Universiteit Leiden
The Netherlands

BSc Computer Science

Optimizing the energy efficiency of the Nvidia Jetson

Martijn Frericks

Supervisor:
Ben van Werkhoven

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

23/07/2024

Abstract

This thesis extends the capabilities of Kernel Tuner by introducing a new observer for Nvidia Jetson devices. The extension focuses on monitoring the clock frequency, temperature and power consumption to try and optimize the energy efficiency of Nvidia Jetson devices. For testing the new observer, a Nvidia Jetson Xavier NX was used. We first tested if the temperature of the device influenced the performance and/or the energy efficiency. We discovered that the device was able to keep the temperature between 45 and 48 degrees Celsius during 500 executions of a matrix multiplication kernel. Minimal change in performance and energy efficiency were observed under these conditions. Further experiments aimed at benchmarking how well this extension was able to optimize the energy efficiency by tuning the thread block size and clock frequency for five different kernels. The results showed varied preferences for the most energy efficient configuration among kernels, which highlights the importance of tuning each kernel separately. Additionally, four tuning strategies were evaluated: GlobalEnergy, BlockToFreq, FreqToBlock and RaceToIdle. The GlobalEnergy strategy, which considers all possible configurations for energy optimization, proved to be the most energy efficient. The RaceToIdle strategy, focused on minimizing execution time, consumed the most energy. BlockToFreq and FreqToBlock, which first optimize one parameter before the other, successfully reduced the search space but resulted in a maximum of 4.55% higher energy consumption compared to GlobalEnergy.

Contents

1 Introduction

Computers play a big role in almost every aspect of daily life. The increasing role of computers is largely due to technical advancements, especially in the domain of computations, enabling users to perform more complex tasks. GPUs (Graphical Processing Units) play an important role in performing these complex tasks [BIM16]. GPUs are mainly used in HPC (High Performance Computing) and for example in research in fields such as artificial intelligence. GPUs consist of several multiprocessors which enable a GPU to perform multiple calculations simultaneously at significant speeds, so it can process data in parallel to speed up computations [OHL+08] [GPKB12]. Due to its exceptional computational speeds and performance, it is widely used in edge computing [CLMS20] and scientific research, such as deep learning [PFG+22]. Edge computing is the concept of capturing and processing data as close to its source or end user as possible. To do so, the processing and data analysis is done on the device itself and the data is not sent to a central computer to do the computing. By doing so, the latency that data transfer causes is removed. This enables them to be used in a wide variety of applications, such as Internet of Things and robotics. Internet of Things connects devices to the internet, enabling them to exchange information. GPUs are crucial for real-time processing of data and for making decisions based on the data [FCX+17].

1.1 Problem

There is however a problem with GPUs. High-end GPUs can use up to 365 Watt (GeForce GTX 590 2013) and 294 Watt (Radeon 2013), while high-end CPUs only use up to 45 Watt (Core i7-3770T) and 150 Watt (Xeon E7-8870) [MV14]. These GPUs and CPUs are used in desktop, however there are so-called edge GPUs as well. Edge GPUs are used in edge computing and consume less power than desktop GPUs. Using a lot of power can have severe consequences, especially for devices that are not connected to the power grid, but rely on batteries that have only limited power. Due to this, use cases of GPUs are limited while they do not have an infinite supply of power. For example, at the moment of writing autonomous vehicles are becoming more and more advanced. They use sensors, which process all the information collected. To process this, GPUs are used. But such vehicles are not connected to the power grid and so only have a limited power they can use. If the GPU uses a big portion of the power available, the radius in which a vehicle has to charge is shortened, which is disadvantageous [LLCP20]. This represents only one of the multiple fields where GPUs can be used, but causes limitations due to its power consumption.

1.2 Solution

This thesis will investigate optimizing the energy efficiency of GPUs for edge computing. To achieve this, Kernel Tuner will be used to auto-tune applications on the Nvidia Jetson GPU. Kernel Tuner is software which can be used to auto-tune kernel code, allowing users to tune a kernel for a specific purpose, such as improving energy efficiency or performance. A major factor in how well a GPU uses its energy is how well the GPU is optimized for the task the GPU needs to perform. Factors such as clock frequency, thread block size or temperature can affect the total power consumption. However, it is a complex task to find the optimal configuration for a specific GPU. For this problem a solution was created: auto-tuning [BDG+18]. Auto-tuning is the practice of fine-tuning a application, for

example on a GPU, to maximize its throughput, optimize its energy efficiency, or achieve optimal performance tailored to specific objectives. Kernel Tuner and Auto-tuning will be further explained in Section 2. Kernel Tuner does not yet have an implementation to auto-tune GPUs on Nvidia Jetson devices which means that applications on these devices cannot yet be optimized to be energy efficient using Kernel Tuner. This thesis contributes an implementation to the Kernel Tuner software to auto-tune applications on the Nvidia Jetson GPUs. This implementation will be looking at the clock frequency, energy, thread block size and temperature of the GPU to try to improve the energy efficiency. The implementation consists of an observer [SVVWB22], which will be called TegraObserver. An Observer in Kernel Tuner is a so-called hook into the software, which can measure parameters, such as power consumption, during execution.

1.3 Research Objective

To make users of embedded GPUs able to use the new TegraObserver, this thesis will focus on four objectives. These four objectives are created to make sure the software is reliable and makes a difference to preserve power. The four objectives are:

- Write an observer, called TegraObserver, to monitor the GPU clock frequency, temperature and power consumption and set the clock frequency on Nvidia Jetson devices.
- Write tests in Kernel Tuner to test the individual implementations to verify the program works as intended.
- Benchmark these implementations with kernels to test if the implementations work in real-world use cases.
- Discuss the results of the benchmarks to see where and why power is being conserved.

1.4 Contribution

This thesis will contribute a new observer for Kernel Tuner designed specifically for Nvidia Jetson devices. This observer will enhance the energy efficiency of these devices by enabling the monitoring of the temperature, power consumption and clock frequency. Additionally, the user can set a specific clock frequency before benchmarking in order to tune the clock frequencies as well. By implementing this observer into Kernel Tuner, this contribution will enable Kernel Tuner to optimize Nvidia Jetson devices to increase their energy efficiency. This contribution will be particularly significant for users who use the Nvidia Jetson in remote environments or in applications which are not directly connected to the power grid. By empowering these users to auto-tune the GPU to be more energy efficient, they will be able to extend the operational lifespan of these devices. Additionally, making the Nvidia Jetson devices more energy efficient will reduce both cost and environmental damage. The contribution to Kernel Tuner can be found on the Kernel Tuner GitHub repository [KTg].

