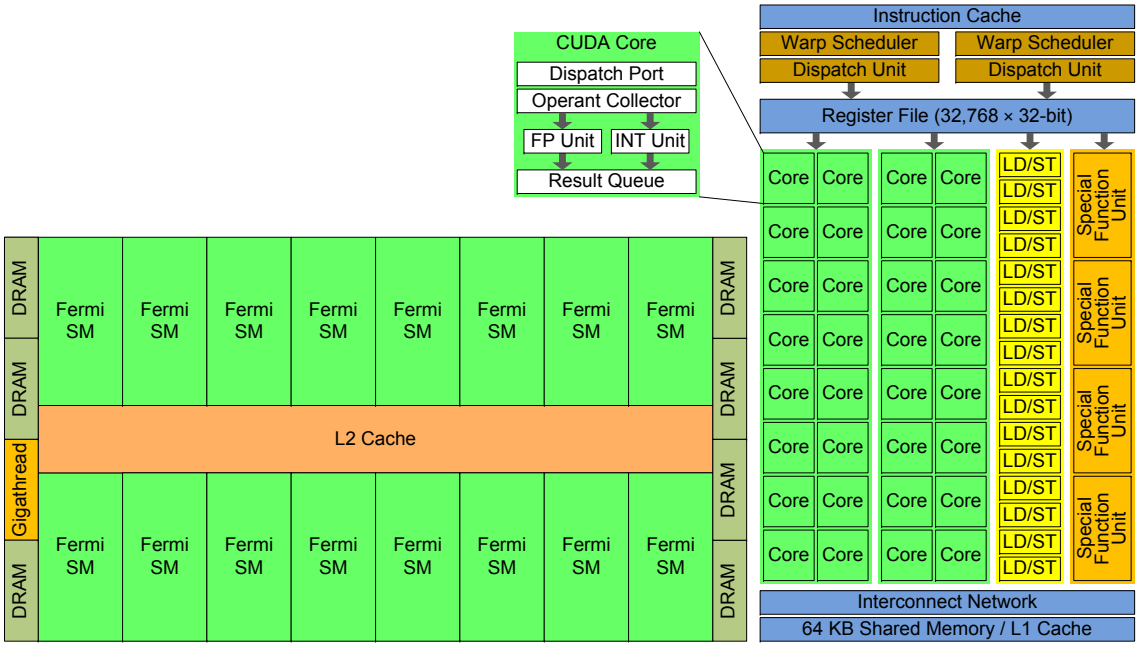

Figure 2.3: A simplified Nvidia Fermi microarchitecture (on the left) and Fermi SM (on the right), taken from [12]

Nvidia improved several key aspects for the Fermi architecture based on user feedback from their previous architecture, Tesla. They remarkably increased the single/double-precision operations performance compared to the Tesla generation. It was due to the reason that there was only one single/double-precision unit per SM in Tesla family [15] [16]; however, Nvidia placed a single/double-precision unit for each core in Fermi CUDA cores.

Furthermore, Error Correcting Code (ECC) ability for the correction of single bit error and detection of double bit errors were introduced in Fermi. This capability is a substantial feature in data-sensitive applications to improve data integrity and reliability in HPC platforms. In addition, in an environment with a large number of GPUs, e.g. data centers, this can be a critical issue since the probability of errors have a linear relationship with the number of installed GPUs.

Increasing the performance of the atomic memory operations was another improved feature in the Fermi microarchitecture. Atomic operations are common in most parallel applications with shared resources. The achievable performance of Fermi memory atomic operation was up to 20x more than the previous Tesla generation. Moreover, in multitasking environments, GPU resources can be used in an interleaved manner among the different kernels. Therefore, a fast context switching can play an important role to enhance performance. This issue was vastly improved in Fermi with a 10x faster context switching.

2.1.1.1. Fermi Streaming Multiprocessor (SM)

As it is shown in Figure 3, every SM contains 32 CUDA cores, 16 load/store units to handle memory operations, four special function units and 64KB configurable memory used as L1 cache and shared memory in a block of threads. Additionally, this figure illustrates that every SM comprises of four execution blocks namely, two 16-core blocks, one 4-SFU block

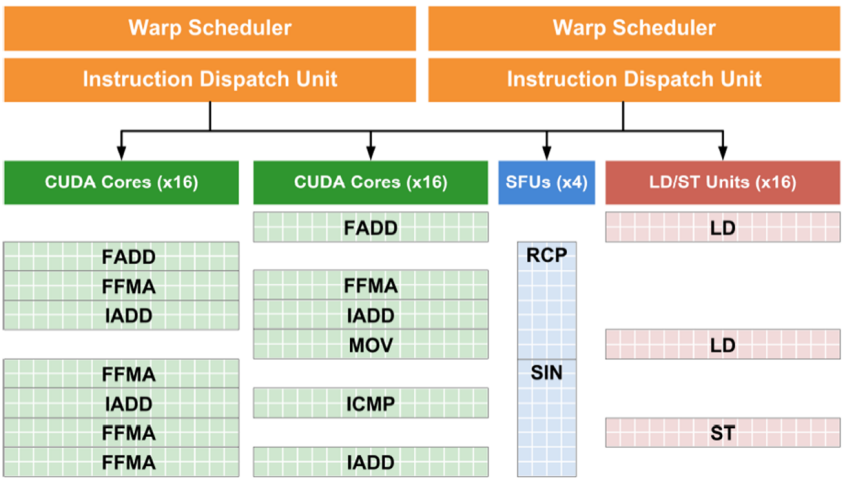

Figure 4.2: Fermi execution model in each SM, taken from [15]

and one 16-Load/Store unit block. The special function units are used for transcendental and reciprocal operations like square root and sine.

In addition, each SM contains two warp schedulers and two instruction dispatch units to schedules and execute two warps concurrently. That is, these two warp schedulers select two ready warps and then issue two instructions for any combination of the two execution blocks per-clock. A diagram of the Fermi execution model in a SM is demonstrated in Figure 2.2.

Each CUDA core contains one Arithmetic Logic Unit (ALU) and a Floating-Point Unit (FPU). Single and double precision Fused Multiply Add (FMA) are supported by FPU. A single-precision FMA instruction, also known as Multiply Accumulate Operation (MAC), is done in one clock cycle as (2.1):

$$a \leftarrow a + (b \times c) \tag{2.1}$$

2.1.2. Kepler Microarchitecture

The Kepler GPU microarchitecture was designed and implemented based on the Fermi platform with more improvement on performance, power-efficiency, and programmability. This improvement has been achieved through revising the architecture of computation units and adding some new capabilities. For example, the new architecture approach in Kepler for SM, so-called the next-generation Streaming Multiprocessor (SMX), features 192 single-precision CUDA cores, 64 double-precision units, 32 SFU and 32 Load/Store units (see Figure 2.3). Every SMX contains four warp schedulers and eight dispatcher units, thereby enabling to execute four warps and dispatch two independent operations for each warp concurrently. Figure 2.4 illustrates this concept by a schematic view of warp scheduler which dispatching two independent instructions and runs them concurrently. In Kepler, Nvidia improved power-efficiency significantly by reducing the GPU clock and replacing the complex warp scheduler units in Fermi with a simpler and more power efficient one.

Shuffle instructions with more CUDA cores, bigger per-thread registers, and faster atomic operations were four remarkable techniques to increase performance in Kepler. Shuffle

Figure 2.3: SMS Kepler architecture, taken from [17]

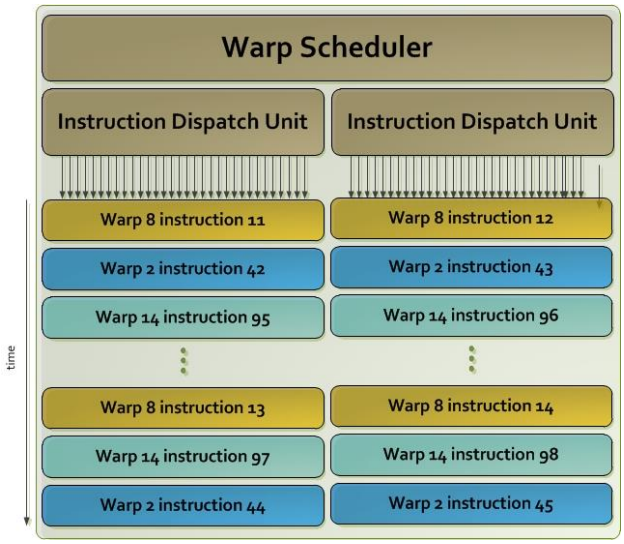

Figure 2.4: Dispatching and running tow independent instructions by each warp scheduler in Kepler, taken from [17]

instructions enables threads within the same warp share data, instead of using expensive load-store shared memory operations.

Being able to read and write data in shared memory without conflicts is a critical issue in parallel computing. Atomic operations are significantly helpful to let threads read/write data without any interference to other threads. Nvidia does not only provide us faster atomic

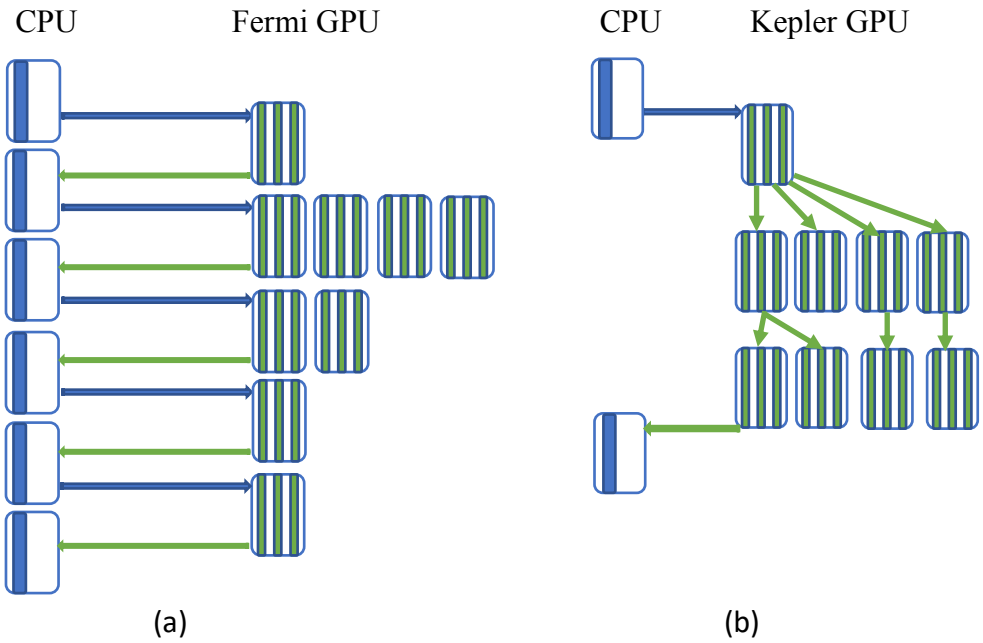

Figure 2.4: The ability of creating more works by GPU (dynamic parallelism) (b) vs non-dynamic parallelism execution (a), taken from [17]

operations in Kepler than Fermi, but also the program benefits from more different atomic instructions.

Dynamic parallelism is another significant characteristic of the Kepler family by which a kernel can execute another kernel without any need for CPU interaction. That is, with Kepler the GPU is able to generate more parallelism by itself. This feature can bring improvements to recursive and data-dependent kernels and enable a larger part of the program to execute directly on GPUs. Moreover, CPU can make use of dynamic parallelism capability since it can be available to do other additional works.

Hyper-Q is another technique implemented in Kepler to improve concurrency related to the executing of different streams. In Fermi, there is only one hardware queue from which the scheduler selects the next job to execute based on a First In, First Out (FIFO) queue. In Kepler, Nvidia uses separate work queue for each stream which leads to achieving more performance in multi-stream programs.

Fermi and Kepler memory hierarchies share some similarities: a configurable L1 cache and shared memory in each SM and an L2 cache shared between all the SMs. As it is depicted in Figure 2.6, “Read-only Data” memory unit is the only significant difference between the memory system in Kepler and Fermi. Nvidia puts this memory component to enable directly loading the operations per SM with a dedicated data path.

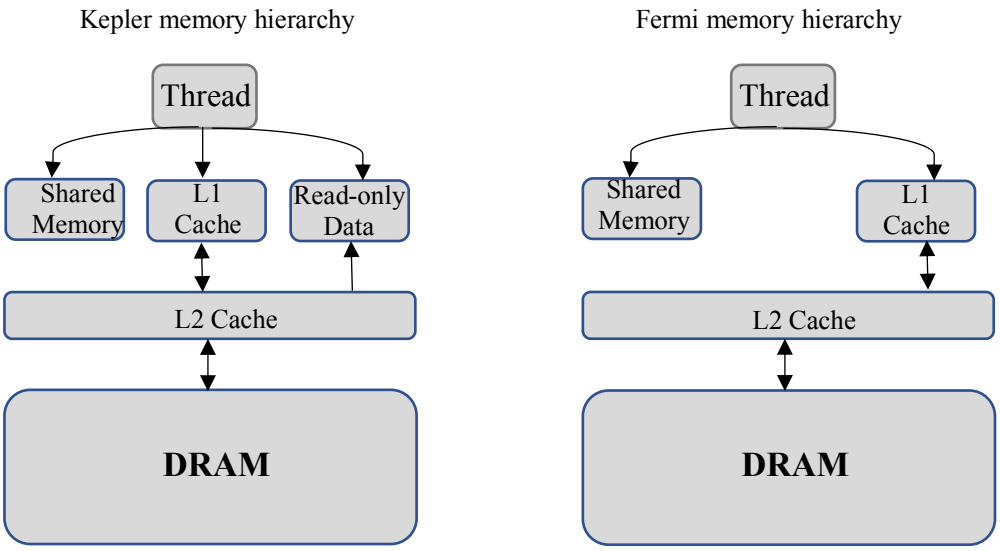

Figure 2.6: Fermi memory hierarchy vs Kepler memory hierarchy, taken form [17]

2.1.3. Maxwell Microarchitecture

Maxwell is the architecture that follows Kepler. The Maxwell generation introduced a new SM design named Maxwell Streaming Multiprocessor (SMM) with 100% and 35% enhancement in power-efficiency and performance, respectively [20]. This improvement makes Maxwell more suitable for embedded devices with a limited source of energy. This progress was achieved through several improvements in the architecture design such as SMs and memory system.

In SMM, every processing block (i.e. 32 CUDA core, 8 Load/Store units and 8 Special Function Units) contains its own warp scheduler, dispatch unit and instruction buffers. This new architecture design simplifies the warp scheduling since every warp scheduler dispatches one warp to its dedicated processing block, thereby significantly increasing the CUDA core utilization. Figure 2.7 shows the diagram of the SMM.

Reduced arithmetic instruction latency is another striking improvement in SMM. This leads to improvement throughput and utilization. Moreover, besides to larger L2 cache, SMM enables a 64KB dedicated shared memory for each SM, instead of it being shared between shared memory and L1 cache like in the previous architectures.

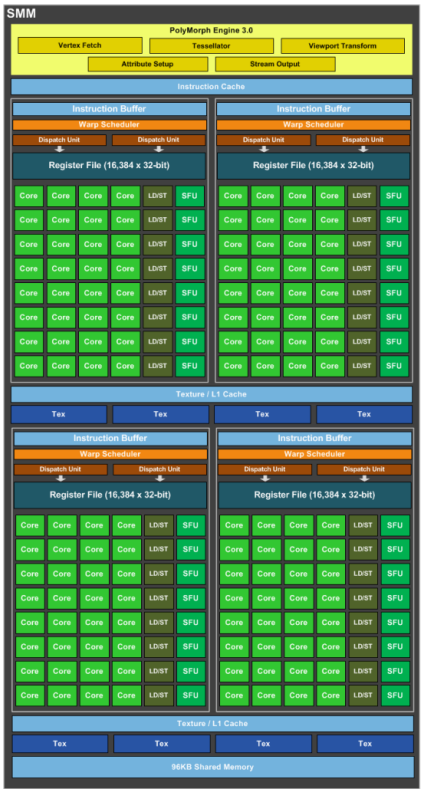

Figure 2.7: Maxwell Streaming Multiprocessor, taken from [20]

2.1.4. Pascal Microarchitecture

After Maxwell, it is the turn of Pascal. Because there are many major improvements in this architecture, compared with Maxwell, in this section, we are going to dedicate a subsection to several of them.

2.1.4.1. Extreme performance

Pascal can provide a theoretical peak performance of 5.3, 10.6, and 21.2 TFLOPS for double precision floating point (FP64), single precision floating point (FP32) and half precision floating point (FP16), respectively. The corresponding figure for single precision floating point is about 3x more than that of Kepler and 1.5x more than that of Maxwell GPUs [21].

The new half precision floating point operations can be beneficial to many deep learning applications. This leads to increasing the speed up and reducing the amount of data transferred from/to memory.

2.1.4.2. NVLink

Multi-GPU platforms are becoming more prevalent in the HPC clusters. One of the main bottlenecks to increasing achievable performance in a machine with multiple GPUs is the limited bandwidth in the communication links between the CPU and the GPUs. Pascal

introduces a new high-speed interface, NVLink, which supports 160 GByte/Second data transfer speed by bidirectional connectivity between these devices.

2.1.4.3. HBM2

High Bandwidth memory 2 is a new high-performance RAM interface, implemented in Pascal, with an improvement in memory bandwidth, energy-efficiency and die space. It provides three time more memory bandwidth than that of the Maxwell architecture, and thus enabling to tackle memory-bounded applications with better performance. Moreover, HBM2 features to connect multiple vertical memory dies to each other through microscopic wires. This capability is beneficial to build denser GPU servers or in embedded systems with more space saving.

2.1.4.4. Unified memory

Historically, CUDA forces developers to explicitly manage memory allocation and data transfers between CPU and GPU and this is both difficult and prone to errors. In Pascal, CPU and GPU memory address is united in a single 49-bit virtual address space, as it is shown in Figure 2.8. The 49-bit address space can cover the virtual address space in the most state-of-the-art CPUs and GPUs together, thereby enabling the applications to process big datasets.

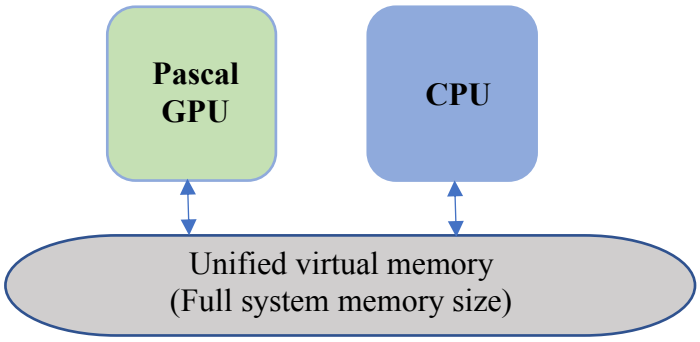

Figure 2.8: Pascal large unified memory, taken from [21]

CPU code

```
1.  void sortFile(FILE *fp, int N) {
2.    char *data;
3.    data = (char *)malloc(N);
4.
5.    fread (data, 1, N, fp);
6.    qsort(data, N, 1, compare);
7.
8.
9.    use_data(data);
10.   free(data)
11. }
```

GPU code (with operating system support)

```
1.  void sortFile(FILE *fp, int N) {
2.    char *data;
3.    data = (char *)malloc(N);
4.
5.    fread (data, 1, N, fp);
6.    qsort<<<...>>>(data, N, 1, compare);
7.    cudaDeviceSynchronize();
8.
9.    use_data(data);
10.   free(data)
11. }
```

Listing 2.1: An example of using unified memory in Pascal with global system allocator, taken from [21]

Supported unified virtual address space makes possible to access the data with only one pointer in GPU or CPU. Supporting page faulting in Pascal leads to easily using unified memory since programmers do not require to deal with synchronization of the shared data between GPU and CPU. It is because when GPU request a data located in CPU memory, it faults, and then the data is transferred automatically from CPU to GPU memory. Page fault capability enables global data coherency in this unified virtual memory system. It should be noted that it is required to modify the operating system settings to be able to work with global allocator in malloc function. As an example, Listing 2.1 shows how unified memory simplifies programming and transferring the data to GPUs.

2.1.5. Volta Microarchitecture

In this section, we are going to explain several key interesting characteristics of the Volta.

2.1.5.1. Maximum Performance and Maximum Efficiency Mode

To increase the flexibility of the data center architectures, two different running modes have been introduced in Volta GPUs, named maximum performance mode and maximum efficiency mode. The former aims to run the program up to its TDP to increase performance and the latter aims to saving power. Maximum efficiency mode can be implemented using the power cap capability in NVML library or NVIDIA-SMI tool. Interestingly, the same effects can be obtained in previous architectures by putting a limit on power draw. Nvidia simply made these modes more official in the Volta documentations.

2.1.5.2. Volta Streaming Multiprocessor

Nvidia designed and implemented a new architecture for Volta SMs to enhance energy-efficiency, performance and programmability. For example, tensor cores features in Volta SMs to improve the training of neural networks. Tensor cores can improve the performance of deep learning training up to 12x in comparison with standard 32-bit Floating Point (FP32) operations in Pascal GPUs. Each Tensor Core is specially designed to perform the operation in (2), as shown in Figure 2.9.

$$D_{4\times4} = (A_{4\times4} \times B_{4\times4}) + C_{4\times4} \tag{2.2}$$

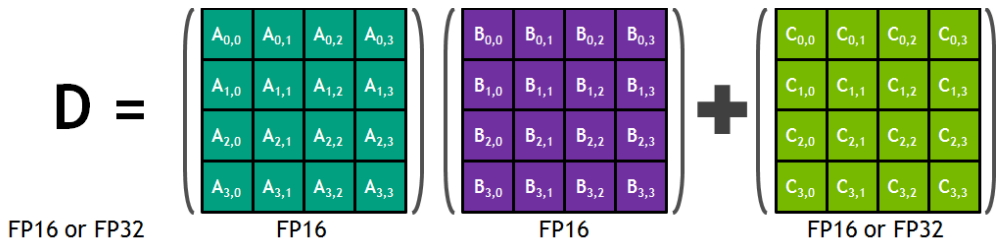

Figure 2.9: Tesnor Core operations on in Volta SMs taken from [22]

Here, the data in the matrices A and B can are FP16 (half-precision floating point) and the in the D and C can be either FP16 or FP32.

2.1.5.3. Volta Multi-Process Service

Traditionally, time-sharing is the typical method to share GPU resources among different applications which is a very expensive method because of heavy context switching. One of the key features of Volta GPUs is Multi-Process Service (MPS) to implement sharing GPU resources between several processes. MPS allows different applications from the same user to run simultaneously on a single GPU, thereby increasing the GPU utilization.

2.1.5.4. Independent Thread Scheduling

Another key feature in Volta GPUs is independent thread scheduling to enable fine-grain synchronization among parallel threads in a warp. As it is illustrated in Figure 2.10, every warp, in the pre-Volta GPUs, runs in a Single Instruction Multiple Thread (SIMT) fashion i.e. many threads execute the same instructions on different data elements with one shared program counter and one shared stack. In this paradigm, code with branch divergence is serialized, reducing performance. Here, no synchronization is allowed among the threads.

However, in Volta, independent thread scheduling enables thread synchronization in a warp using __syncwarp() method in CUDA. This happens because every thread can schedule and run independently using its own program counter and stack. It should be noted that the execution model stays SIMT in Volta since the same operations are running by CUDA cores at each cycle.

2.1.6. Turing Microarchitecture

Turing is the newest Nvidia architecture and it contains several advanced and improved features dedicated to accelerating 3D graphic and Artificial Intelligence (AI) applications [23]. The new Turing SM contains two major architectural changes. First, the Turing SM can run floating-point and integer operation concurrently. In the pre-Turing generations, the floating-point path data would be idle when an integer operation is running. Second, the shared memory,

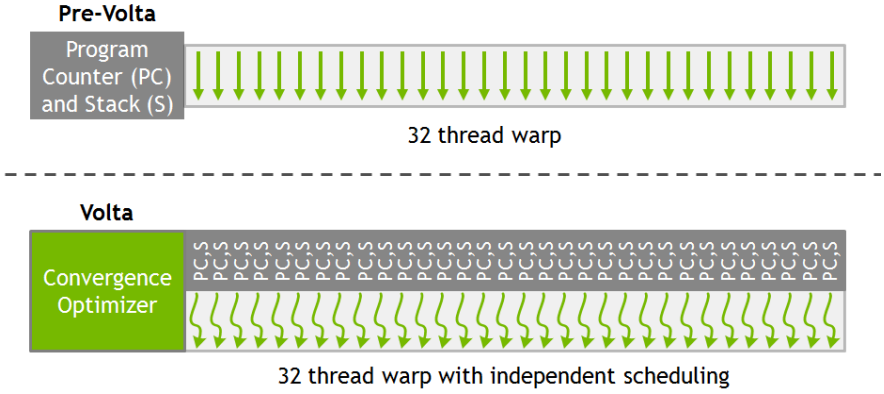

Figure 2.10: Independent thread scheduling architecture in Volta vs the pre-Volta architecture, taken from [22]

texture and L1 cache have integrated into one unit memory. This new memory model allows doubling bandwidth and capacity of the L1 cache.

2.2. Programming Model of GPUs

The two most widespread programming models for GPU platforms are Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL). These two programming models significantly simplified the programming of the non-graphics programs for GPUs [14].

Both models provide the same paradigms and principles using Single Instruction Multiple Thread (SIMT) model. SIMD and SIMT are two parallelisation models on which multiple execution units run the same operations on different data elements concurrently. In the SIMD model, the same instruction runs on a vector of data by different execution units in parallel. However, in SIMT model, which is evolved from SIMD, every execution units have its own register units, addresses and flow paths.

Both CUDA and OpenCL provide a three-tiered hierarchy of execution. In this thesis, we use the CUDA terminology to avoid any misunderstanding. In CUDA, every thread runs on one core with its own thread ID in its relative thread block, per-thread private memory, program counter, registers and data. Every block is a group of threads working together in execution units called warps; usually, a warp is comprised of 32 threads. All the warps in a block execute on the same SM. Each warp executes the same instruction based on the SIMT model. Threads in a same block can communicate with each other through synchronization and shared memory.

A grid is a group of blocks from the same kernel that can synchronize with other kernels through barriers and communicate using the global memory. Therefore, one or more kernels can run in one GPU in parallel. Figure 2.13 shows the execution model in the CUDA platform.

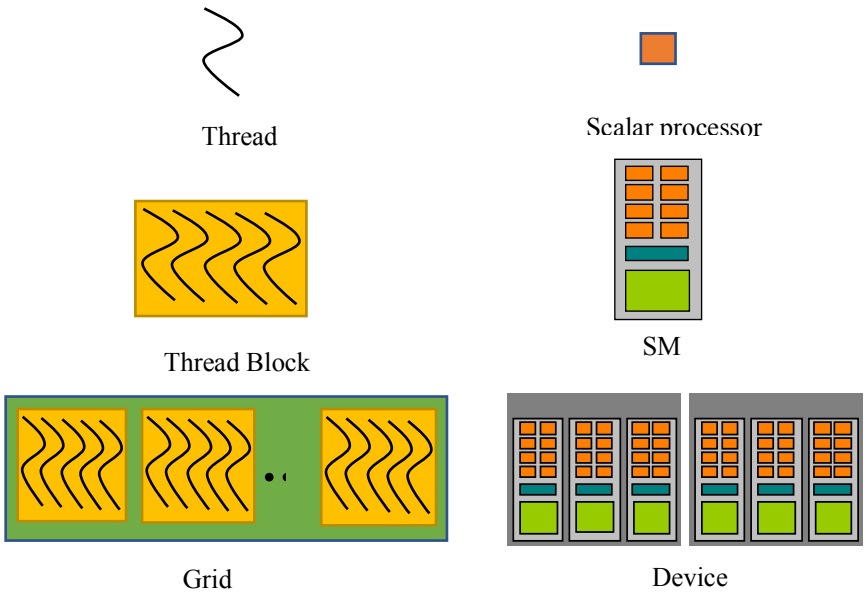

Figure 2.13: Execution model from CUDA environment to GPU, taken form [11]

In CUDA terminology, “device” refers to the GPU and “host” refers to the CPU. Similarly, the code running on GPU and CPU is called device code and host code, respectively. Device code is written for one thread and executes by all threads. GPU memory management, e.g. memory allocation, memory deallocation, and kernel calling are done by host code.

2.3. DVFS

DVFS is the combination of two power management techniques called Dynamic Voltage Scaling (DVF), the ability to scale the processor voltage, and Dynamic Frequency Scaling (DFS), the ability to scale the working frequency. DVFS aims to conserve energy consumption with minimum influence on performance by decreasing the frequency or voltage, although it can also be used to improve performance by increasing the operating frequency of processor. It is a common technology used in a variety of processors in different environments such as data centers and embedded system with a limited source of energy. This technology is called SpeedStep [27] and Cool’s’ Quiet [] by Intel and PowerNow [28] and PowerTune [29] by AMD [30].

DVFS is a hardware technique, however, the decision about when to scale the frequency or voltage of the processor is done by software. Although it should be possible to scale voltage according to the DVFS definition, it is usually frequency scaling that it is controlled via software. In this method, processor can work with a constant voltage at different working frequencies, or with a constant working frequency with different voltages. This concept is shown in Figure 2.14 by calculating energy per operation of the SA-1100 processor for different voltages and frequencies [31].

Complementary metal–oxide–semiconductor (CMOS) technology is commonly used in the current computer systems, especially in processors. The power consumption in CMOS logic circuits is composed of three components: dynamic, static, and short circuit. Traditionally, dynamic power dissipation, which is caused by transistors switching, is considered as the main power consumption in these circuits. The dynamic power consumption is calculated by (2.3) [24]:

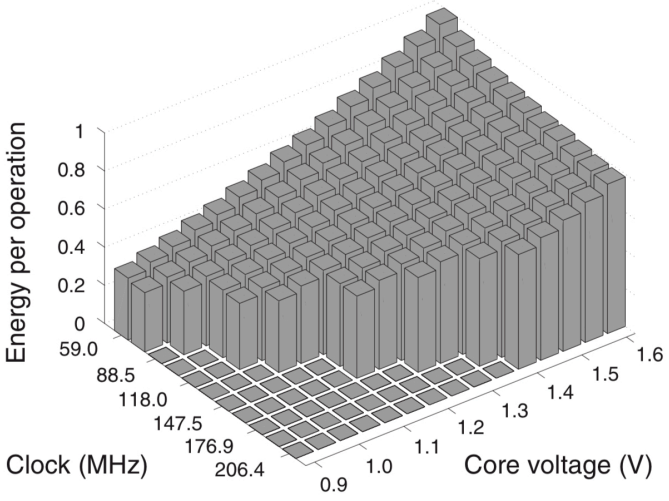

Figure 2.14: Energy per operation of SA-1100 for different voltages and frequencies, taken from [31]

$$P_{dynamic} = ACV^2 f \tag{2.3}$$

Here, A is the number of switches per each clock cycles, C is the total capacitance load, V represents the supply voltage, and f indicates the frequency in which the processor is working. Energy consumption denoted as E is given by (2.4), where we represent $P_{dynamic}$ by P.

$$E = \frac{P}{f} \tag{2.4}$$

Furthermore, working frequency have almost a linear relationship with supply voltage. This relationship is presented with Sakurai's alpha-power model [7] as:

$$f = k.\frac{(V_{dd} - V_{th})^2}{v_{dd}} \tag{2.5}$$

In equation (2.5), k is a constant number, V_{th} is the threshold voltage transistor, and V_{dd} represents the supply voltage. According to the equations (2.3), (2.4) and (2.5), power consumption and energy consumption have a cubic and quadric relationship, respectively, with supply voltage [8].

$$P \approx V^3 \tag{2.6}$$

$$E \approx V^2 \tag{2.7}$$

$$T = \frac{1}{f} \approx \frac{1}{v} \tag{2.8}$$

Dynamic power dissipation can be throttled by DVFS-based algorithms. We can reduce dynamic power consumption by reducing voltage or working frequency. But, the execution time will increase linearly when we decrease the voltage or frequency. Hence, DVFS can be exploited to control and throttle the performance or power consumption with scaling the voltage or operating frequency.

As an example, assume that a compute-bounded program can execute with a working frequency of f in 10 second (t). The total energy consumption can be calculated by (2.9), where E and P represent the energy consumption and power consumption, respectively. The frequency can go down from f to $\frac{f}{2}$, if we can decrease supply voltage from v to $\frac{v}{2}$. Therefore, the total execution time increases to 20 seconds. As a result, the revised power consumption (P') and energy consumption (E') can be calculated by (2.9) and (2,10) as follow:

$$E = P.t \tag{2.9}$$

$$P' = AC\left(\frac{V}{2}\right)^2\left(\frac{f}{2}\right) = \frac{1}{8}P \tag{2.10}$$

$$E' = P'.(20) = \frac{1}{8}P(20) = \frac{1}{4}E \tag{2.11}$$

This shows how we can improve energy-efficiency in CPUs by scaling down working frequency or voltage. However, this is not always the case for GPUs because when we are talking about power consumption in GPUs, we are taking into account the entire GPU board including, memory, processing units and many other supporting chips. Therefore, the power consumption model in GPUs consists of more than only processing units. To the best of our knowledge, there is no specific relation between energy consumption and working frequency for a specific program in GPUs.

2.4. Auto-tuning

One of the most important questions before running every GPU code is which initial parameter values, e.g. grid size and block size, can provide the optimal results for our given goal, e.g. performance or energy consumption. Finding the best configuration of an application's parameters is time consuming and challenging for programmers. This is because the search space of all possible configurations can be very large. Selecting the best configuration automatically is called auto-tuning. In this thesis, we define auto-tuning using a modified version of the definition presented in [26]:

> "Auto-tuning is an optimization problem with the objective of finding automatically good statistics parameter settings to provide the best value for a given goal before the execution of a program with a constant problem size on a specific platform"

Auto-tuning techniques can be divided into main groups: empirical and non-empirical (model-based) [25]. In the empirical method, an auto-tuner searches the optimal configuration among the entire search space using some optimization strategy. In the non-empirical method, an auto-tuner uses some models to predict performance. In this thesis, we are going to use the empirical auto-tuning.

Chapter 3

Experimental Analysis

The aim of this chapter is to present our experiments and analyze the results. First off, we describe the methodology used for the experiments and the experimental setup used throughout the thesis. We then describe the benchmarks used for our experiments. In the next section, we analyze the results to find out the impact of tuning power capping on average power consumption, energy consumption, energy efficiency, performance, memory frequency, and core frequency. Afterwards we discuss the pros and cons of our auto-tuning methodologies. Then, we present the related works in the last section.