1.5 Thesis overview

This thesis is organized into multiple sections. Section 2 briefly describes the parameters we intend to configure to tune the NVIDIA Jetson for energy efficiency and reviews related work to

provide an overview of existing research on GPU energy efficiency. In Section 3, we explain how the implementations of the TegraObserver work and the reasons behind the choices we made for these implementations. Section 4 will describe the experiments we conducted to benchmark the TegraObserver and the experimental setup used for these experiments. Section 5 will compare and discuss the results of all the benchmarks. Lastly, Section 6 will summarize the findings in this thesis.

2 Background

This section will first describe the three parameters which will be used to optimize applications on the Nvidia Jetson GPUs. Secondly, this section will look briefly into related works about the energy efficiency of GPUs, Kernel Tuner and auto-tuning.

2.1 Parameters

This thesis focuses on three parameters to try and find a configuration that is energy efficient. These parameters are temperature, clock frequency and thread block size. This subsection gives an overview of the three parameters and explain why these specific parameters are important to make a GPU more energy efficient.

2.1.1 Clock frequency

An important factor for the performance of a GPU is the clock frequency. The unit of measurement of the clock frequency is hertz (Hz) or megahertz (MHz). The clock frequency reflects the number of cycles per second at which the GPU core operates. There is however a limit to this frequency. A higher clock frequency means that the transistors switch more frequently, which will produce an enormous amount of heat [Gee05]. The dissipation of heat must be kept under control and so, the clock frequency cannot rise above certain limits. The clock frequency has a lot of influence on the energy consumption of the GPU. Higher clock frequencies can increase the performance of the GPU, and by doing so increase the total power consumption as well. On the other hand, lower clock frequencies may result in a longer time to calculate and so, use more power while the GPU takes longer to perform the calculations. So, it is important to find a middle ground where the power consumption for a configuration is minimal.

2.1.2 Temperature

Another important factor to keep in mind while trying to make an applications on the GPU more energy efficient is the temperature of the GPU while it is operating. When the GPU is performing calculations and operations, the heat that is produced increases. If the fans of the GPU can't keep up with the heat production, the GPU will become too hot which could potentially damage the component of the computer. To make sure the GPU will does not overheat, a fail-safe was created: thermal throttling [BCVMFB20]. When the temperature of the GPU exceeds certain limits, thermal throttling will automatically lower the clock frequency and power consumption to mitigate any potentially dangerous and destructive effects. By doing so, any configurations which were previously energy efficient will be overwritten to become less energy efficient. To make sure

this will not happen, it is imperative to keep measuring the temperature of the GPU to make sure it will not reach the temperature which activates thermal throttling.

2.1.3 Thread block size

The last parameter this thesis focuses on is the thread block size the GPU utilizes. The thread block size is an important factor when we want to make applications on the GPU more energy efficient. The thread block size plays multiple roles. First of all, the thread block size determines how many threads will be scheduled per block. Having multiple threads in a block can be advantageous, while threads inside a block can communicate faster due to shared memory. Secondly, GPU contain numerous streaming multiprocessors (SM), and each SM has its own shared memory and registers. Block are scheduled on these SMs. Too many blocks per SM can be disadvantageous, while it may exceed hardware limitations and lead to inefficient resource utilization and can result in contention for memory and registers. By under utilizing the GPU, it will not be able to optimally parallelize thread execution. This causes a decrease in performance. If the thread block size is too small, the GPU might become underutilized and so, use too much energy when it could have used more of its resources to save energy. So, it is imperative to find a middle ground in which the GPU uses all of its resources optimally, and so, does not use more energy than necessary [ISAHA22].

2.2 Related work

This section reviews research and techniques aimed at making GPUs more energy efficient. Thereafter, we explain how energy is measured and consumed by the GPU. Following this, three real-world examples of Nvidia Jetson use cases are presented, with an explanation of why an Nvidia Jetson was specifically chosen. Finally, the code that was written before starting this thesis is discussed.

2.2.1 Historical research

In 1965, one of the most famous assertions about the growth of computational power was written by Gordon Moore. He predicted that the number of transistors on a microchip would double approximately every two years, which would lead to an exponential growth of computational power [Sch97]. This prediction is known as Moore's law. 9 years later, in 1974, another significant principle related to transistor scaling was written by Robert H. Dennard. Dennard's scaling states that the power consumption of a transistor remains constant while reducing the physical dimensions. This allows for doubling the number of transistors and keeping the power consumption constant [JN16]. Complementing these principles is Koomey's law, written years later in 2010, which states that the energy efficiency of computers doubles roughly every 18 months [BPD21]. This highlights the shift in focus from performance to improving the energy efficiency.

A milestone in making GPUs more energy efficient was the NVIDIA Kepler Architecture, created in 2012. This was one of the first GPUs that implemented techniques to make the GPU more energy efficient [AYM22]. This architecture used techniques such as clock gating to achieve this goal. Clock gating selectively stopped clock signals to parts where they were not used, which reduced the power consumption [WPW00].

Since then, numerous advancements have been made to make GPUs more energy efficient. Some relevant techniques are:

- DVFS is a technique used for reducing the energy consumption of processors by varying the voltage and frequency at runtime [MK15]. The main idea is that the activity levels of a GPU can vary and have idle periods. Reducing the voltage and runtime when the application has a low activity, energy can be saved, while still meeting the performance requirement of the running workload [MK15, GVM+13]. However, while this method might be able to lower the power consumption for dynamic power, it cannot change the power consumption in leakage power [ZLT+19, MV14].
- CPU-GPU workload division-based techniques: this is also known as heterogeneous computing. While the GPU is better capable of handling parallelizable tasks, the CPU is better in complex decision tasks. Energy can be saved while the CPU uses less power [S+09, MV14].
- Power gating completely shuts off entire blocks which are not in use. By doing so, this technique tries to limit the amount of power leakage [JMSN05].
- Power capping is a technique which sets application-specific power limitations. This approach ensures that the GPU operates within defined power consumption parameters, to prevent it from using too much power, and by doing so, make it more energy efficient [SVVWB22].