In this section, we are going to define few concepts to be able to analyze the results. We use a brute-force search to find the configuration which provides the optimal value for an objective function. In this thesis, we defined four objective functions: power consumption, energy consumption, energy-efficiency, and performance. Possible combinations of values for tunable parameters form the search space (S), can be categorized into two groups: the set of kernel parameters (C) and the set of GPU parameters (G). If we denote the configuration of the tunable parameters, which provide the optimal value for a given objective function, with $C_{optimal}$, then:

$$C_{optimal} = \{C_{kernel}, C_{GPU}\}, where\ C_{GPU} \in G\ and\ C_{kernel} \in C \tag{3.1}$$

Here, kernel configuration, combination of kernel parameter values, and GPU configuration, combination of GPU parameter values, for optimal score are denoted with C_{kernel}, and C_{GPU}, respectively. The kernel configuration consists of a set of different parameters (e.g. block size, grid size) which is dependent on the program. In this thesis, we are going to focus mostly on the GPU configurations.

Power capping refers to the ability to limit the GPU power consumption within a specific power budget [35]. Power capping on GPUs can provide a new opportunity to achieve better values for power-related objective functions. We propose three different sets of tunable parameters for GPU configuration: (1) core frequency, memory frequency and power capping (2) core frequency and memory frequency, and (3) power capping value. For the sake of simplicity, we shall call these three strategies of the tunable GPU configurations by first, second and third strategy, respectively. Hence, if we denote core frequency, memory frequency and power capping with $f_{core}, f_{memory}\ and\ p_c$, respectively, we can show these three strategies by (3.2), (3.3), and (3.4).

$$G1 = \{f_{core}, f_{memory}, p_c\} \tag{3.2}$$

$$G2 = \{f_{core}, f_{memory}\} \tag{3.3}$$

$$G3 = \{p_c\} \tag{3.4}$$

It should be noted that when a specific power capping value is set, the memory/core frequency can be altered based on the internals of the GPU. This happens because the GPU decreases the frequency of memory or cores to be able to keep the total power consumption of the entire GPU board below the power-capping threshold. Therefore, in a GPU with a set power cap, core frequency and memory frequency work as of **"*core frequency capping*"** and **"*memory frequency capping*"** where set. Therefore, the GPU requirement in terms of power capping has more priority than the predetermined value of the core/memory frequency. As the best of our knowledge, there is no study regarding the impact of GPU-level power capping. Hence, we are going to mostly focus on the impact of power capping parameter on GPUs.

The average power consumption of each configuration is always less than the power capping level. The sequence of the captured instantaneous power values gradually fluctuates during power profiling. If the average power consumption is close to the power capping level, then there are likely several instantaneous power values which reach to the power capping level in the course of execution. We define a 10-watt interval to discover the instantaneous power values of which configuration may reach to the power capping level at runtime.

As an example, if the power capping level is 100 watts and the average power consumption of a given configuration is in the range of 90 to 100 watts, then we assume that the power capping parameter could influence this configuration. This is vital for us since we would like to find out which configuration might impact our objective functions using the first or third GPU configuration strategy.

The number of all possible configurations for a given program is the ***search space size***. We define the ***power capping size*** as the number of configurations with average power consumption close to the power capping level, and the ***power capping ratio*** as the power capping size over the search space size. When the average power consumption in a given configuration of a program is not close to the set power capping level, then applying power capping on GPU is not able to influence the obtained objective function value (for this specific configuration and program). Hence, in this case, only altering the core and memory frequencies can impact the results.

Theoretically, the first strategy should provide the best values for each objective function. It can be explained by the fact that the search space of the second and third strategies are a subset of the first one.

3.1. Experimental setup

In this section, we explain what is the experimental setup and how the experiments are carried out. We start by describing the underling hardware we used and the tools to perform power profiling. Next, we describe how we deal with issues related to small execution time and temperature.

We performed the experiments on the VU cluster of the Distributed ASCI Supercomputer 5 (DAS-5) [32], a distributed supercomputer with six different clusters. A standard node in DAS-5 is equipped with a dual 8-core Intel Haswell E5-2630v3 CPU and 64MB memory with the operating system being CentOS Linux 7. The machines in each cluster communicate through 1 Gbit/s Ethernet interfaces.

Additionally, DAS-5 contains special nodes with one or more GPUs from different generations. Table 4.1 lists comprehensive information regarding some available GPUs on the DAS-5. In this table, along with the name of every platform, we show its architectural family, theoretical peak performance for single and double precision floating-point operations, memory bandwidth, the ratio between peak performance and peak bandwidth, supported sampling frequency of the instantaneous power measurement, (minimum, maximum and default) supported power cap, and TDP.

Nvidia provides two utilities to monitor and manage the GPUs devices: Nvidia Management Library (NVML) and Nvidia System Management Interface (nvidia-smi). The former is a C-based tool on top of NVML which allows to alter the core and memory frequencies and set a power cap. For power profiling, we use nvidia-smi which can measure instantaneous die temperature and power draw using the built-in power sensors, with error measurement in the range of +/- 5 watts [33]. The measured power draws are obtained from all the different components in the entire GPU board.

The supported sampling frequency of the instantaneous power draw was experimentally measured for each GPU. For instance, the power value refresh rate for Nvidia Tesla K20m is about 15 ms. As a result, we have to sample the power draw at least with a rate of 1/15=66 Hz. If we measure the power draw more frequently, we do not get any new value. On the other hand, if we measure the power draw not frequently enough, it is possible to miss a new power measurement value.

Model		**Nvidia Tesla K20m**	**Nvidia GTX Titan (GeForce)**	**Nvidia GTX TitanX**	**Nvidia GTX980**	**Nvidia GTX TitanX (GeForce)**
Generation		Kepler	Kepler	Maxwell	Maxwell	Pascal
Theoretical Peak Performance with Default Frequency	**Single Precision (GFLOPS)**	3524	4709	6691	4981	10974
	Double Precision (GFLOPS)	1175	1570	209	155	3429
Bandwidth GB/S		208	288	336	224	480
FLOPS/B (Single Precision Point)		16.9	16.3	19.9	22.2	22.8
Supported Sampling Frequency of The Instantaneous Power Measurement = (1/Refresh Rate) Hz		1/15ms=66	1/100ms=10	1/100ms=10	1/100ms =10	1/20ms=50
Minimum Supported Power Capping (Watt)		150	150	150	100	125
Maximum Supported Power Capping (Watt)		225	265	275	225	300
Default Power Capping (Watt)		225	250	250	180	250
TDP (Watt)		225	250	250	165	250

Table 3.2: Characteristics of the platforms used in the thesis

To ensure enough samples, especially for kernels with short execution time, we can either use a bigger problem size or repeat the execution for several times. Because finding the best configuration is also dependent on the problem size, we apply the latter method to increase the kernel run time. To do so, we run the kernels a number of times enough to obtain at least 1000 power measurements, given the measured power sample frequency.

The power consumption of each kernel for a specific configuration is computed as the average of all power measurements, while the total energy consumption for a kernel is computed as the product of average power consumption to average execution time. To guarantee the accuracy of the results, we run all programs two times, and use the average values.

Kernel Tuner is an open-source tool made by Ben Van Werkhoven to ease the testing and auto-tuning of CUDA, C and OpenCL kernels [34]. Programmer specifies the supported input parameter through a Python interface and then Kernel Tuner automatically finds the best configuration in the search space using a specified search strategy. We modified and used the Kernel Tuner to do power profiling for each configuration.

The GPU configurations are set using the nvidia-smi tool. To reduce the size of the search space, we set a 10-watt interval between two consecutive power capping values, and also set a 100 MHz step between core frequencies. Additionally, GPU Boost is disabled to keep the selected core/memory frequency constant during runtime.

We noticed that the GPU temperature remains constant if we repeatedly run the same kernel for a while. We use this concept to mitigate the temperature-related effects on power profiling. To do so, we experimentally selected a 10-second safeguard period between each kernel configuration and a 45-second period between each GPU configuration. During the 10-second safeguard, the GPU increases/decreases its temperature to a stable value by repeatedly running the relative kernel. The GPU configuration safeguard is larger because changing the core/memory frequency has a larger impact on temperature.

For example, the temperature of the Nvidia K20 with a core/memory frequency of 324/324 MHz and a fixed kernel configuration executing matrix multiplication is about 25 degrees, while it is about 35 degrees when the core/memory frequency is increased to 758/2600, however, the maximum measured difference between different kernel configurations was about 4 degrees.

3.2. The Benchmarks

In this section, we introduce three benchmarks (i.e. Stencil, Matrix Multiplication, and vector-add) to evaluate the auto-tuning impact for the determined objective functions on the six different Nvidia GPUs in DAS-5. We selected these benchmarks, available in Kernel Tuner, since they have a reasonable number of kernel search space, and the power consumption of the different configurations can reach the maximum allowed power consumed by the GPU at any given time.

These programs are compiled and benchmarked based on CUDA 9.00 and Python 3.5.2. The studied applications with their characteristics are listed in Table 4.2. Based on the

computed arithmetic intensity and the used platforms, all the studied benchmarks are memory-intensive.

3.2.1. Stencil

Stencil kernels are a class of common codes used in different fields such as computer simulations [43] and cellular automata [42] where every array element is iteratively updated according to a fixed pattern. In our benchmark, a $2D$ matrix with the size of $n \times m$ is used as input, with each element of the output matrix being the average of five neighbouring input elements. In CUDA version of this code, each thread should perform 5 operations for each 6 single-precision floating-point (4 bytes) memory accesses. As a result, the Arithmetic Intensity[1] (AI) of the stencil code is $\frac{5}{24} = 0.2$. There are two tunable kernel parameters in this application namely, the block size in x axis and the block size in y axis. The listing 3.1 shows this stencil CUDA code.

3.2.2. Matrix Multiplication

Matrix multiplication code is used to compute the product of two square matrices with the same size. There are different implementations of matrix multiplication, we used one optimized using the tiling technique. Block thread dimensions and tiling factors in both *x* and *y* axes are the four tunable kernel parameters in this application. The computed arithmetic intensity of this program is $\frac{1}{6}$,thereby categorizing as a memory-intensive program.

```
1.  #define domain_width    4096
2.  #define domain_height   2048
3.
4.  __global__ void stencil_kernel(float *x_new, float *x_old) {
5.      int x = blockIdx.x * block_size_x + threadIdx.x;
6.      int y = blockIdx.y * block_size_y + threadIdx.y;
7.
8.      if (y>0 && y<domain_height-1 && x>0 && x<domain_width-1) {
9.
10.     x_new[y*domain_width+x] = ( x_old[ (y  ) * domain_width + (x  ) ] +
11.                                 x_old[ (y  ) * domain_width + (x-1) ] +
12.                                 x_old[ (y  ) * domain_width + (x+1) ] +
13.                                 x_old[ (y+1) * domain_width + (x  ) ] +
14.                                 x_old[ (y-1) * domain_width + (x  ) ] ) / 5.0f;
15.
16.     }
17. }
```

Listing 3.1: The stencil CUDA code

[1] Arithmetic Intensity is the ratio between arithmetic, floating-point, operations and bytes accessed in memory.

3.2.3. Vector-add

Vector-add is a program that sums two input vectors, sorting the output in a third vector. This application is memory-intensive, for almost all the platforms, with AI of $\frac{1}{12} = 0.03$. This is the simplest example with the dimension of block size in *x* axis as the only tunable kernel parameter.

3.3. Experimental Analysis

In this section, we present and analyze the results of the experiments described in the previous sections. Our goal is to experimentally find the set of parameter values, kernel parameter values and GPU parameter values, which provides the best score for an objective function.

Moreover, we would like to know whether the GPU-level power capping can be exploited as a tunable parameter or not, to address questions like: what would be the impact of the power capping level on different objective functions? Can we find anything to help us to find the optimal power capping value? What is the advantage and disadvantage of each GPU configuration strategy?

The metrics used to express performance and energy efficiency, for matrix multiplication and stencil benchmarks, are the number of single-precision floating-point operations per second, and the number of single-precision floating-point operations per watt, respectively. However, we use the number of transferred byte from/to memory per second and per watt for the corresponding units in vector-add benchmark since it is a completely memory-bound application. The studied programs are executed on every platform, for each possible combination of kernel and GPU parameters. After each set of experiment, we present an overview of our main findings.

3.3.1. Auto-tuning on Nvidia Tesla K20m

The first GPU we use is the Nvidia Tesla K20m. Running our benchmarks with all the possible tunable parameters on this platform never reached the power capping level. Hence, we are not able to determine how the first and the third GPU configuration strategies can affect the objective function values. To avoid any possible error measurement, we consider the top three records for each objective function.

Benchmark	Arithmetic Intensity (AI)	Compute-intensive/Memory-intensive	Problem Size	Number of Kernel Parameters
Stencil	$\frac{5}{24}$	Memory-intensive	4096 × 2048	2
Matrix multiplication	$\frac{1}{6}$	Memory-intensive	4096 × 4096	4
Vector-add	$\frac{1}{12}$	Memory-intensive	1×10^7	1

Table 3.2: An overview of the benchmarks

In Table 3.3 we present the optimal values for every objective function, highlighted by gray cells, with the corresponding configurations. The optimal value of the energy consumption and the average power consumption should be minimum scores; however, the optimal values for energy-efficiency and performance are the highest values. Based on this table, we can derive the following conclusion which are also valid for the other GPU platforms based on our experiments.

- The tunable parameter values to obtain the best score depends on the objective function. For instance, the tunable parameters to achieve the best performance in matrix multiplication benchmark should be 32, 8, 4, 4, 758, 2600 for the parameters of block_size_x, block_size_y, tile_size_x, tile_size_y, f_{core}, f_{memory}, respectively. However, the parameters to obtain the lowest total energy consumption are 64,8,2,8, 614, 2600.

In the rest of this thesis, we do not take into account the average power consumption as the objective function. It is because the minimum average power consumption is always obtained by the minimum memory and core frequencies. Moreover, it always yields the worst performance.

3.3.2. Auto-tuning on Nvidia GTX Titan

In this section, we present and analyze the auto-tuning experiments on the Nvidia GTX Titan. Since no permission had been given to alter the core and memory frequencies on this platform,

		Benchmark											
		Vector-add				**Stencil**				**Matrix Multiplication**			
Kernel Parameters Configuration	**block_size_x**	960	192	256	192	32	256	256	256	16	64	64	32
	block_size_y	-	-	-	-	1	1	1	1	2	8	8	8
	tile_size_x	-	-	-	-	-	-	-	-	8	2	2	4
	tile_size_y	-	-	-	-	-	-	-	-	8	8	8	4
GPU Configuration	**f_{core} (Mhz)**	324	614	640	758	324	640	758	758	324	614	758	758
	f_{memory} (MHz)	324	2600	2600	2600	324	2600	2600	2600	324	2600	2600	2600
	p_c	-	-	-	-	-	-	-	-	-	-	-	-
Objective function	**Average Power Consumption (mW)**	39.8	102	107	120	32.3	91.7	110.6	110.6	39.6	93	119	132.9
	Energy Consumption (mJ)	233.5	81.8	82	89	913	265.3	270.9	270.9	61464	25069	26352	28838
	Energy-efficiency (GFLOPS/W) (GB/W)	87	1837	1912	1800	1.62	54.65	63.2	63.2	1.4	20.4	23.55	21.7
	Performance (GFLOS/S) (GB/S)	20.4	150	157	161.6	1.5	14.5	17.1	17.1	88	512	620	633.5

Table 3.3: Optimal values for every objective function, highlighted by gray cells, with the corresponding configurations. for every benchmark on Nvidia Tesla K20m.

we could not provide any experiments regarding the first and second GPU configuration strategies. Hence, our findings here are based on the third GPU configuration method.

We conclude that applying different power capping level on Nvidia GTX Titan can influence any given objective function. It is because matrix multiplication, stencil and vector-add codes have the power capping ratio of 50%, 24%, 44%, respectively. In the figures 3.1 to 3.9, the relationship between power capping level and energy consumption, energy efficiency and performance values are depicted for all benchmarks. Values are representative for the objective function value of one execution round under a specific power budget for a given configuration. In these Figures, the lower part of the scatter plots related to energy consumption, and the upper part of the scatter plots related to the energy efficiency and performance are important for us. It is because lower energy consumption, higher energy-efficiency and higher performance are always desirable. We observe that the best power capping level is different for each objective function and benchmark. This leads us to conclude that the best power capping value depends both on the application and the objective function.

3.3.2.1. Impact of tuning power capping level on energy consumption, energy efficiency and performance on Nvidia GTX Titan

Our first experiments measure the energy consumption, energy-efficiency and performance of the three benchmarks for all the combination of kernel and power capping level using the third GPU configuration strategy on the Nvidia GTX Titan. The goal of these experiments is to tune the kernel and GPU parameters to find the configuration which yield the best objective function value. In this section, we are interested in providing an analysis of the impact of tuning power capping level on these objective functions.

As illustrated in Figure 3.1, the total energy consumption for matrix multiplication follows a slight upward trend with the power capping level and then it levels off when the power capping value reaches 200 watts. We can see a similar pattern in stencil and vector-add benchmarks. In these figures, the minimum energy consumption is achieved by the lowest power capping level (150 watts).

These scatter plots can provide insights into to the first and second GPU configuration strategy. For example, we expect to see a similar pattern in the scatter plots of power capping and energy consumption using the first strategy. It is because the search space of the third GPU configuration strategy is a subset of the search space of the first GPU strategy. In section 3.4.3.4, the experiments, obtained on the Nvidia GTX980, verifies this assumption.

Moreover, these plots show that we obtain different energy consumption values for the same power capping level. This is because for every power capping level, there are different combination of kernel configurations, each leading to different total energy consumption. This observation applies to other objective functions as well.

As shown in Figure 3.4- 3.6, the difference between the maximum energy efficiency obtained by any power capping value is negligible. It means that the power capping level could slightly influence the energy-efficiency for matrix multiplication and stencil benchmark. However, this is not the case for the vector-add program. The optimal energy-efficiency decrease when the power capping level increase and then it stays constant. The optimal energy-efficiency is achieved using power capping value of 150 watts, for all these benchmarks.

In Figures 3.7 – 3.9, we observe that performance increases for every program when the power capping level goes up. We expect this behaviour because when the power budget improves, there is more power to increase the memory and core frequency thereby, executing the benchmarks faster. Then, the performance levels off since all of these benchmarks are memory-intensive and increasing the power budget does not lead to increasing memory frequency and then performance.

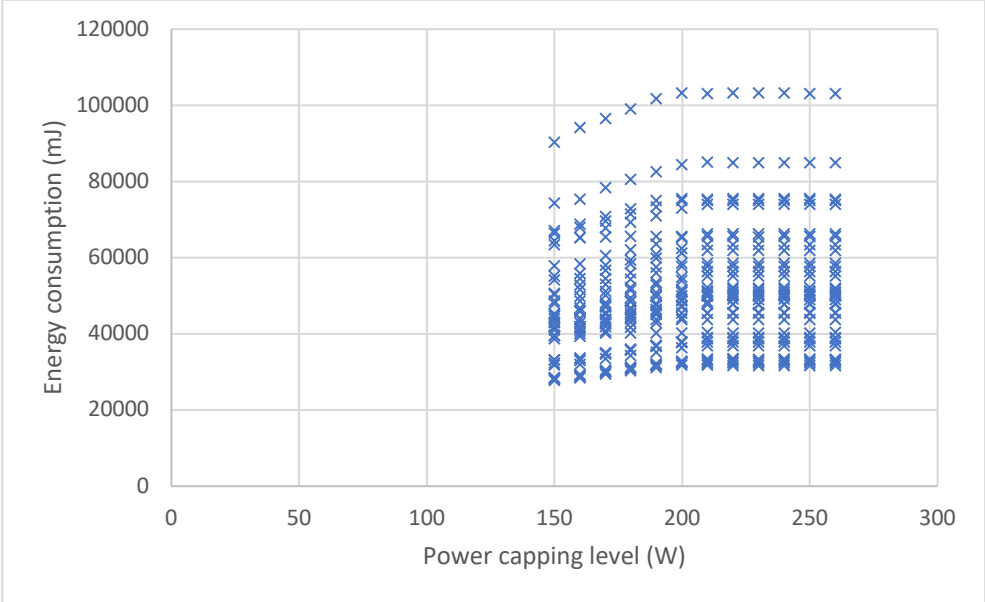

Figure 3.1: Impact of tuning power-capping level on energy consumption using the third strategy for matrix multiplication benchmark on Nvidia GTX Titan. Values are representative for the total energy consumption (mJ) of one execution round under a specific power budget for the given configuration

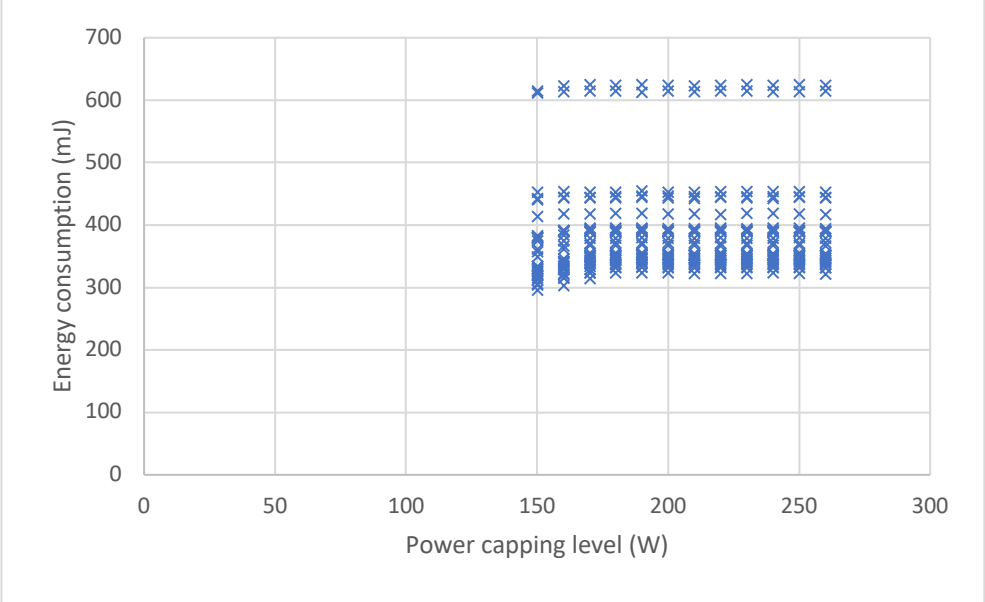

Figure 3.2: Impact of tuning power-capping level on energy consumption using the third strategy for stencil benchmark on Nvidia GTX Titan. Values are representative for the total energy consumption (mJ) of one execution round under a specific power budget for the given configuration

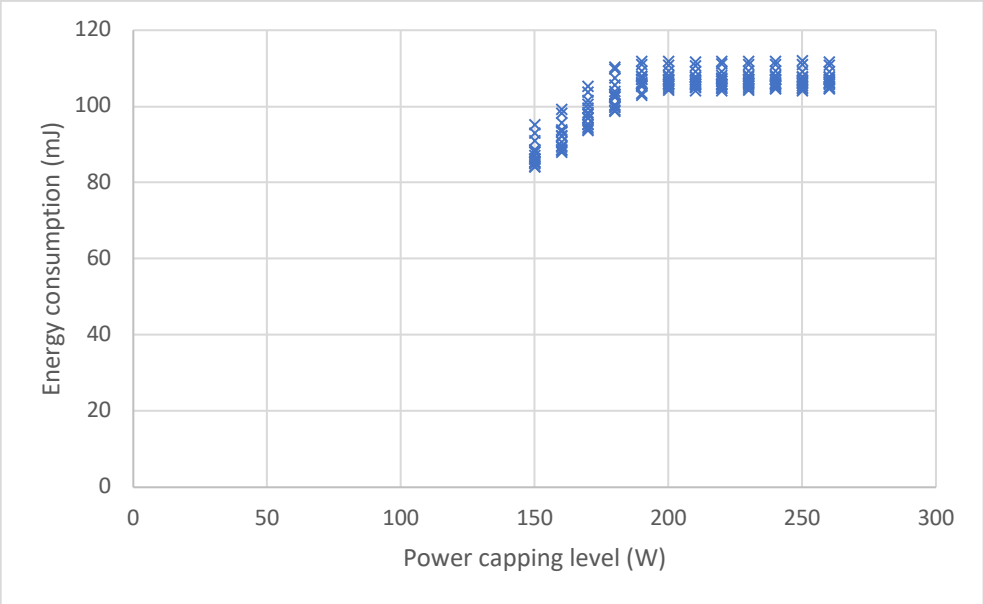

Figure 3.3: Impact of tuning power-capping level on energy consumption using the third strategy for vector-add benchmark on Nvidia GTX Titan. Values are representative for the total energy consumption (mJ) of one execution round under a specific power budget for the given configuration

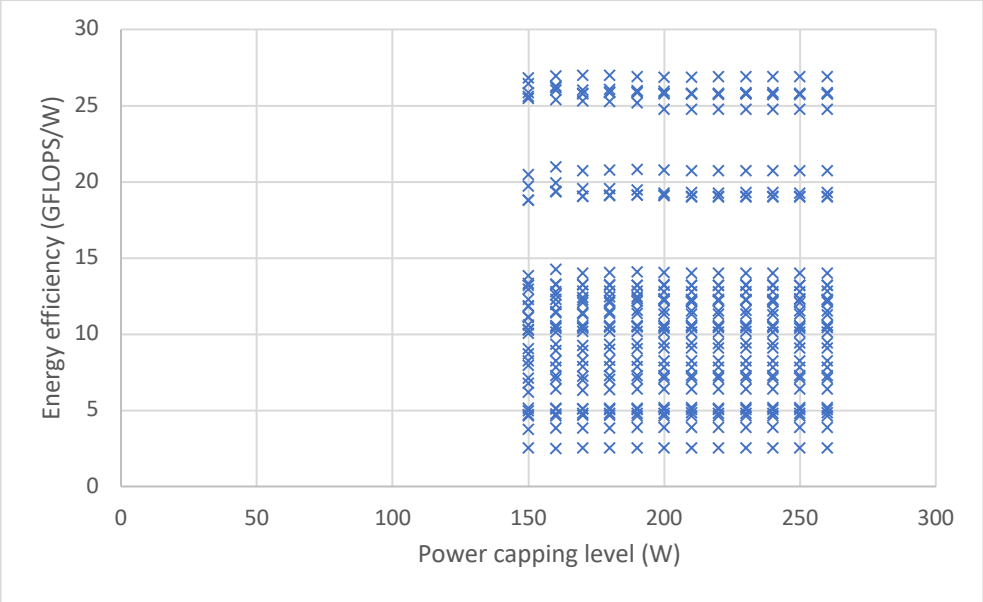

Figure 3.4: Impact of tuning power-capping level on energy efficiency using the third strategy for matrix multiplication benchmark on Nvidia GTX Titan. Values are representative for the energy efficiency (GFLOPS/W) of one execution round under a specific power budget for the given configuration

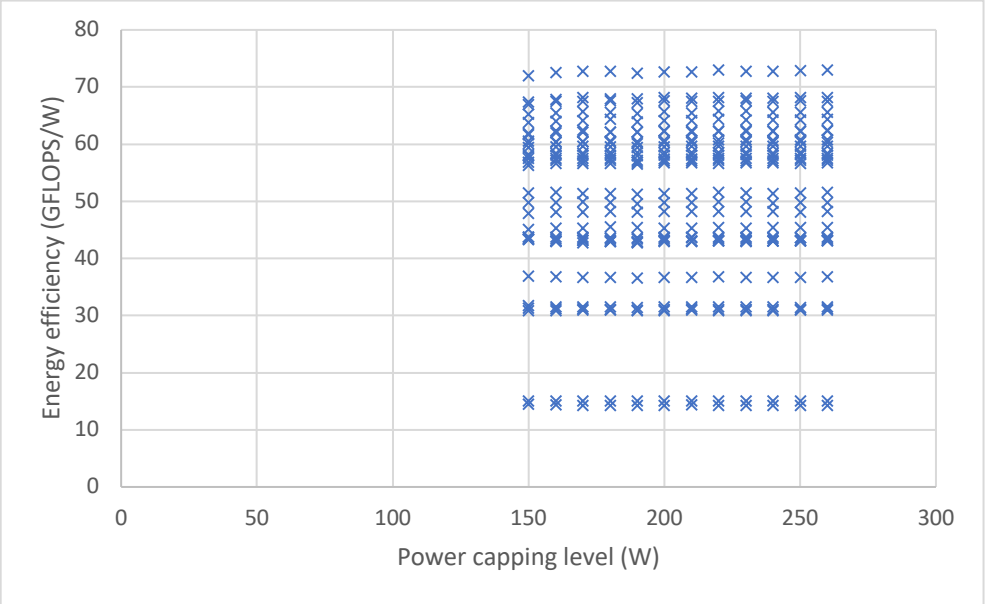

Figure 3.5: Impact of tuning power-capping level on energy efficiency using the third strategy for stencil benchmark on Nvidia GTX Titan. Values are representative for the energy efficiency (GFLOPS/W) of one execution round under a specific power budget for the given configuration

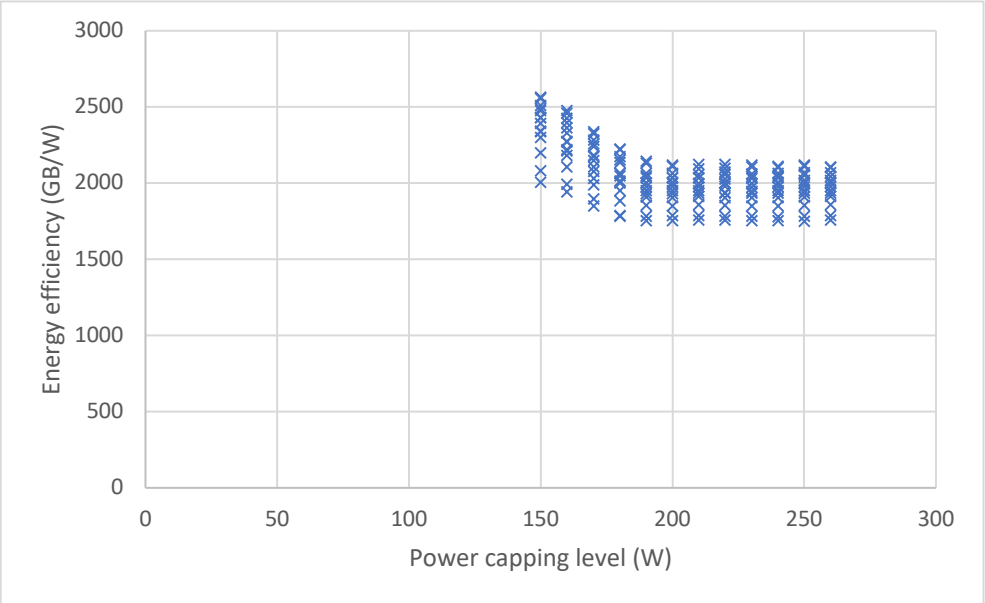

Figure 3.6: Impact of tuning power-capping level on energy efficiency using the third strategy for vector-add benchmark on Nvidia GTX Titan. Values are representative for the energy efficiency (GB/W) of one execution round under a specific power budget for the given configuration

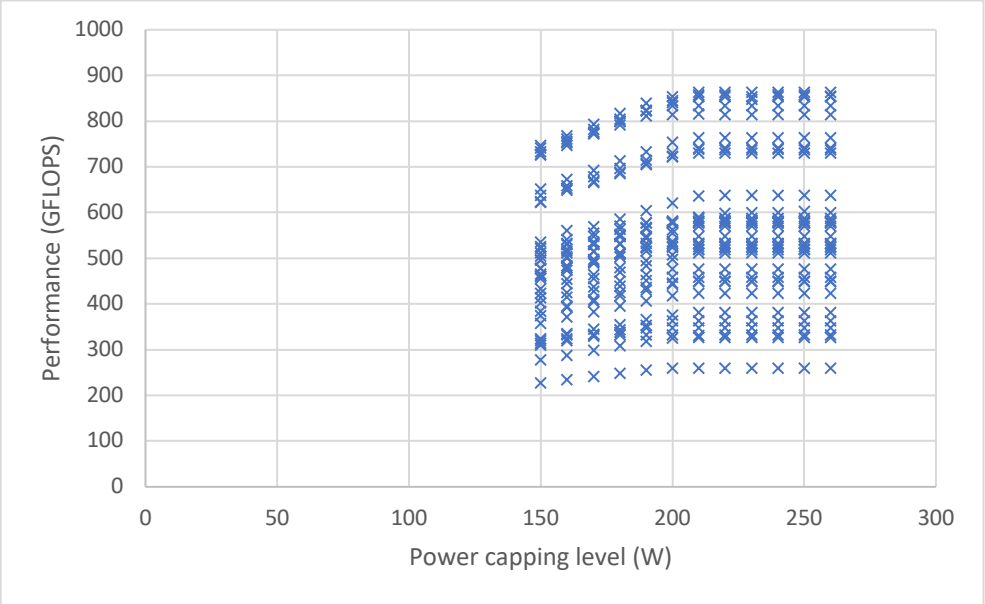

Figure 3.7: Impact of tuning power-capping level on performance using the third strategy for matrix multiplication benchmark on Nvidia GTX Titan. Values are representative for the obtained performance (GFLOPS) of one execution round under a specific power budget for the given configuration

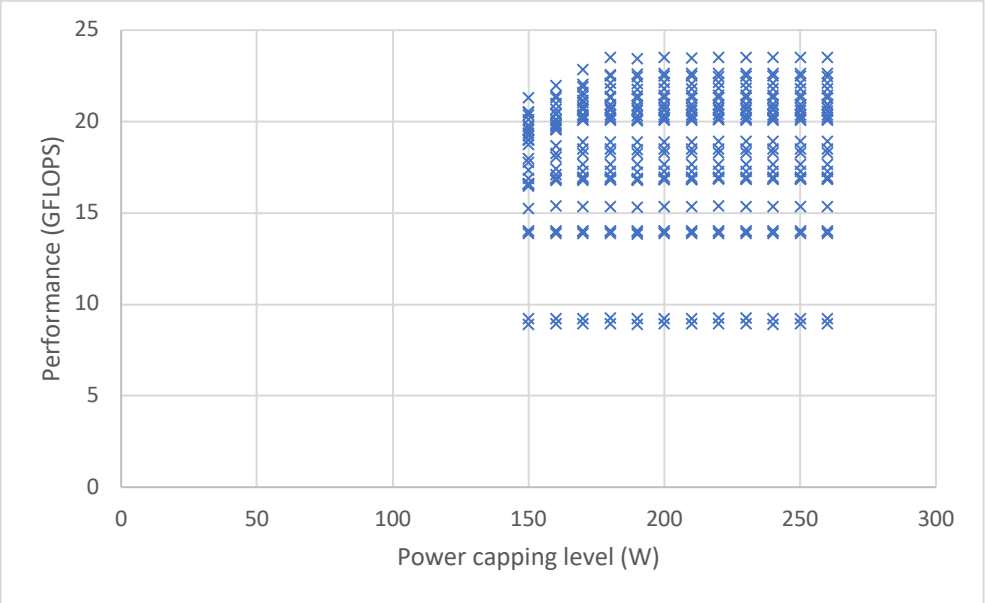

Figure 3.8: Impact of tuning power-capping level on performance using the third strategy for stencil benchmark on Nvidia GTX Titan. Values are representative for the obtained performance (GFLOPS) of one execution round under a specific power budget for the given configuration