2.2.2 Power consumption

Power consumption is a critical aspect of a GPU when thinking about its performance and environmental influence. As said in the introduction, GPUs are becoming more powerful and so, are demanding more energy. To measure the power consumption of a GPU, and so determine the energy efficiency of the GPU, GFLOPS/Watt is a widely used metric. It measures how many floating-point operations can be executed per second, per watt. The reason why we want to make applications on the GPU more energy efficient is clear: to lower the environmental impact, make it more suitable for edge devices that do not have infinite power at their disposal, cost reduction for companies and many more. If we take the previously explained parameters, we will hopefully get a better understanding of how we can achieve this. The total power which is consumed by a GPU can be calculated by combining the dynamic power and leakage power [MV14].

- Dynamic power: This is the power that is used by the GPU when it is active and performing operations. To calculate the total number of power that the GPU uses when switched on, a formula has been made. $P_{dynamic} = \alpha * C * V^2 * f$. In this formula, α is the activity factor, C is the capacitance parameter, V is the voltage that is supplied and f is the operating frequency [HGS19]. This power consumption is determined at runtime.
- Static power or leakage power: This is the power that is consumed due to leakage when the transistors of the GPU are off. To measure this power, the formula $P_{static} = V_{cc} * N * K_{design} * I_{leak}$ is used. In this formula, V_{cc} is the supply voltage, N is the number of transistors in the design, K_{design} is a constant factor that represents the technology characteristics and I_{leak} is a normalized leakage current for a single transistor [HK10].

2.2.3 Power monitoring

There are multiple ways to measure the total amount of power consumption for a component, in this case the GPU. The first two ways to measure power are called direct measurements. Direct

measurements directly sample the power from either internal or external sensors. Both periodically measure the total energy as the integral of the power over the execution time [FYK+17].

- External power meters include inline universal meters for measuring AC (alternative current) power and DC (direct current) meters which are connected between the power supply and the component [BIM16]. While they may be able to read the total power which is consumed, they cannot separate the power consumption for individual components [BIM16, SVVWB22].
- Internal sensors can solve this problem by measuring the power consumption directly on the component itself [JOL+23]. They do not need any extra hardware and can be directly accessible. There are some drawbacks to using internal sensors, however. [BZZ14] describes that they concluded that the power consumption lagged behind kernel activity and that the shape of the energy profile did not match the kernel activity, concluding that it is not a perfect tool to measure the power consumption and so, can be inconsistent.

The other way to measure power consumption is called indirect measurements. Here we estimate the power consumption using models. A common model that is used to estimate the power consumption is the linear regression model. Zhang et al. [CLZ+11] used a linear regression tree and sophisticated random forest methods to correlate the power consumption with a set of independent performance variables.

The Nvidia Jetson Xavier NX, used later for the experiments in Section 4, is equipped with an internal sensor to monitor the temperature and energy. However, the internal sensor measures the combined power consumption of the GPU and CPU, rather than the power consumption of the GPU alone. This can lead to some inaccurate results, because the CPU uses power during the measurements as well. Süzen et al. [SDŞ20] conducted experiments using the Nvidia Jetson TX2 and Nvidia Jetson Nano. These experiments involved a CNN algorithm designed to classify 13 different fashion products using a dataset of 45K images to evaluate the power consumption of hardware components, including the CPU and GPU. Their findings indicated that, while the GPU consumes the most power, the CPU used on average 40% of the power consumed by the GPU. These findings indicate that using the onboard sensor, which measured both GPU and CPU power consumption, may introduce some inaccuracies.

2.2.4 Kernel Tuner and auto-tuning

Tuning applications on the GPU is the concept of adjusting and optimizing specific parameters to increase the performance of a specific task. However, the number of possible configurations can be large and different kernels may differ in which configurations are optimal. To find the optimal configuration for each of these cases auto-tuning is used. Auto-tuning automates the exploration of all the different parameters that the user wishes to optimize to achieve better performance [SvWB22, GGXS+12].

To auto-tune kernels, tools such as Kernel Tuner were developed to perform this task. In Kernel Tuner, the user can input kernel code and specify which parameters should be used to auto-tune the given code. By default, Kernel Tuner uses brute force to auto-tune kernel code, which tests all possible combinations of parameters but may take a long time. Alternatively, the user can manually specify a different strategy for tuning the kernel code, such as Basin Hopping or Bayesian

Optimization. Kernel Tuner can then measure and benchmark the different configurations using all the entered information. Based on the statistics of these measurements, the best configuration of parameters is given. To measure the performance at run time, Kernel Tuner uses so-called Observers. Observers make it possible to quantify measurements other than execution time to make it easy for the user to control what exactly is being measured by Kernel Tuner [Obs]. These Observers are so-called programmable hooks in the Kernel Tuner software, as seen in Figure 1, which allow users to change or expand Kernel Tuner's benchmarking behavior at any of the lower levels.

Kernel Tuner already has an observer which works on Nvidia GPUs, using NVML (Nvidia Management Library) [SVVWB22]. NVML allows the user to monitor metrics such as temperature and power consumption. However, NVML does not work on the Nvidia Jetson devices, while Nvidia Jetson device utilize a differ architecture, known as the Tegra architecture. This is why this thesis contributes a new observer, the TegraObserver, to Kernel Tuner to be able to read from or write to the Nvidia Jetson devices.

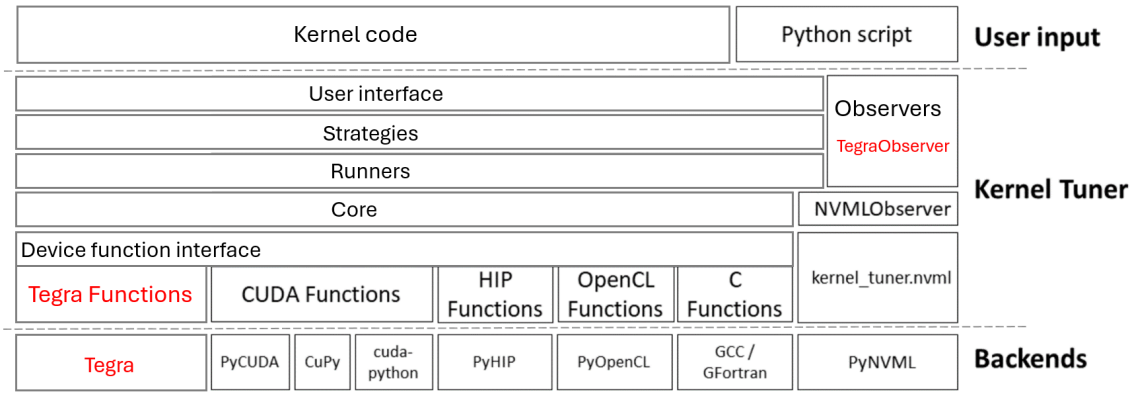

Figure 1: The current architecture of Kernel Tuner. In red are the extension this thesis contributes to Kernel Tuner, specifically the TegraObserver and the Tegra platform.