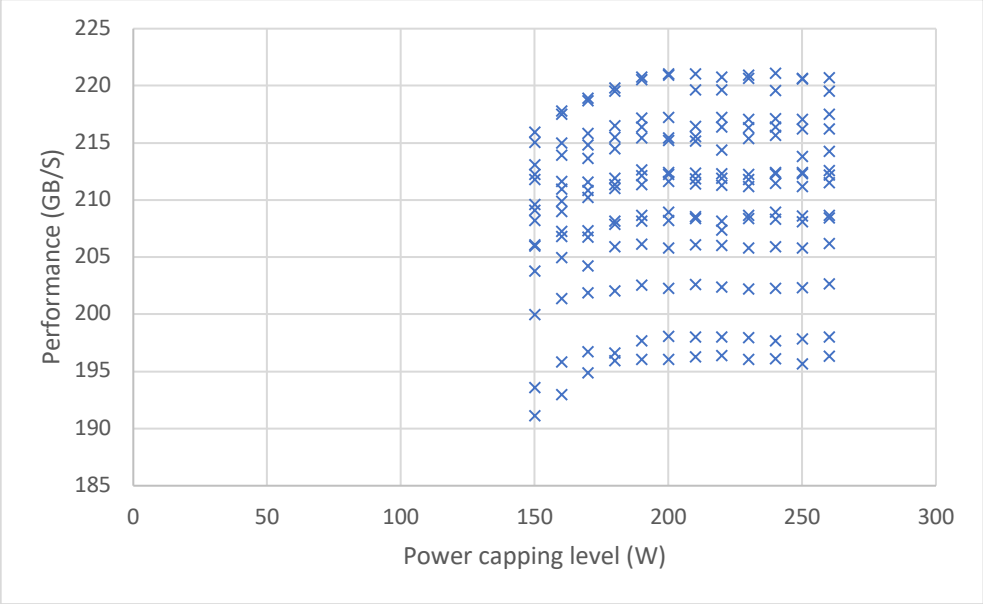

Figure 3.9: Impact of tuning power-capping level on performance using the third strategy for vector-add benchmark on Nvidia GTX Titan. Values are representative for the obtained performance (GB/S) of one execution round under a specific power budget for the given configuration

3.3.3. Auto-tuning on Nvidia GTX TitanX (Maxwell) and Nvidia GTX980

In this section, we present and analyse results on Nvidia GTX TitanX (Maxwell) and Nvidia GTX980 platforms. The first interesting question is whether applying GPU-level power capping can provide any benefit in comparison with the traditional strategy, the second one, or not. Table 4.2 and 4.3, contain the optimal achievable values of all objective functions using different GPU configuration strategies for Nvidia GTX TitanX (Maxwell) and Nvidia GTX980, respectively.

In Table 3.4, the optimal values for every objective function on Nvidia GTX TitanX (Maxwell) are obtained by either the first or the second GPU configuration methodology. We can derive that applying power capping on this GPU does not provide any benefit since the difference between the achieved optimal values using the first and second GPU configuration strategies are negligible. Accordingly, we do not further analyze the experiments on this platform. Moreover, we present Table 3.5 for comparison between these optimal values and the default configuration (with GPU boost enabled and default power capping value). This table shows that we could improve power metric values for all the benchmarks, but the improvement for performance was negligible, except for the vector-add benchmark.

On the other hand, we can improve the objective function values using the first strategy on Nvidia GTX980. This result, for example, in a significant improvement in energy consumption, in comparison to the second strategy, for vector-add, stencil and matrix multiplication to 21%, 23% and 17%, respectively. Moreover, the corresponding figures for energy-efficiency show a significant enhancement up to 23%, 11% and 9%. Interestingly, the difference between the obtained performance was negligible. It can be explained by the fact that the best performance always can obtain by the maximum core and memory frequencies, supported by maximum possible power capping level.

		Benchmark								
		Vector-add			Stencil			Matrix Multiplication		
	Strategy Number	1	2	3	1	2	3	1	2	3
Objective functions	Energy Consumption	**67**	**67**	69	**152**	**152**	178	**11037**	11074	11092
	Energy-efficiency	**4037**	4035	3733	**200**	**200**	199	**241**	240	225
	Performance	**275**	**275**	259	**40**	**40**	**40**	**2876**	2843	2744

Table 3.4: The optimal values of each objective function for all the benchmarks with different GPU configuration strategies on Nvidia GTX TitanX (Maxwell). (1), (2) and (3) indicate that the results obtained using the first, second and third GPU configuration strategy, respectively.

		Benchmark								
		Vector-add			Stencil			Matrix Multiplication		
	Strategy Number	1	2	3	1	2	3	1	2	3
Objective functions	Energy Consumption	**28%**	**28%**	24%	**35%**	**35%**	15%	**1.5%**	2%	2%
	Energy-efficiency	**34%**	34%	24%	**3%**	**3%**	2%	**9%**	9%	6%
	Performance	**6%**	**6%**	0%	**0%**	**0%**	**0%**	**2%**	2%	1%

Table 3.5: Comparison between the optimal value and default GPU configuration for all the benchmarks on Nvidia GTX TitanX (Maxwell). (1), (2) and (3) indicate that the results obtained using the first, second and third GPU configuration strategy, respectively.

		Benchmark								
		Vector-add			Stencil			Matrix Multiplication		
	Strategy Number	1	2	3	1	2	3	1	2	3
Objective functions	Energy Consumption	**61**	78	70	**145**	189	145	**9887**	11956	9925
	Energy-efficiency	**3185**	2470	2364	**200**	177	198	**187**	171	171
	Performance	**194**	193	167	**33.6**	33.5	33.2	**2114**	2051	2080

Table 3.6: The optimal values of each objective function for the all benchmarks with different GPU configuration strategies on Nvidia GTX980. (1), (2) and (3) indicate that the results obtained using the first, second and third GPU configuration strategy, respectively.

		Benchmark								
		Vector-add			Stencil			Matrix Multiplication		
	Strategy Number	1	2	3	1	2	3	1	2	3
Objective functions	Energy Consumption	**30%**	10%	19%	**23%**	0%	23%	**24%**	2%	23%
	Energy-efficiency	**40%**	28%	22%	**14%**	1%	13%	**16%**	6%	6%
	Performance	**16%**	16%	0%	**0%**	0%	0%	**6%**	3%	4%

Table 3.7: Comparison between the optimal value and default GPU configuration for all the benchmarks on Nvidia GTX980. (1), (2) and (3) indicate that the results obtained using the first, second and third GPU configuration strategy, respectively.

Moreover, we present Table 3.7 for comparison between these optimal values and the default configuration on Nvidia GTX980. This table shows that we could improve all objective function using all the GPU configuration strategy, except for the obtained performance value using the third strategy. It is because GPU configuration in the third strategy with the highest power capping value is the same as default configuration. Moreover, we could not enhance our objective function value using the second strategy for stencil code. It may be because the best memory and core frequency values here is the same as the corresponding values in the default configuration. In this section, we are going to study in detail the impact of the first and the third GPU configuration methodologies on this platform.

3.3.3.1. Impact of tuning power capping level on energy consumption, energy efficiency and performance on Nvidia GTX980

In this section, our experiments measure the energy consumption, energy-efficiency and performance for any possible configuration of all the benchmarks using the first GPU configuration on Nvidia GTX980. Then, we present and analyze the impact of tuning power capping level on energy consumption, energy efficiency and performance.

The lowest energy consumption is achieved for the matrix multiplication when the power capping value is 120 watts. However, the corresponding score for stencil and vector-add codes are 100 watts. The energy consumption behavior for matrix multiplication benchmark shows a different pattern than those of stencil and vector-add programs (see Figures 3.10- 3.12). This is because the optimal power capping level depends on the application.

Comparing Figures 3.10 and 3.1, we can observe a difference in the impact of tuning power capping levels on Nvidia GTX980 and Nvidia GTX Titan for matrix multiplication. However, the experiments for stencil and vector-add do not show any difference in the behavior these two platforms. It should be noted that we are interested in analyzing the lower part of these scatter plots since the lower energy consumption is always desirable.

As it is illustrated in Figures 3.13, energy efficiency, for matrix multiplication code, improves when the power capping value increases. It gets to its peak value (about 150 GFLOPS/W) with the power capping of 150 watts and then we observe a downward trend in the amount of performance per watt.

We can see different pattern for stencil benchmarks, the energy efficiency increase, then decrease and it stays constant (see Figure 3.14). This pattern is different than the one we described in the section 3.4.2 for the Nvidia GTX Titan platform. The relationship between the power capping level and the amount of GB/S is the same as the one obtained on the Nvidia GTX Titan, with a different optimal power capping level.

As demonstrated in Figure 3.16, we see that performance for matrix multiplication improves when power capping level increases. This behavior is expected since the power budget increase and GPU can increase with core or memory frequencies. We can see almost the same pattern for stencil benchmark.

However, the pattern for vector-add benchmark is different than one might expect. Although we can see a negligible improvement in performance (0.5%) when the power capping

value is 100 or 170, the achievable performance is almost the same for all the power capping values. Given the fact that vector-add is an absolutely memory-intensive application with low arithmetic intensity ($\frac{1}{12}$), this means the GPU used its peak memory frequency for all the power capping values and then it likely increases the core frequency when we the power budget increases.

To conclude, we could find one generic trend between power capping values and performance. Since the GPU increases the memory/core frequency when the power capping budget improves, the obtainable performance increases to a maximum value and then it stays constant. This maximum performance is determined by program specifications e.g. how much the program is compute-intensive or memory-intensive.

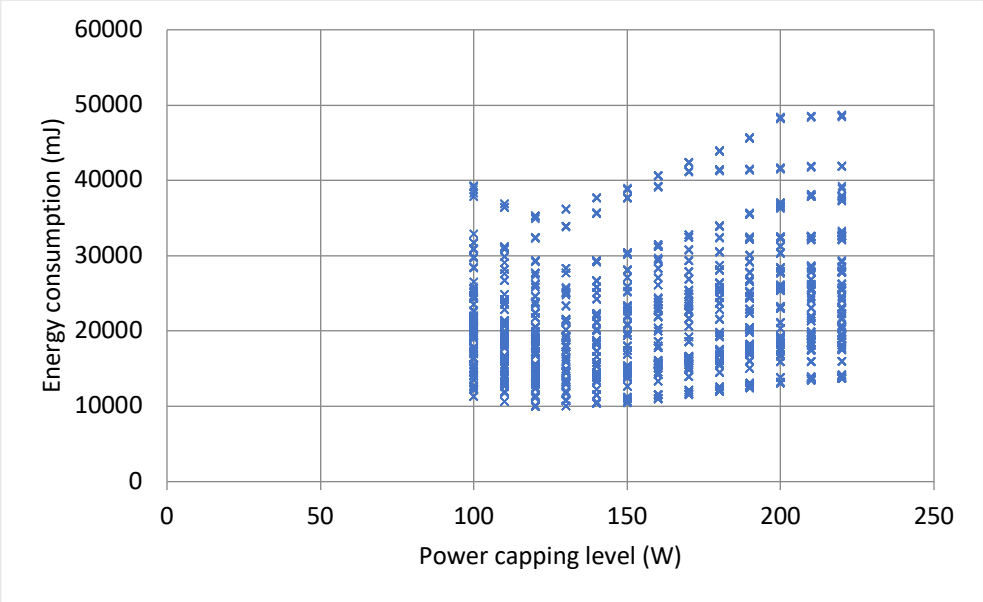

Figure 3.10: Impact of tuning power-capping level on energy consumption using the first strategy for matrix multiplication benchmark on Nvidia GTX980. Values are representative for the total energy consumption (mJ) of one execution round under a specific power budget for the given configuration

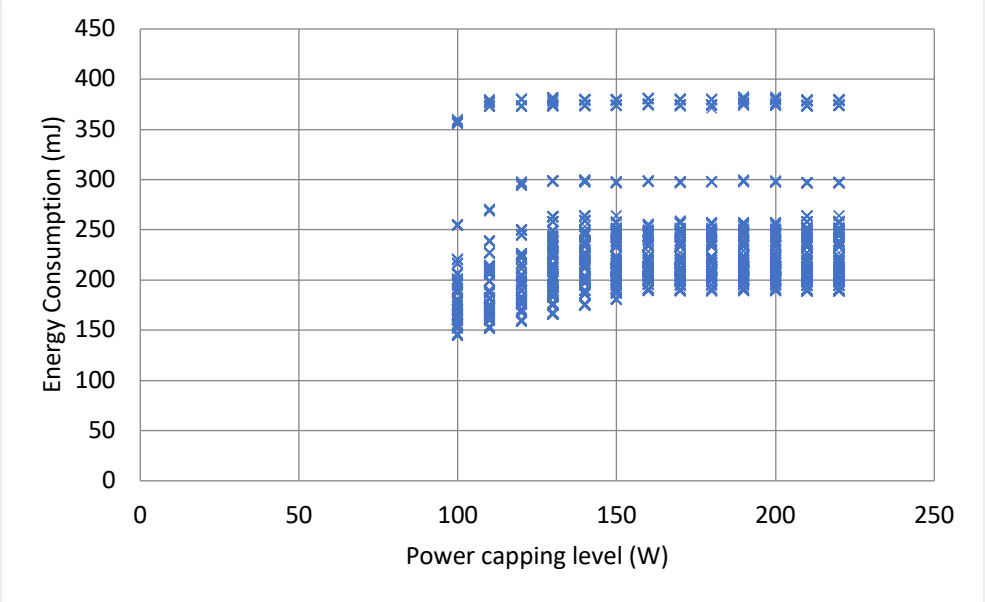

Figure 3.11: Impact of tuning power-capping level on energy consumption using the first strategy for stencil benchmark on Nvidia GTX980. *Values are representative for the total energy consumption (mJ) of one execution round under a specific power budget for the given configuration*

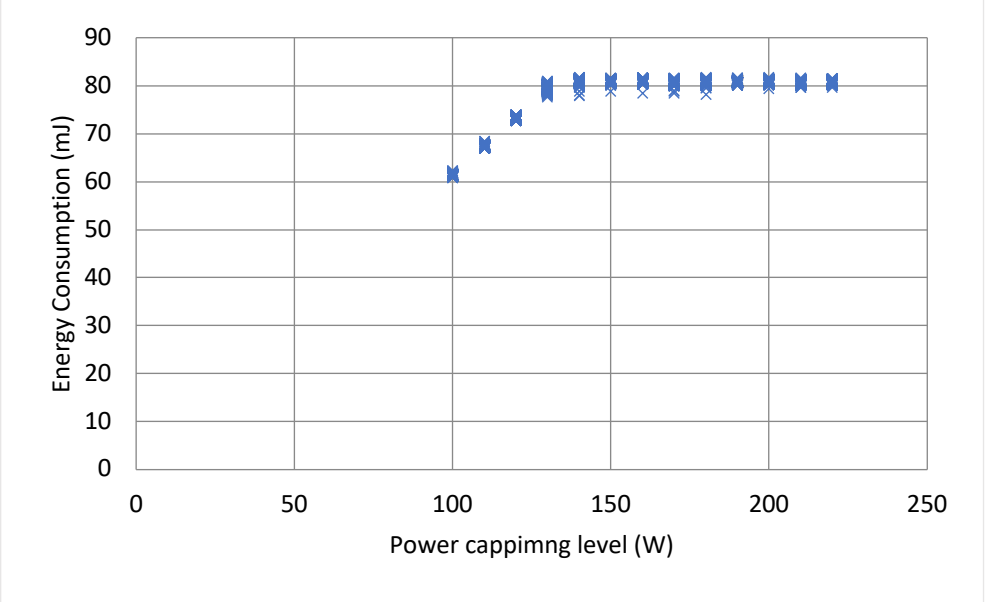

Figure 3.12: Impact of tuning power-capping level on energy consumption using the first strategy for vector-add benchmark on Nvidia GTX980. Values are representative for the total energy consumption (mJ) of one execution round under a specific power budget for the given configuration