2.2.5 Nvidia Jetson applications

The Nvidia Jetson is a device that uses low power and has high throughput, which makes it a promising platform for a variety of tasks. Mittal et al. [Mit19] highlights the significance of the NVIDIA Jetson CPU-GPU heterogeneous architecture, small form factor and low power consumption for various applications, including robotics, autonomous driving and drone navigation using deep learning. To give an idea of the utilization of the Nvidia Jetson, here are three examples of real-world use cases.

- Sanchez et al. [SLS+20] uses the Nvidia Jetson Nano as a core component of an edge computing system. The Jetson Nano is used for gathering data from wearable devices which monitor the health of workers. The reason that the Jetson was chosen for this role in the proposed system, is because it can monitor and analyze data in real-time. This will improve and ensure the health of the user.

- Another real-world use case of the Nvidia Jetson is in IoT. Uddin et al. [UAR+21] describes the application of the Nvidia Jetson as a platform for image processing and deep learning

for traffic cameras. In the proposed system, four cameras will be installed on an intersection of four roads which monitor the traffic in real-time and can capture the number plates of vehicles. The Nvidia Jetson will receive the data from the cameras. If any vehicle ignores a red light, the object detection algorithm of the Nvidia Jetson will capture the number plate in real-time which will then be sent to law enforcement authorities. Due to IoT, the Nvidia Jetson can connect to all four cameras and communicate with them.

- There are even more possibilities in the field of deep learning. The design of hardware accelerators for neural network (NN) applications involves walking a tightrope amidst the constraints of low power, high accuracy and throughput, as said by Mittal et al. [Mit19]. The Nvidia Jetson might be a platform that is a perfect combination of these constraints.

2.2.6 Provided code for TegraObserver

Before starting with this thesis, there was already some code written for the TegraObserver which could read and set the clock frequency. In this code, two classes are created. One class called Tegra for reading and setting the clock frequencies on the Nvidia Jetson device. The other class utilizes this Tegra class in a new observer, called TegraObserver, to read or set the clock frequency when observing a kernel execution. Using the default settings of Kernel Tuner, the kernel is executed 7 times in which the observer observes the clock frequency. When all iterations are finished, the observer averages the results and returns these.

3 Implementation

In this section, the user interface of TegraObserver is explained, along with the extensions made to Kernel Tuner to enable optimizations of GPU applications on Nvidia Jetson devices for energy efficiency and the ContinuousObserver used for accurate power readings.

As described in Section 2.2.4, Kernel Tuner already has an observer that utilizes NVML to get and set properties such as clock frequency on Nvidia GPUs. However, the Tegra architecture cannot be used with NVML. For this reason, this thesis will contribute an observer which can get and set properties such as the clock frequency. The extension described in Section 3.2 (temperature readings) and Section 3.3 (power readings) will use the file system of the Nvidia Jetson devices to open specific files to read from or write to the Nvidia Jetson device. The section focuses primarily on the implementations of energy and temperature measurement, while there already is an extension for clock frequency.

3.1 User interface TegraObserver and example

To use the new TegraObserver effectively, there are some essential steps that the user must follow. To demonstrate the user interface, we will look at the benchmarking process of a matrix multiplication kernel using Kernel Tuner. Figure 2 provides an abstract overview of the kernel tuning process using the TegraObserver, showing the four possible monitoring options for the observer.

First of all, the user must enter all the necessary information to benchmark a kernel. Figure 2 shows that the input can be divided into two different sets. The first set of parameters is for specifying the kernel and its execution details. This includes specifying the kernel along with

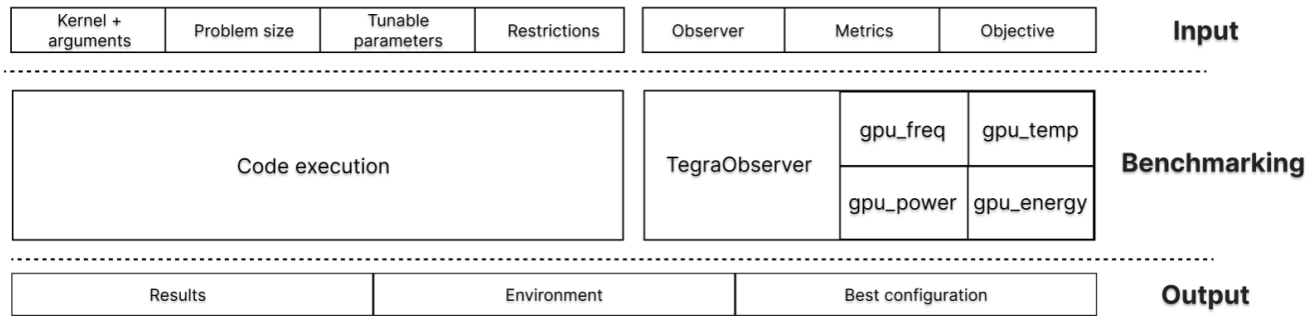

Figure 2: Interface of Kernel Tuner and the TegraObserver.

all arguments it utilizes. Additionally, to enable kernel tuning, the user should specify which parameters are to be tuned and define any restrictions on the possible configurations. The second set of input parameters is to configure and use the observer. To use the new observer, the user can utilize three parameters. First of all, the user must initialize the observer, in this case the TegraObserver. The TegraObserver offers four monitoring options: gpu_freq (GPU clock frequency), gpu_temp (GPU temperature), gpu_power (GPU power consumption) and gpu_energy (GPU energy consumption). The user can select one or more of these options based on their specific needs. In Figure 54, we can see an example of initiating the TegraObserver with all four options. Secondly, the user can enter metrics that will be computed from the measurements collected by the observer. For example, using the measurements collected, the user can compute the GFLOPS/W using the formula $GFLOPS/W = (totalflops/time)/power$, where time and power are measured by the observer. Lastly, the user can specify the optimization goals, for instance, maximizing energy efficiency measured in GFLOPS/W. The optimization goal is based on parameters measured or calculated using the observer, which is why it included in this input subset.

Figure 54 shows a code examples for the matrix multiplication kernel. This figure shows the initiation of the arguments used by the matrix multiplication kernel. Secondly, this figure shows the initiation of the TegraObserver using all four possible metrics. Following that, the tunable parameters are specified. We can call the tegra class from the TegraObserver to request which clock frequencies are possible. Additionally, we tune four different thread block sizes. Using the restrict variable, we specify that only square matrices should be benchmarked. After this, Figure 54 shows an example of user defined metrics. In this case, we calculate the total GFLOPS/W to measure the energy efficiency. The total amount of GFLOPS, execution time and GPU power are essential to measure the energy efficiency. The last line of Figure 54 shows an example of a call to the tune_kernel function, which will start the benchmarking process. To call this function, we first specify the name and the kernel which we want to tune, in this case the matrix multiplication kernel, alongside the problem size, arguments and tunable parameters. Thereafter, we specify the observer (TegraObserver), the metrics and the objective. The objective for this example is GFLOPS/W. We aim for higher GFLOPS/W values, so we set objective_higher_is_better to True. Kernel Tuner will now use these arguments to tune the matrix multiplication kernel and will return the configuration which performed the best, based on the objective.

3.2 Temperature readings

To read the temperature of the Nvidia Jetson at run time, two functions have been created. The first function finds the correct file where the GPU temperature information is stored when the Tegra class is initiated. The code is shown in Figure 55. When a kernel is executed and the user wants to read the temperature, the second function, shown in Figure 56, is executed. This function opens the temperature file, which was found using the first function. Now we only need to read the content stored in it. This is returned at last.

However, the user may prefer to manually enter the path to the temperature readings. When constructing the observer, the option to specify a path to the temperature directory is available. In this case, where the path to the temperature directory is manually entered, the function get_temp_path() will not be executed.

3.3 Power readings

Secondly, the extension to read the power consumption of the Nvidia Jetson device. When the Tegra class is initiated, we find the correct files where the current and voltage, used by the GPU, are stored. This code is shown in Figure 57. However, there are two different ways to store the power information. For example, the Nvidia Jetson Xavier uses hwmon (hardware monitoring) to store the hardware information, while the Nvidia Jetson Xavier AGX uses iio (industrial input output). To find the correct files on all devices, the code recursively tries all sub directories, starting from a common origin directory `"/sys/bus/i2c/drivers/ina3221"` until it finds the files which contain the power information. Secondly, we need to find the correct channel. Nvidia Jetson devices use channels to store information of hardware components. So, for example, channel 1 stores the information of the CPU and channel 2 the information of the GPU. The code to find the correct channel for the GPU is shown in Figure 58. When both directory and channel are found, we finally can read the power the device is using. To do so, we read both the current and voltage files and multiply those to get the total power used, in Watt. The code to read the power is shown in Figure 59.

However, the user may want to manually enter the path to the power directory as well. When constructing the observer, the option to specify a path to the power directory is available. In this case, where the path to the power directory is manually entered, the function get_power_path() will not be executed.

3.4 ContinuousObserver

During our experiments with various kernels, we observed that when we used the TegraObserver to measure power consumption, it sometimes took over a second to stabilize the power readings. Figure 3 illustrates the time required to reach a stable power level using the TegraObserver. If we use the observer on small kernels, which may execute in under a second, we will only read power levels which are still rising and have not yet been stabilized, which can result in inaccurate power readings. To address this issue, an addition was made to the TegraObserver. Now, the ContinuousObserver from Kernel Tuner is used to measure power consumption. The main difference between the TegraObserver and the ContinuousObserver, is that the ContinuousObserver does not run the kernel once, but over an extended period of time. By doing this, we can reach a stable

power level and take the median to get an accurate power reading. We use the median to make sure that the measurement is not influenced by outliers, which would happen if we use the mean. For the ContinuousObserver used by the TegraObserver, we decided that running a kernel for three seconds is sufficient to reach a stable power level, as can be seen in Figure 3.

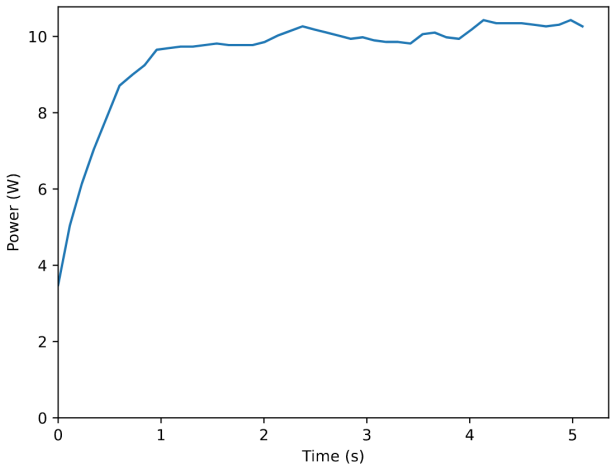

Figure 3: The time it takes to reach a stable power level when using the TegraObserver without the ContinuousObserver.

3.5 Tuning strategies

To tune a kernel, there are numerous options. In Section 2.2.4 we have discussed that Kernel Tuner can use Basin Hopping and Bayesian Optimization among other strategies to tune a kernel. In this thesis, we will use four different tuning strategies: RaceToIdle, BlockToFreq, FreqToBlock and GlobalEnergy. The first technique, RaceToIdle, tunes for the fastest execution time. GlobalEnergy focuses on the best energy efficiency across the entire search space, which includes every combination of thread block size and clock frequency. For the rgb2gray kernel, used in Section 4.5, a total of 48 different thread block configurations are possible and each thread block configuration is tested on all 15 different clock frequencies. This results in a total of 720 (48 * 15) different configurations. This is a relatively small number of possible configurations, while we only tune for two different parameters. However, the number of possible configurations can increase rapidly when we tune for more than two parameters. We can use BlockToFreq and FreqToBlock try to reduce the search space by optimizing one parameter while holding the other constant. Both strategies then select the best configuration for energy efficiency from these results and optimize the other parameter. Through BlockToFreq and FreqToBlock, we can reduce the number of configurations to 63 (48 + 15), which is a reduction of 91.39% compared to GlobalEnergy. This strategy of tuning parameters individually can be applied when tuning for more than two parameters as well. In these cases, all parameters except one should be kept constant. The best choice for the parameter which is tuned should then be used when tuning the other parameters, until all parameters have been tuned.

4 Experiments

This section explains the experimental setup which was used to conduct five benchmarks, the result of the experiment involving the temperature and the results of the benchmarks.