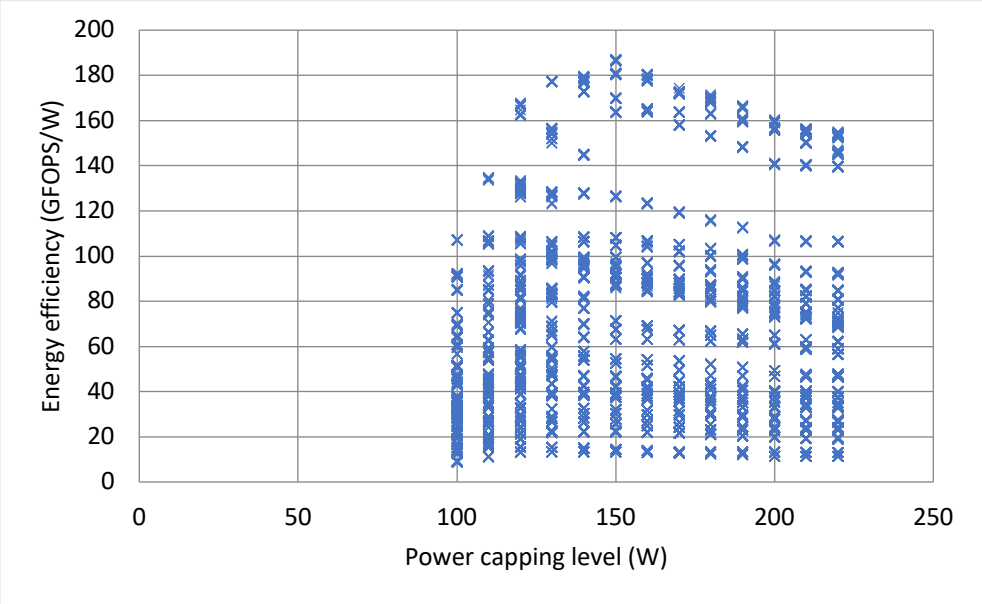

Figure 3.13: Impact of tuning power capping level on energy efficiency using the first strategy for matrix multiplication benchmark on Nvidia GTX980. Values are representative for the energy-efficiency (GFLOPS/W) under a specific power budget for the given configuration

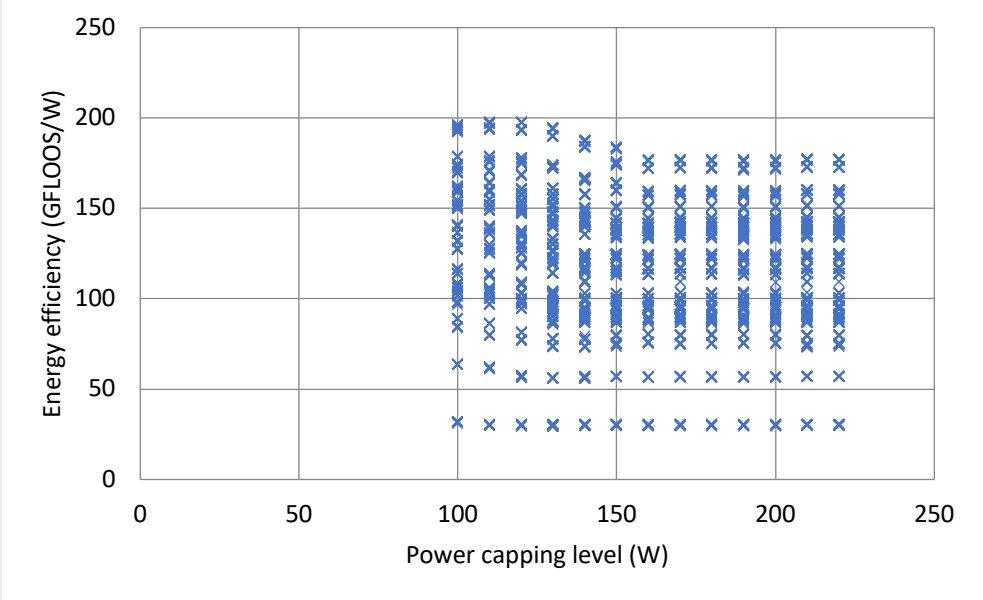

Figure 3.14: Impact of tuning power-capping level on energy efficiency using the first strategy for stencil benchmark on Nvidia GTX980. Values are representative for the energy-efficiency (GFLOPS/W) under a specific power budget for the given configuration

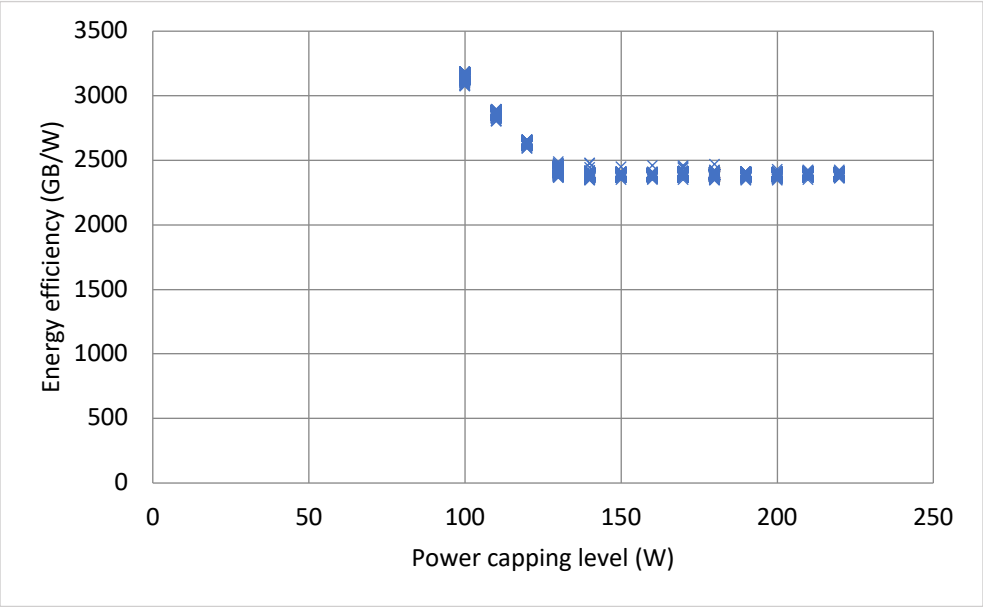

Figure 3.15: Impact of tuning power-capping level on energy efficiency using the first strategy for vector-add benchmark on Nvidia GTX980. Values are representative for the energy-efficiency (GB/W) under a specific power budget for the given configuration

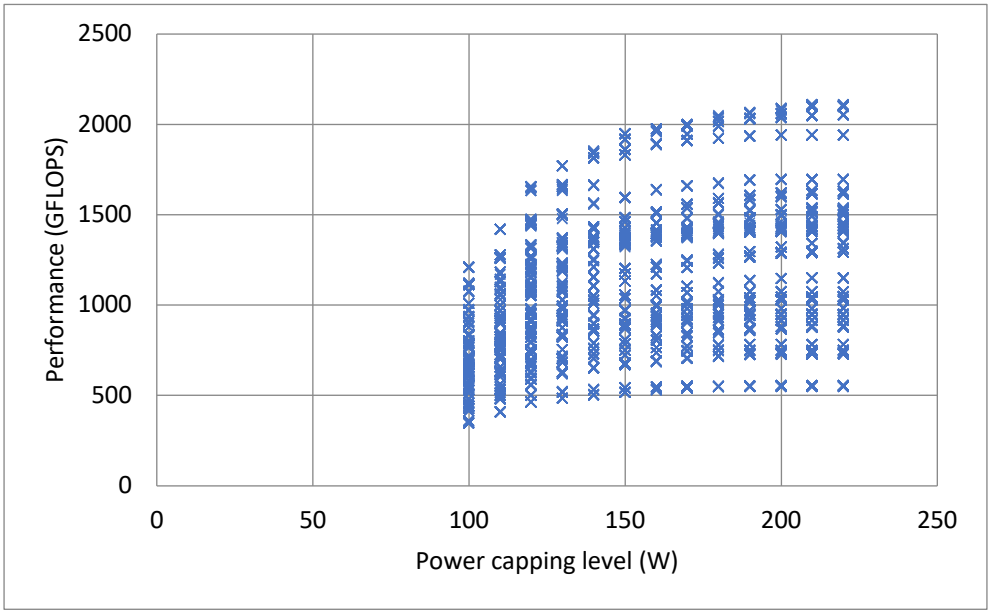

Figure 3.16: Impact of tuning power capping level on performance using the first strategy for matrix multiplication benchmark on Nvidia GTX980. Values are representative for the achievable performance(GFLOPS) under a specific power budget for the given configuration

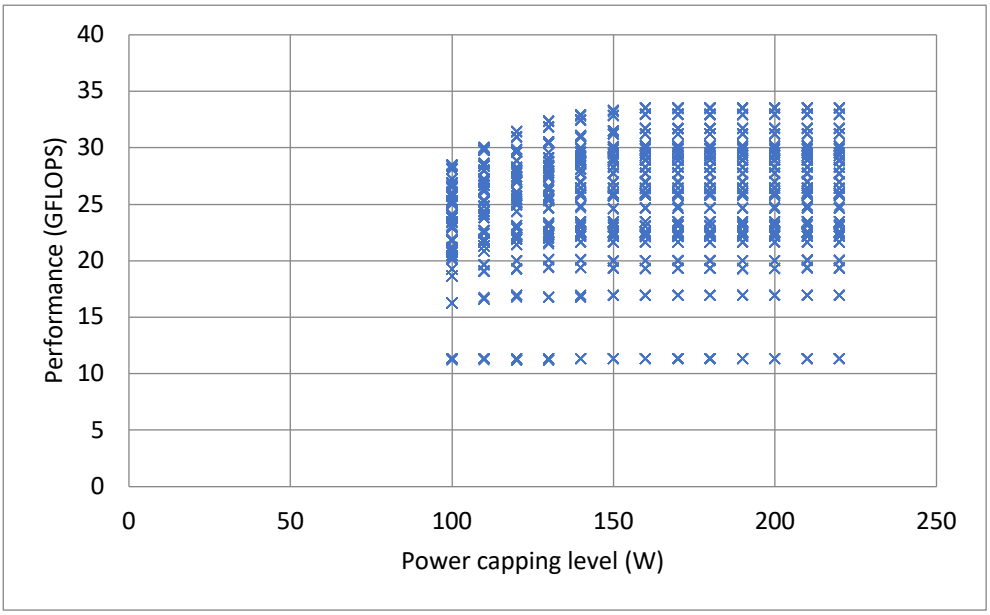

Figure 3.17: Impact of tuning power-capping level on performance using the first strategy for stencil benchmark on Nvidia GTX980. Values are representative for the achievable performance(GFLOPS) under a specific power budget for the given configuration

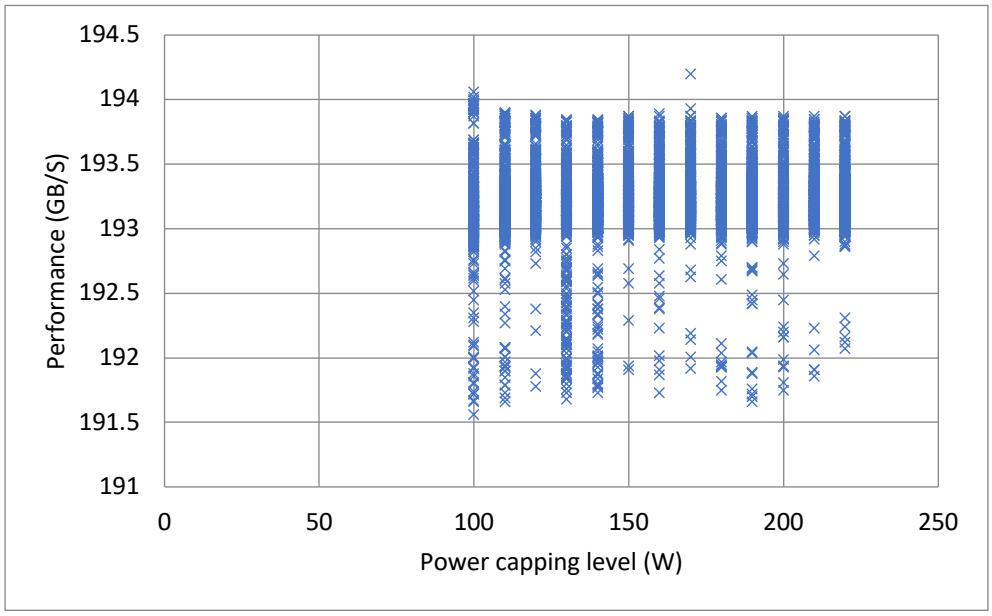

Figure 3.18: Impact of tuning power-capping level on performance using the first strategy for vector-add benchmark on Nvidia GTX980. Values are representative for the achievable performance (GB/S) under a specific power budget for the given configuration

3.3.3.2. Impact of tuning power capping level on memory and core frequencies on Nvidia GTX980

In this section, we briefly present and analyze the impact on memory and core frequencies using the first GPU configuration strategy on Nvidia GTX980. Due to lack of time, we provide experiments and analysis only for this platform.

As it is can be seen in Figure 3.19, the average memory frequency increases when the power budget on Nvidia GTX980 for matrix multiplication benchmark is arises. Moreover, the memory frequency does not reach its peak in contrast with core frequency. For example, the average memory frequency with power capping level of 100 watts is between 450 MHz and 1000 MHz. However, average core frequency spreads over all possible values (see Figure 3.20). This means that, with a specific power budget, the GPU reduces the maximum memory frequency, but it increases the core frequency as much as it can. It seems that improving core frequency have more priority than increasing memory frequency in this GPU. The way this GPU scale memory/core frequency with a specific power budget is dependent to the internal algorithms used. We also observe the same behavior for the stencil and vector-add benchmarks.

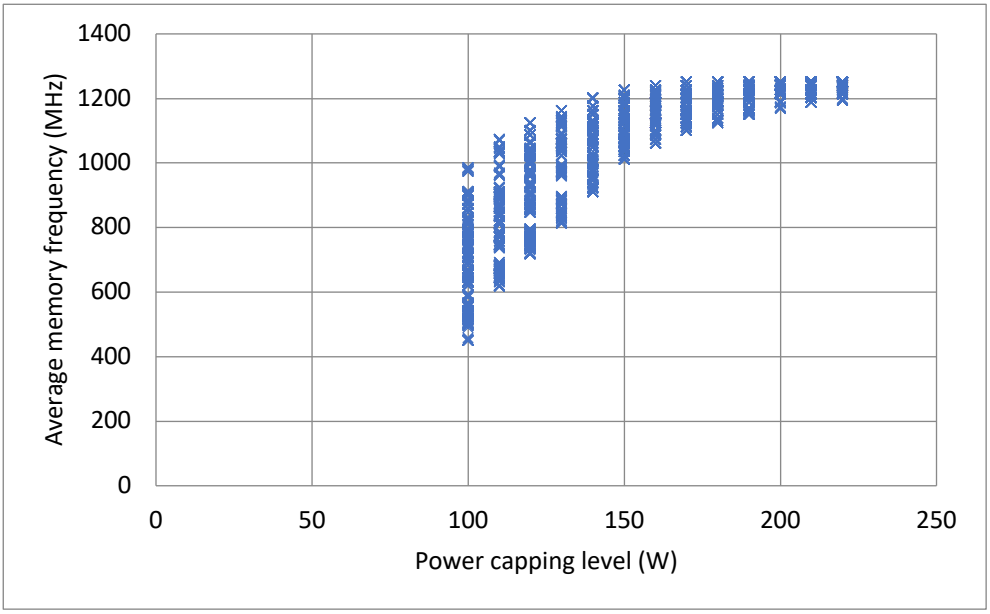

Figure 3.19: Impact of tuning power-capping level on average memory frequency using the first GPU configuration strategy for matrix multiplication benchmark on Nvidia GTX980. Values are representative for the average memory frequency (MHz) under a specific power budget for the given configuration

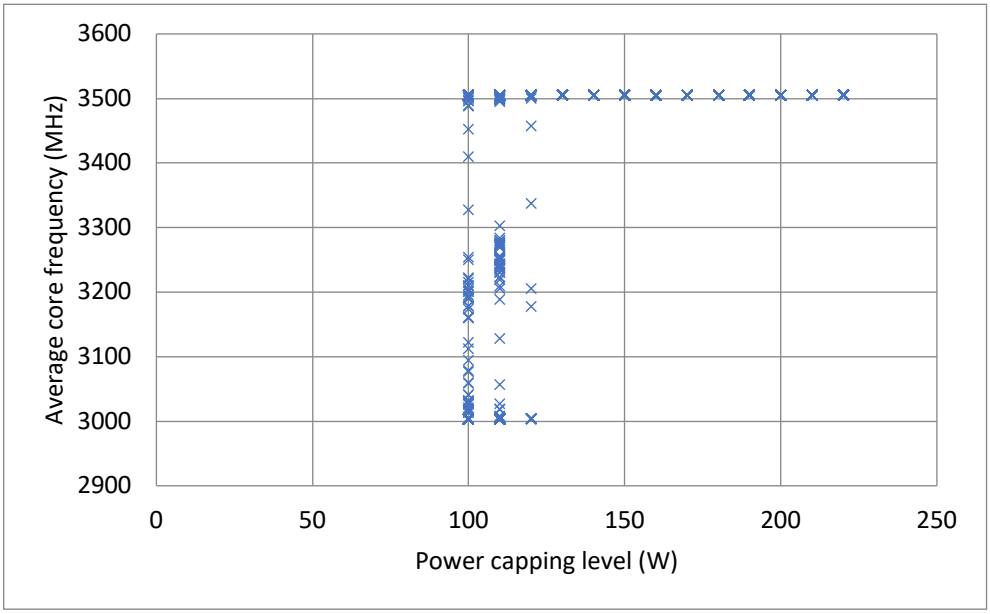

Figure 3.20: Impact of tuning power-capping level on average core frequency using the first GPU configuration strategy for matrix multiplication benchmark on Nvidia GTX980. Values are representative for the average memory frequency (MHz) under a specific power budget for the given configuration

3.3.3.3. Comparison between the first and third strategy

In this section, we are going to compare the impact of tuning power capping level using the first and third strategies. Our goal is to understand how these strategies differ, and if a generic pattern can be identified.