4.1 Experimental setup

	Jetson AGX Xavier series			Jetson Xavier NX series	
	Jetson AGX Xavier Industrial	Jetson AGX Xavier (64GB)	Jetson AGX Xavier (32GB)	Jetson Xavier NX (16GB)	Jetson Xavier NX (8GB)
AI Performance	30 TOPS	32 TOPS		21 TOPS	
GPU	512-core NVIDIA Volta architecture GPU with 64 Tensor Cores			384-core NVIDIA Volta™ architecture GPU with 48 Tensor Cores	
GPU Max Frequency	1211 MHz	1377 MHz		1100 MHz	
CPU	8-core NVIDIA Carmel Arm®v8.2 64-bit CPU 8MB L2 + 4MB L3			6-core NVIDIA Carmel Arm®v8.2 64-bit CPU 6MB L2 + 4MB L3	
CPU Max Frequency	2.0 GHz	2.2 GHz		1.9 GHz	

Figure 4: The specifications of the Nvidia Jetson AGX Xavier series and the Nvidia Jetson NX series. This thesis utilizes the 8GB version of the Nvidia Jetson Xavier NX.

The Nvidia Jetson Xavier NX (8GB) which was used to perform the benchmarks runs on Ubuntu 20.04.6 LTS (Focal Fossa), uses Python version 3.8.10 and CUDA Toolkit version 11.4. The hardware of the Nvidia Jetson Xavier NX is shown in Figure 4 [spe]. For the experiments, the 20W mode of the Nvidia Jetson Xavier NX was used. By capping the power, we ensure that the device does not consume excessive power and so, produce a lot of heat. This could otherwise be disadvantageous for operational conditions.

Firstly, we will examine the effect the temperature has on the energy efficiency. To do so, we will execute 500 iterations of the same configuration using a matrix multiplication kernel. By analyzing the relationship between energy efficiency, performance and temperature, we can determine if temperature affects the energy efficiency and/or performance.

To benchmark the energy efficiency tuning capabilities of Kernel Tuner with Nvidia Jetson GPUs, we will run five different kernels on the Nvidia Jetson Xavier NX. Each benchmark we have chosen to use are kernels which are meant to run on low-power devices, such as the Nvidia Jetson devices. Each benchmark will look into the effect of the thread block size and clock frequency to investigate how well we can tune GPU applications on the Nvidia Jetson Xavier NX to be more energy efficient.

The Nvidia Jetson Xavier NX has in total 15 different clock frequencies on which it can run, between 0.11475 GHz and 1.10925 GHz. For the benchmarks, we will only show 5 different clock

frequencies to make the graphs more readable. The five different clock frequencies were chosen with space-evenly, starting with the lowest clock frequency and with steps of 3.

The maximum thread block size we will use is 1024, while the maximum of threads per block is 1024 for CUDA programming on Nvidia GPUs [LZR+19]. Not all benchmarks will use the same thread block sizes when executing the kernel. Each benchmark will describe which thread block sizes it will use to optimize.

To determine the energy efficiency of each benchmark, we will analyze three different figures: one for power consumption, another for performance and one to indicate the energy efficiency of the kernel. Thereafter, we will analyze the energy efficiency for each configuration and the reduction in performance when using the best configuration for energy efficiency compared to the best configuration for performance.

Lastly, we will compare the four tuning strategies, discussed in Section 3.5. To do so, we will plot plot the energy consumption in Joules for all four strategies to see if there are any significant differences in power consumption between the strategies.

4.2 Benchmarks

This section describes the five benchmarks used in our experiment. Each benchmark has unique features and functions, enabling us to explore various aspects of kernel code.

4.2.1 Matrix multiplication

Matrix multiplication is a kernel which multiplies two matrices. For the experiment, we use two 4096x4096 matrices and randomly assign values to these matrices. Matrix multiplication kernels can be computationally expensive. Multiplying two n x n matrices results in a complexity of $O(n^3)$ [Sto10]. This demonstrates that as the matrix size increases, the computational complexity grows polynomial, placing significant demands on the GPU. Matrix multiplications are used in various fields, such as simulations, data analysis, physics and more. For example, matrix multiplication are used in weather forecasting [HCDP19] and in fluid simulation [WWR+16] to accelerate computations.

4.2.2 Rgb2gray

Rgb2gray is a kernel which converts an image to grayscale. In our experiment, we created a 20000x20000 image and assigned random RGB values to each pixel. This image is then processed by the kernel uses the formula $0.299f * c.x + 0.587f * c.y + 0.114f * c.z$ to turn each pixel into grayscale. The rgb2gray kernel is used in various applications. In computer vision, it simplifies the processing and analysis of visual data, making it easier to detect shapes and patterns in grayscale images [KC12]. In medical imaging, converting images like X-rays or MRIs to grayscale enhances the visibility of structures, aiding in diagnostic accuracy [YYAS11]. Converting images to grayscale simplifies data analysis while preserving crucial information, making it valuable across different fields where clear visualization and efficient processing are essential.

4.2.3 Raycasting

Raycasting is a kernel which is a tool used by robots engaged in terrain mapping. It functions by sending out multiple rays at different angles to figure out how far away the nearest object is for

each ray 4.6. In our experiment, we simulated this mapping process with 10000 rays, each launched at random angles. To emulate the process of mapping a terrain, we utilized a precomputed distance map of the terrain, which provides the closest object distance from any point. Raycasting finds wide application in various fields, including autonomous vehicles. Here, the kernel's ability to determine distances to objects and map terrain is crucial. It enables autonomous vehicles to analyze their environment, which is crucial for navigation and obstacle avoidance 4.6. This makes the raycasting kernel a valuable tool for enhancing navigation and obstacle avoidance capabilities in robotics.

4.2.4 Bandpass correction

This kernel is used in LOFAR (LOw Frequency ARray). In radio astronomy, LOFAR is an array of radio telescopes operating in the 10 to 250 MHz frequency range [RBMvN10]. It comprises thousands of antennas organized into a distributed sensor network for sky monitoring. There are two kinds of antennas, Low-Band Antennas (LBA) and High-Band antennas (HBA). Each antenna samples signals as complex numbers representing amplitude and phase at specific times. The antennas are omni-directional and can observe in all directions and in all frequencies at the same time. Observers select specific directions and frequencies, called subbands, to focus data collection. Data is filtered and subbands are split into narrower frequency bands called channels to improve noise removal accuracy. The signals from these antennas are aggregated centrally after initial processing within groups known as stations, which reduce computational load. Each station independently employs FPGAs for tasks like analog-to-digital conversion, filtering, frequency selection and signal combination from multiple receivers [RBMvN10]. Radio astronomy institutions, such as ASTRON, are looking at Nvidia Jetson devices to replace the FPGAs.

The bandpass correction kernel compensates for an anomaly introduced by the FPGAs in LOFAR stations. In both LBA and HBA antenna systems, an issue arises from not removing the conversion of correlator data into the frequency domain using a poly-phase filter. This process leaves a signature in the data that varies in frequency [DGDD+19]. The bandpass correction kernel fixes this issue by applying a filter to ensure all parts of the signal have the same strength. The bandpass correction kernel can be found at https://git.astron.nl/RD/AARTFAAC.

4.2.5 Correlator

The correlator kernel is used in LOFAR as well. The correlator kernel processes samples received from individual or grouped stations. Given the weak, noisy signals from the sky, the kernel tries to find meaningful patterns in this chaos of signals. The correlator correlates the signals from different receivers and integrates the correlations over time to reduce the amount of data [vNR10]. Correlation may reveal patterns that might otherwise be hidden by noise. For the benchmark, we will focus on 1, 2, 3 or 4 stations per thread. For each configuration, we calculate the number of threads required to process the specified number stations per thread. The correlator kernel can be found at https://git.astron.nl/RD/AARTFAAC.

4.3 Temperature

As described in Section 2.1.2, the temperature of a device can influence the performance and energy efficiency. To test how much the temperature of the Nvidia Jetson Xavier NX changes and if the

temperature influences the performance and energy efficiency of this device, we run 500 iterations of the matrix multiplication kernel using the same configuration each time. Figure 5 shows that the Nvidia Jetson Xavier NX was able to keep the temperature relatively constant during the experiment, while it was able to keep the temperature between 45.5°C and 48°C during all 500 iterations. In Figure 5 we can see that both the performance and energy efficiency stay almost constant. The maximum energy efficiency, which was measured during these iterations, was 16.90 GFLOPS/W, while the lowest was 16.72 GFLOPS/W. This means that the difference between the highest and lowest measurement was only 1.08% during all 500 iterations. For the performance, the highest performance measured was 87.23 GFLOPS/s, while the lowest performance measured was only 86.93 GFLOPS/s. This means that the difference between the highest and lowest measurement was only 0.35% during all 500 iterations. The consistent temperature control by the Nvidia Jetson Xavier NX ensured that both energy efficiency and performance remained largely unchanged during the kernel execution.

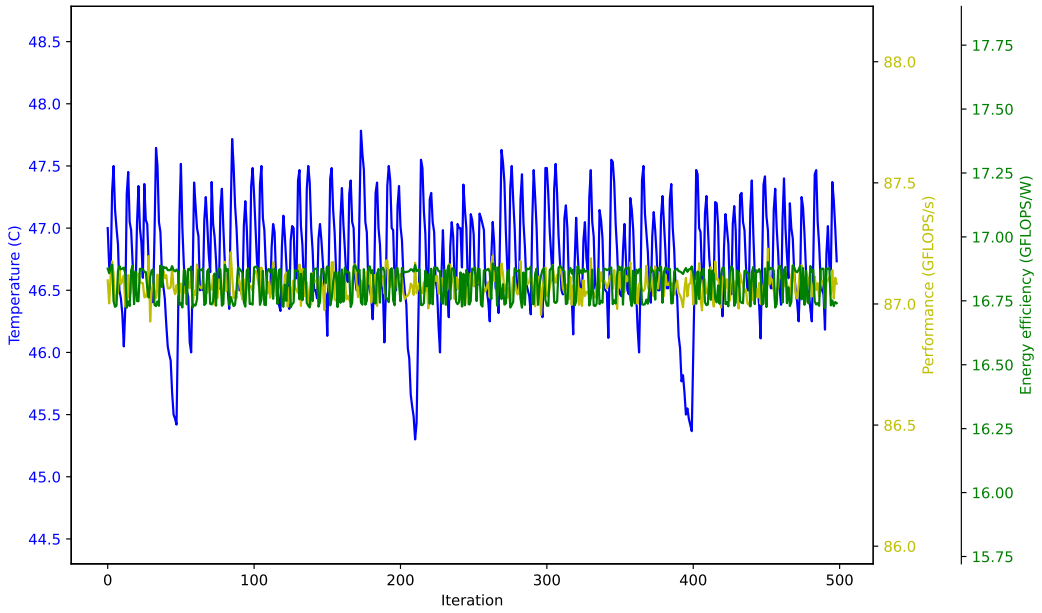

Figure 5: Effect of temperature on performance and energy efficiency.

4.4 Matrix multiplication

This benchmark will only use 4 different thread block sizes: 8x8, 16x16, 24x24 and 32x32. Each of these configurations will be tested to evaluate their impact on performance and energy efficiency.

4.4.1 Comparison of thread block size and clock frequency

Starting with the power consumption of the various clock frequencies. Figure 6 indicates a clear trend: higher clock frequencies consume more power. Notably, the difference between the clock frequencies and their respective power consumption shows minimal difference.

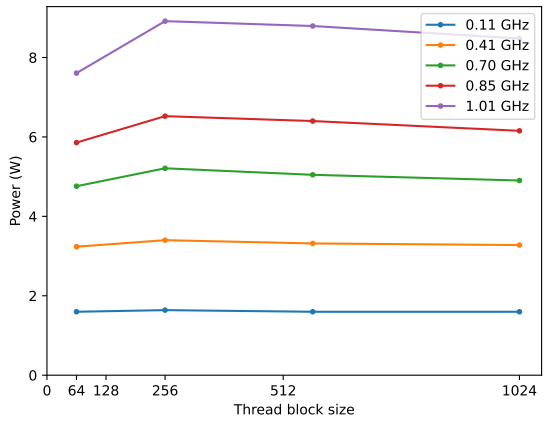

Figure 6: The power consumption of five different clock frequencies.

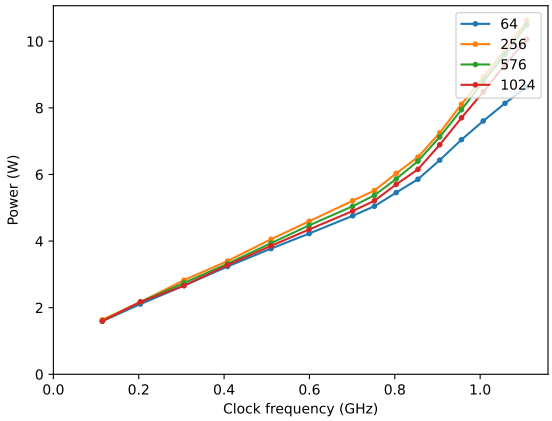

Figure 7: The power consumption of the four different thread block sizes.