There are similarities in the scatter plots for the power capping level and the energy efficiency using the first and third GPU configuration strategy for all benchmarks. Comparing Figure 3.21 and 3.13, the scatter plots obtained using the first strategy for matrix multiplication code is denser than that achieved by the third strategy. However, they follow the same shape and pattern. The same results can be observed when we compare the 3.22 and 3.14 scatter plots relative to stencil code, and the 3.23 and 3.15 scatter plots relative to vector-add code.

This can assist us to design and implement a faster auto-tuner to find the optimal GPU configuration. The third strategy has a smaller search space than the first one, thereby obtaining results faster. We can use this data to prune the search space of the first strategy. It is because the scatter plots obtained by the first and third strategy follow the same pattern and we need only to search among the configurations that have a close value to the obtained optimal power capping value in the third strategy. We did not depict the plots relative to the other objective functions since they share the similar pattern with the first strategy.

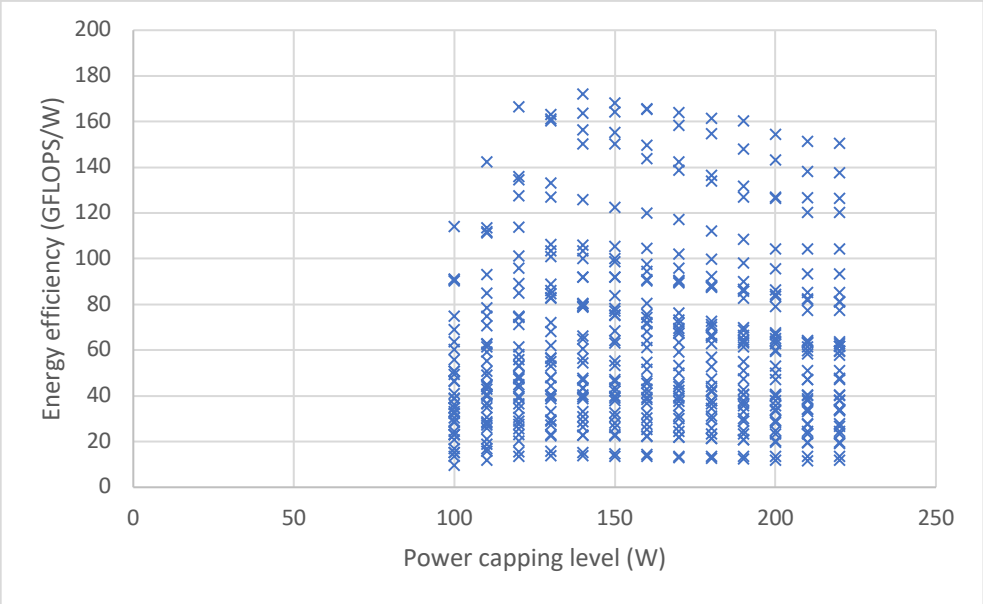

Figure 3.21: Impact of tuning power capping level on energy efficiency using the third strategy for matrix multiplication benchmark on Nvidia GTX980. Values are representative for the energy-efficiency (GFLOPS/W) under a specific power budget for the given configuration

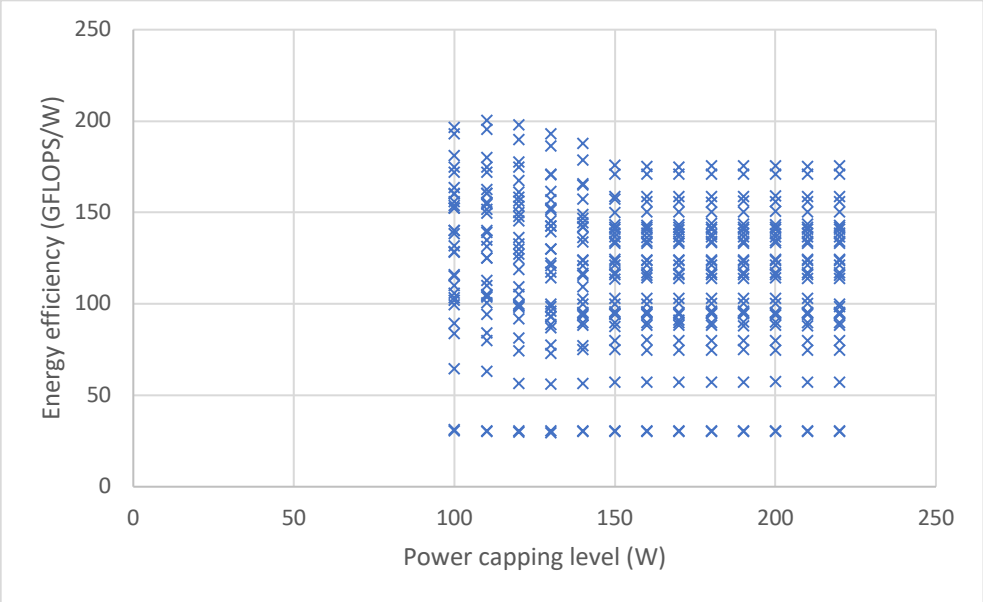

Figure 3.22: Impact of tuning power-capping level on energy efficiency using the third strategy for stencil benchmark on Nvidia GTX980. Values are representative for the energy-efficiency (GFLOPS/W) under a specific power budget for the given configuration

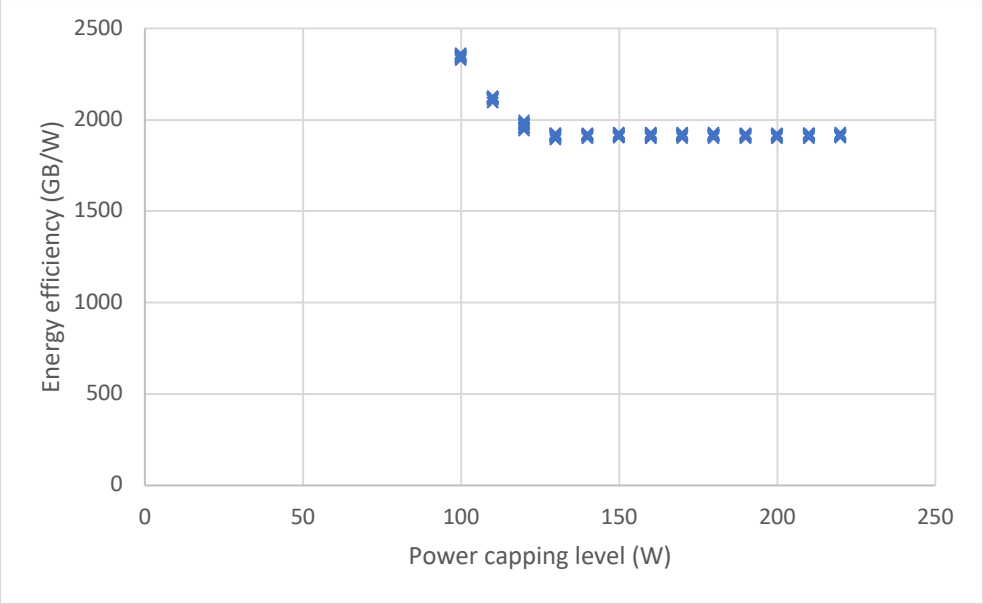

Figure 3.23: Impact of tuning power-capping level on energy efficiency using the third strategy for vector-add benchmark on Nvidia GTX980. Values are representative for the energy-efficiency (GB/W) under a specific power budget for the given configuration

3.3.4. Auto-tuning on Nvidia TitanX (Pascal)

Because of lack of permission to alter the core and memory frequencies on Nvidia TitanX (Pascal), we could not investigate the impact of the first and second GPU configuration strategies on this platform. We also do not present the scatter plots of tuning power capping level for different objective function with the benchmarks using the third strategy since they provide no interesting results to analyse.

The only noticeable result on this platform, obtain by the third strategy, is that we observed a power capping ratio of 49%, 35% and 40% for matrix multiplication, stencil and vector-all programs, respectively. It means that applying power capping on this GPU could influence any given objective function.

3.4. Power capping as a new tunable parameter

Another question that arises in the context of the GPU-level power capping is whether the power capping level can be considered as a tunable parameter. To answer this question, we present different bar charts to show the relationship between the optimal power capping values and all objective functions, for all platforms. Based on the scatter plots in this chapter and Figures 3.1- 3.9, we conclude that the optimal power capping level is a tunable parameter in the context of auto-tuning in GPUs since it depends on the following parameters:

- **Objective function**
  Comparing Figures 3.10 and 3.12, we can observe that the best optimal power capping level to reach to the lowest energy consumption for matrix-multiplication on Nvidia GTX980 is 120 watts. However, the corresponding figure to achieve the highest energy

efficiency is 150 watts. This example shows the dependency between the optimal power capping level and objective function type.

- **Application**
  Clearly, the application characteristics can influence on the optimal power capping level. For example, the best obtained power capping level for matrix multiplication and vector-add programs for energy efficiency objective function on Nvidia GTX980 are 150 and 100 watts, respectively. (See Figure 3.15 and 3.13)
- **GPU**
  We can find different optimal power capping level for different GPU architectures with the same objective function and benchmark. This concept is observable from Figures 3.24 – 3.26.
- **GPU configuration strategy**
  Based on the bar chart in Figure 3.27, we observe that the best power capping value have a relationship with the GPU configuration strategy. For example, the minimum energy consumption for matrix multiplication on Nvidia GTX980 with the first GPU configuration strategy is obtained with a power budget of 150. However, the corresponding figure with the third strategy is 140 watts. The data related to the Nvidia GTX980 shows a small difference between the optimal power capping level of the first and third GPU configuration. However, the difference between the optimal values for the Nvidia GTX TitanX (Maxwell) is larger. It is because the power capping ratio of all the benchmarks on this platform was small, thereby not being effective enough to use power capping on this platform.
- **Problem size**
  The optimal power capping level may get influenced by problem size. We could not investigate this element because of lack of time.

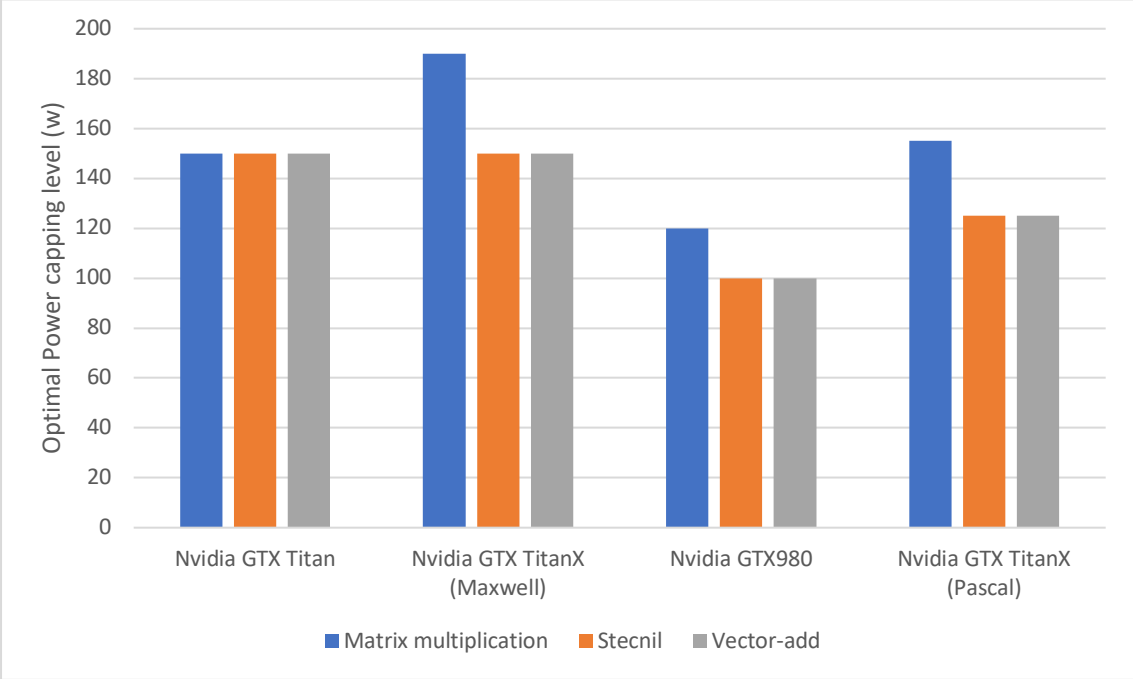

Figure 3.24: Comparison of the tuned power capping levels to obtain the optimal energy consumption for each benchmark on different platforms using the third GPU configuration strategy

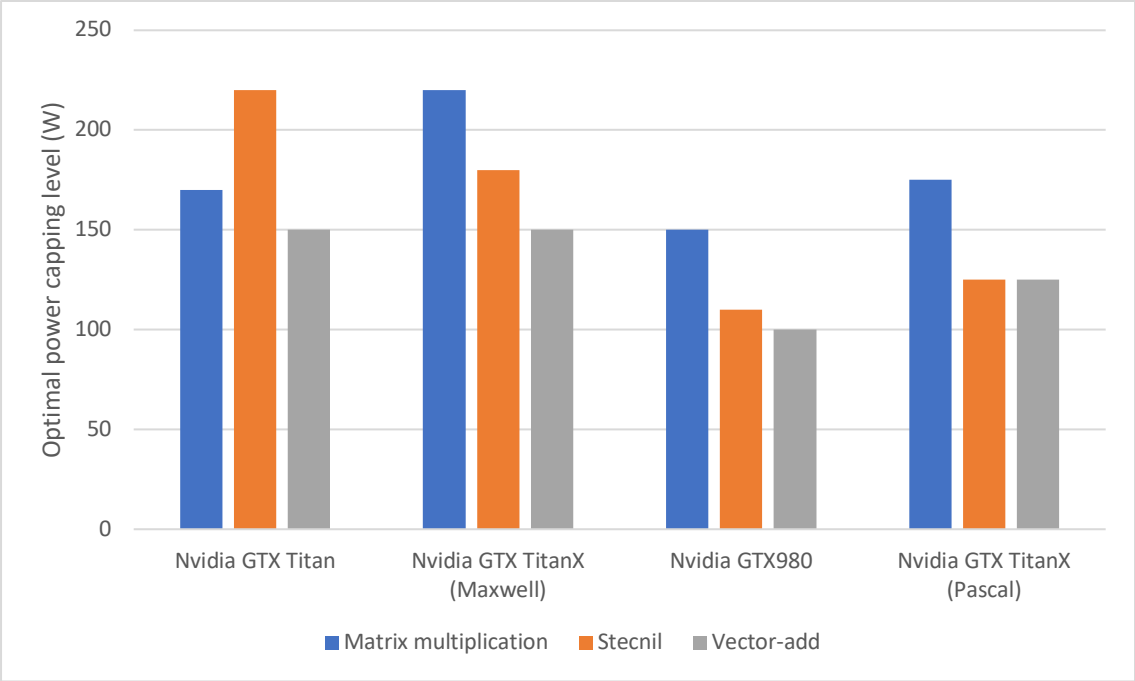

Figure 3.25: Comparison of the tuned power capping levels to obtain the optimal energy efficiency for each benchmark on different platforms using the third GPU configuration strategy

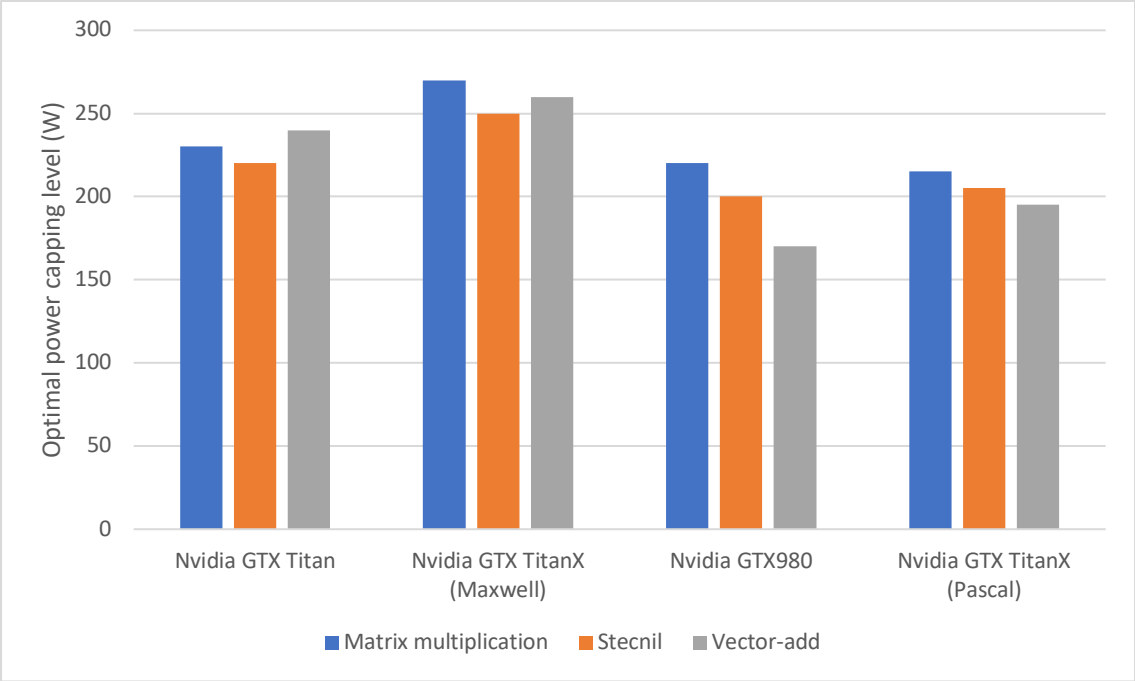

Figure 3.26: Comparison of the tuned power capping levels to obtain the optimal performance for each benchmark on different platforms using the third GPU configuration strategy

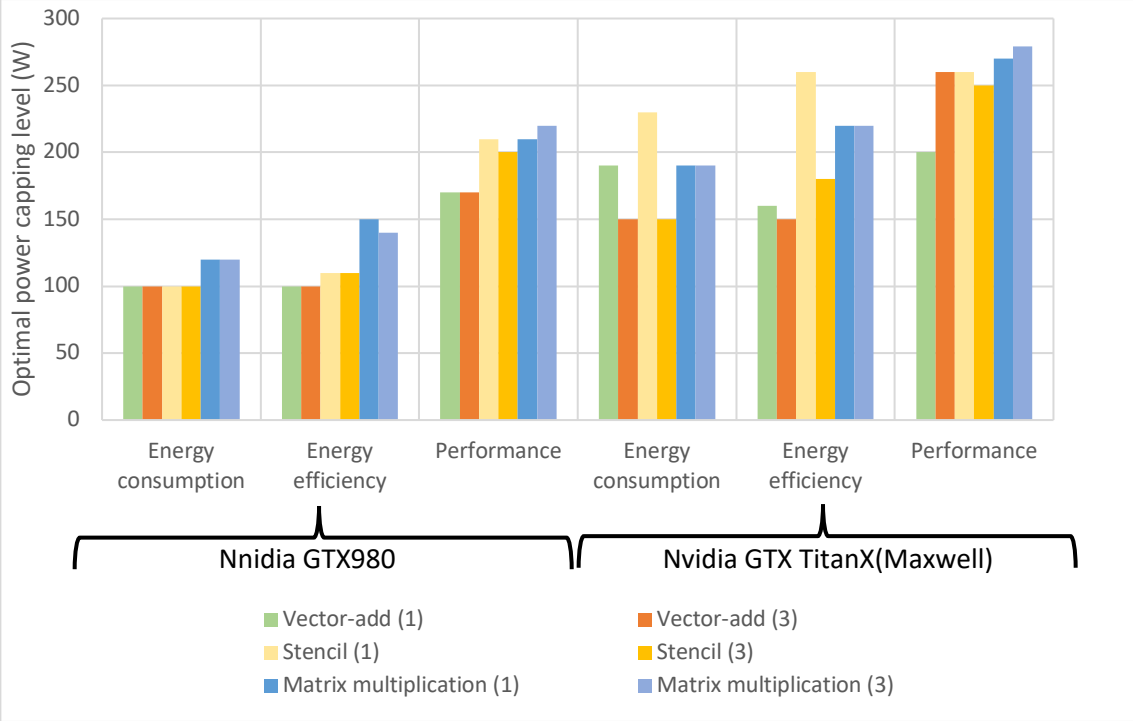

Figure 3.27: Comparison of the tuned power capping level to obtain the best objective function value for each benchmark on two different platforms using the first and third GPU configuration strategy. (1) and (3) indicate that results obtained using the first and third GPU configuration strategy, respectively.

3.5. Selecting the best strategy for the set of GPU parameters

According to our definition from power capping ratio in section 3.4.2, when the power capping ratio of a specific application is larger than zero, there are likely several configurations which average power consumptions are close to the power capping level. Hence, when this power capping ratio increase, the chance of getting better results using the first strategy goes up. This concept is shown in Table 3.8. The power capping ratio is zero for the Nvidia Tesla k20m, which leads to no better results for any given objective function using the first GPU configuration strategy. However, we could improve our objective function values using the first strategy on Nvidia GTX TitanX and Nvidia GTX980. Moreover, comparing the power capping ratio of these two platforms, the obtained values for our objective functions on Nvidia GTX980 are improved remarkably in comparison with those of obtained on the Nvidia GTX TitanX due to higher power capping ratio.

Although expensive, the first GPU configuration strategy can always provide the best results for any given objective function, if we have a high power capping ratio. If we have a small power capping ratio, we can only use the second GPU configuration strategy.

Comparing the second and the third strategy, we could not conclude which one can provide the best results. For example, based on Table 3.4, the lowest energy consumption for stencil code is obtained by the third strategy. However, best performance is obtained by the second strategy for the vector-add benchmark.

	Benchmark								
	Vector-add			Stencil			Matrix Multiplication		
Platform	Nvidia Tesla K20m	Nvidia GTX TitanX	Nvidia GTX980	Nvidia Tesla K20m	Nvidia GTX TitanX	Nvidia GTX980	Nvidia Tesla K20m	Nvidia GTX TitanX	Nvidia GTX980
Search Space Size	673	4914	6287	2977	10891	6481	1611	12776	8363
Power capping size	0	410	2302	0	664	2460	0	3640	7468
Power capping ratio	0%	8.3%	36.6%	0%	6%	37.9%	0%	28.4%	89.2%

Table 3.8: Search space analysis of the benchmarks with first GPU configuration strategy on different GPU platforms

Lack of permission to alter the core and memory frequencies is the main drawback of the first and second GPU configuration methods. Nvidia, by default, does not allow the user to change these frequencies. Therefore, in most cases, it is not possible to use the first and the second strategy, while applying power capping needs less permission.

In time constrained applications, it is crucial to tune the GPU parameters within a time window. So, in this context, the third GPU configuration assists to tune the GPU parameters faster since it has a smaller search space. It can provide a reasonable answer for a given objective function.

3.6. Related Work

This section presents a brief review of the related work on power capping and auto-tuning in GPUs. We firstly present a study about leveraging the power capping in High-Performance Computing (HPC) environments to improve energy efficiency. Then we provide a short-related work on auto-tuning GPU platforms. More literature study has already been conducted and it is linked to the end of this section.

3.6.1. Power capping in HPC environment

Power capping is a technique to enforce a system to work with a predetermined power budget. So far, this method has been studied at different levels, such as rack-level, node-level, CPU-level and memory-level [36] [37]. However, to the best of our knowledge, there is no study at the level of GPU power capping.

Power capping at rack-level can be exploited to increase the number of servers under a power budget in a data center. The total power consumption, in this environment, seldom reaches its theoretical peak value. The infrastructure and the equipped power supply should be able to support this maximum power value. Power capping technique is exploited here to reduce the peak power consumption. This allows the designer to increase the number of racks in a data server with the same power supply. This mechanism can also be used at server-level. That is, by applying a power capping on each server, we can increase the number of servers in each rack.

At component level, power capping has been studied at CPU-level and memory-level [37]. For example, at CPU-level, Intel introduced Intel Running Average Power Limit (RAPL) as a management interface to provide power capping on its products [36][38]. RAPL is a software power model that estimates the energy usage by the processor performance and uses the DVFD to throttle the processor frequency to keep the power consumption below the predetermined threshold.

3.6.2. Auto-tuning

Auto-tuning aims to tune different parameters of a GPU application automatically to improve an objective function value. Since finding the best tunable parameter is also dependent on the program specifications, several researchers specifically studied auto-tuning impact only on a set of applications. For example, in [39], the author studied the effect of auto-tuning on astronomy applications. He showed that auto-tuning is challenging but necessary to enhance performance for astronomical applications. As another illustration, Anzte et al carried out only experiments in auto-tuning dense matrix-matrix multiplication for improving energy efficiency in GPUs [40]. They demonstrated that there is a trade-off between the configuration which provides the best performance and energy-efficiency. In [41], a literature review has already been conducted by us to cover the more related works.

Chapter 4

Conclusions and Future Works

The high energy consumption of GPUs has drawn great attention in academia because of its impact on running costs and temperature in HPC systems. In this thesis, we address the problem of tuning kernel parameters (e.g. block size and grid size) and GPU parameters (e.g. core and memory frequency) to improve either the energy consumption or energy efficiency of GPUs. To do so, we defined three different strategies for tuning the set of GPU parameters, namely (1) memory frequency, core frequency and power capping, (2) memory frequency and core frequency, (3) power capping value. For the sake of simplicity, we called them by the first, second and third strategy, respectively.

We define the *power capping ratio* as the ratio between all configurations of a given program and the number of configurations with a power capping level close to the average power consumption. We observed that tuning the GPU parameters using the first strategy, together with program parameters, can always lead to improved efficiency, when its power capping ratio is high. We also experimentally showed that power capping can be considered as a new tunable parameter for GPUs since its optimal value depends on different parameters such as application, objective function, GPU architecture and GPU configuration strategy.

Comparing the first and third strategies, there are similarities in the relationship between the scatter plots for the power capping level and power-metric values. The plots obtained using the third strategy is always sparser than those of obtained using the first one. Moreover, regarding the impact of tuning power capping level on memory and core frequency, we observed that Nvidia GPUs prioritize increasing core frequency as opposed to memory frequency. Based on different observations, we saw that the Nvidia GTX980 was working with its peak core frequency and different memory frequency under different power budget.

In this thesis, we used a brute-force search method to tune the program and GPU parameters to achieve the optimum, which is time-consuming, and it is not feasible in every case. Future work must investigate how to find the optimal power capping value faster. Another future work would be considering the possible dependency between the problem size and the optimal power capping value.

Moreover, as we explained, power capping ratio have a significant role in this research. However, calculating power capping ratio is expensive, which leads to propose a faster method as a future work. Another future development would be to test more GPU architectures and add new benchmarks.

References

[1] Top500.org. (2018). **TOP500 Supercomputer Site**. [online] Available at: https://www.top500.org/ [Accessed 2 Aug. 2018].
[2] NVIDIA. (2018). **NVIDIA Tesla V100 Data Center GPU**. [online] Available at: https://www.NVIDIA.com/en-us/data-center/tesla-v100/ [Accessed 6 Aug. 2018].

[3] van Werkhoven, B. (2019). Kernel Tuner: **A search-optimizing GPU code auto-tuner**. Future Generation Computer Systems, 90, 347-358.
[4] Hall, P. J. (2011, August). **Power considerations for the Square Kilometre Array (SKA) radio telescope**. In 2011 XXXth URSI General Assembly and Scientific Symposium (pp. 1-4). IEEE.
[5] Barbosa, D., Márquez, G. L., Ruiz, V., Silva, M., Verdes-Montenegro, L., Santander-Vela, J., ... & Kramer, M. (2012). **Power Challenges of Large Scale Research Infrastructures: the Square Kilometer Array and Solar Energy Integration; Towards a zero-carbon footprint next generation telescope.** arXiv preprint arXiv:1210.4972
[6] [3] Price, D., Clark, M., Barsdell, B., Babich, R. and Greenhill, L. (2015). **Optimizing performance-per-watt on GPUs in high performance computing. Computer Science - Research and Development**, 31(4), pp.185-193.
[7] Meijer, M., & de Gyvez, J. P. (2008). **Technological boundaries of voltage and frequency scaling for power performance tuning**. In Adaptive Techniques for Dynamic Processor Optimization (pp. 25-47). Springer, Boston, MA.
[8] Choi, K., Lee, W., Soma, R., & Pedram, M. (2004, November). **Dynamic voltage and frequency scaling under a precise energy model considering variable and fixed components of the system power dissipation**. In Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design (pp. 29-34). IEEE Computer Society.
[9] Mei, X., & Chu, X. (2017). **Dissecting GPU memory hierarchy through microbenchmarking.** IEEE Transactions on Parallel and Distributed Systems, 28(1), 72-86.
[10] Docs.nvidia.com. (2019**). Programming Guide: CUDA Toolkit Documentation**. [online] Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability [Accessed 8 Feb. 2019].
[11] Larkin, J. (2016). [online] Icl.utk.edu. Available at: http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/GPU_Fundamentals.pdf [Accessed 11 Feb. 2019].
[12] Huang, M., Men, L., & Lai, C. (2013). **Accelerating mean shift segmentation algorithm on hybrid CPU/GPU platforms**. In Modern Accelerator Technologies for Geographic Information Science (pp. 157-166). Springer, Boston, MA.
[13] Nugteren, C. (2014). **Improving the programmability of GPU architectures (Doctoral dissertation**, Ph. D. thesis, Department of Electrical Engineering, Eindhoven University of Technology).
[14] Li, A. (2016). **GPU performance modeling and optimization**. Diss. Technische Universiteit Eindhoven.
[15] Glaskowsky, P. N. (2009). **NVIDIA's Fermi: the first complete GPU computing architecture**. White paper, 18.
[16] NVidia, F. (2009). **Nvidia's next generation CUDA compute architecture**. NVidia, Santa Clara, Calif, USA.

[17] NVIDIA Corporation. (2012). **NVIDIA's Next Generation CUDA Compute Architecture:** qua GK110/210.
[18] Jeffers, J., & Reinders, J. (2015). **High performance parallelism pearls volume two: multicore and many-core programming approaches**. Morgan Kaufmann.
[19] Hennessy, J. L., & Patterson, D. A. (2011). Computer architecture: a quantitative approach. Elsevier.
[20] Nvidia, (2014), "**Whitepaper: NVIDIA GeForce GTX 750 Ti**", [Online]. Available: http://international.download.nvidia.com/geforcecom/international/pdfs/GeForce-GTX-750-TiWhitepaper.pdf, [Accessed 29 Feb 2019].
[21] NVIDIA, T. (2016). **P100 white paper**. NVIDIA Corporation.
[22] NVIDIA, T. (2017). **V100 GPU architecture**.
[23] Nvidia, T. (2018). **Nvidia Turing GPU architecture**.
[24] [24] Dayarathna, M., Wen, Y. and Fan, R. (2016). **Data Center Energy Consumption Modeling: A Survey.** IEEE Communications Surveys & Tutorials, 18(1), pp.732-794.
[25] Bergstra, J., Pinto, N., & Cox, D. (2012, May). **Machine learning for predictive auto-tuning with boosted regression trees.** In 2012 Innovative Parallel Computing (InPar) (pp. 1-9). IEEE.
[26] Dobslaw, F. (2010). **Recent development in automatic parameter tuning for metaheuristics**. In Proceedings of the 19th Annual Conference of Doctoral Students-WDS 2010.
[27] Intel, E. (2004). **SpeedStep® technology for the Intel® Pentium® M processor**. White Paper, Intel.
[28] Staff, A. M. D. (2002). **AMD PowerNow! Technology Brief**. Advanced Micro Devices, Inc.
[29] AMD, F. G. **AMD PowerTune Technology, May 2011**. PowerTune Technology Whitepaper, 1-4.
[30] Steigerwald, B., & Agrawal, A. (2011). **Developing green software. Intel White Paper**, 9.
[31] Min, R., Bhardwaj, M., Cho, S. H., Shih, E., Sinha, A., Wang, A., & Chandrakasan, A. (2001). **Low-power wireless sensor networks**. In VLSI Design, 2001. Fourteenth International Conference on (pp. 205-210). IEEE.
[32] Bal, H., Epema, D., de Laat, C., van Nieuwpoort, R., Romein, J., Seinstra, F., ... & Wijshoff, H. (2016). **A medium-scale distributed system for computer science research: Infrastructure for the long term**. Computer, (5), 54-63.
[33] NVIDIA Developer. (2018). **NVIDIA System Management Interface**. [online] Available at: https://developer.nvidia.com/nvidia-system-management-interface [Accessed 24 Dec. 2018].
[34] van Werkhoven, B. (2019). Kernel Tuner: **A search-optimizing GPU code auto-tuner.** Future Generation Computer Systems, 90, 347-358.
[35] Borghesi, A., Collina, F., Lombardi, M., Milano, M., & Benini, L. (2015, August). **Power capping in high performance computing systems**. In International Conference on Principles and Practice of Constraint Programming (pp. 524-540). Springer, Cham.
[36] Petoumenos, P., Mukhanov, L., Wang, Z., Leather, H., & Nikolopoulos, D. S. (2015, December). **Power capping: What works, what does not**. In 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)(pp. 525-534). IEEE.
[37] Komoda, T., Hayashi, S., Nakada, T., Miwa, S., & Nakamura, H. (2013, October). **Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping**. In 2013 IEEE 31st International Conference on Computer Design (ICCD) (pp. 349-356). IEEE

[38] Dutta, B., Adhinarayanan, V., & Feng, W. C. (2018, May). **GPU power prediction via ensemble machine learning for DVFS space exploration**. In Proceedings of the 15th ACM International Conference on Computing Frontiers (pp. 240-243). ACM.
[39] Sclocco, A. (2017). **Accelerating Radio Astronomy with Auto-Tuning**.
[40] Anzt, H., Haugen, B., Kurzak, J., Luszczek, P., & Dongarra, J. (2015). **Experiences in autotuning matrix multiplication for energy minimization on GPUs. Concurrency and Computation**: Practice and Experience, 27(17), 5096-5113.
[41] Sahrifi Esfahani, E (2019). "**A Survey on Solutions for Improving Energy-efficiency in GPUs".** [Online] Available: https://drive.google.com/file/d/1vZSZKDRTEY-tJqTp4A73XCTom01SeiKr/view?usp=sharing
[42] Fey, Dietmar et al. (2010) Grid-Computing: **Eine Basistechnologie für Computational Science**. Page 439. Publisher: Springer.
[43] Schäfer, A., & Fey, D. (2011). **High performance stencil code algorithms for GPGPUs**. Procedia Computer Science, 4, 2027-2036.