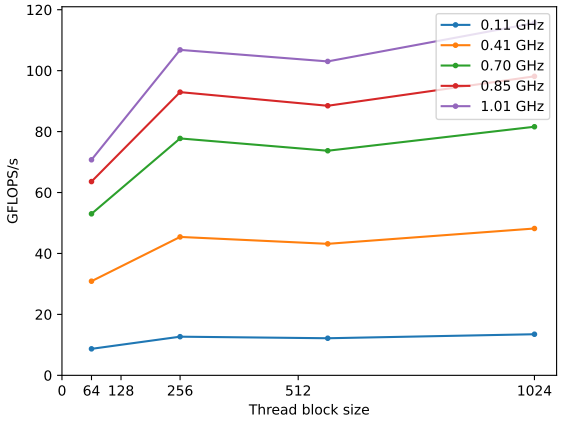

Figure 8: The performance of five different clock frequencies.

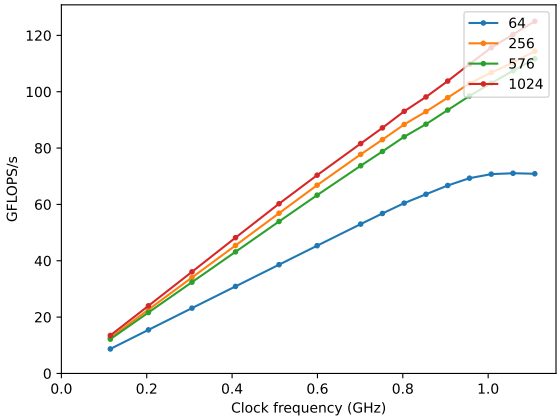

Figure 9: The performance of the four different thread block sizes.

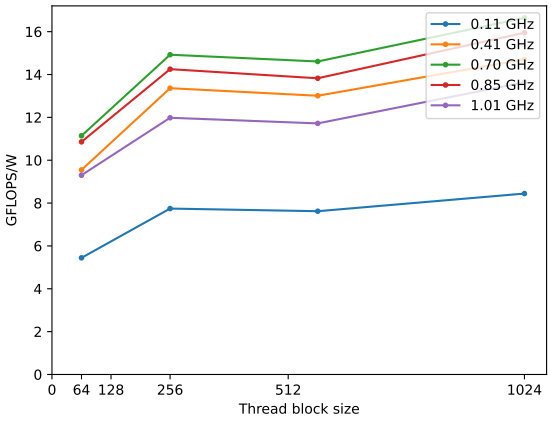

Figure 10: The energy efficiency of five different clock frequencies.

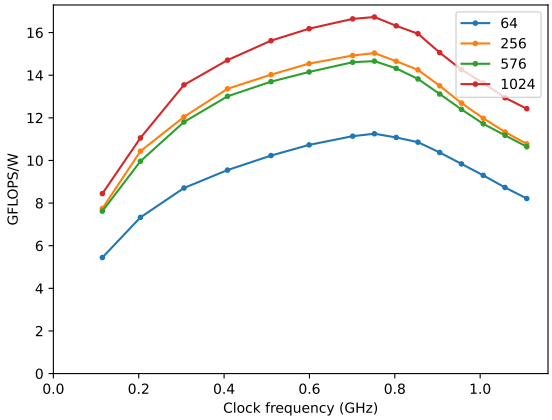

Figure 11: The energy efficiency of the four different thread block sizes.

Turning our attentions to Figure 7, we analyze the power consumption of different thread block sizes. It's noteworthy that all four thread block sizes increase at a similar rate, except for a minor variation observed with thread block size 64.

Moving on to Figure 8, the performance in GFLOPS/s for different clock frequencies is illustrated. Notably, lower clock frequencies demonstrate worse performance, while differences among higher frequencies, 0.70125 GHz, 0.85425 GHz and 1.00725 GHz, are noticeable but less pronounced compared to the lower frequencies.

While we were able to see relatively big differences in performance in Figure 8, Figure 9 shows less pronounced differences in performance. Thread block sizes 256, 576 and 1024 all demonstrate a nearly identical increase in performance, with only minimal differences. In contrast, Figure 9 reveals that a thread block size 64 only performs nearly half as good as that of the others.

Moving on to energy efficiency. Figure 10 highlights the energy efficiency of each clock frequency in GFLOPS/W. Interestingly, the highest performing clock frequency in terms of GFLOPS/s ranks second to last in energy efficiency, while 0.70125 GHz performs the best for in terms of energy efficiency. There is no overlap between the clock frequencies, meaning that a clock frequency of 0.70125 GHz performs the best for all thread block sizes.

Lastly, Figure 11 shows the energy efficiency of the different thread block sizes. The thread block size of 64 has the worst energy efficiency and the thread block size of 1024 has the best energy efficiency. There is no overlap between different block sizes in the figure, so a thread block size of 1024 is better in all cases than the other thread block sizes, etc. What is particularly noteworthy is that the energy efficiency increases when the clock frequency is increased for all four thread block sizes, but for clock frequencies above 7.5225 GHz, the energy efficiency decreases again.

We have identified configurations demonstrating better energy efficiency, but not the reasons why. For matrix multiplication, a higher thread block size is more energy efficient, while it is highly parallelizable. This means that it can store element independently, meaning that threads do not have to wait for one another. High thread block sizes allow the streaming multiprocessors (SMs) to be fully utilized, leading to faster execution and better energy efficiency. Smaller thread block sizes, such as 64, underutilizes the GPU, leading to worse performance and energy efficiency. Secondly, the matrix multiplication kernel prefers a mid-range clock frequency. Higher frequencies execute more calculations per second, but require more power. Lower frequencies use less power, but take longer to execute. The total energy used is given by the formula $E = P * t$, where E is the energy in Joules, P the power in Watt and t the execution time in seconds. Using this formula, we can derive that the middle range frequencies consume less energy overall, while the power increase outweighs the time saved.

4.4.2 Energy efficiency and performance

Figure 12 illustrates the energy efficiency of executing the kernel for each combination of thread block size and clock frequency. Figure 12 highlights that a higher thread block size is more energy efficient, but only in combination with a clock frequency ranging from 0.4 GHz to 0.8 GHz. The figure depicts that the best configurations for the matrix multiplication kernel is a thread block size of 1024 with a clock frequency in the middle range of possible frequencies, as can be seen in Figure 10 and Figure 11 as well.

Figure 13 shows the relationship between performance and energy efficiency for all possible

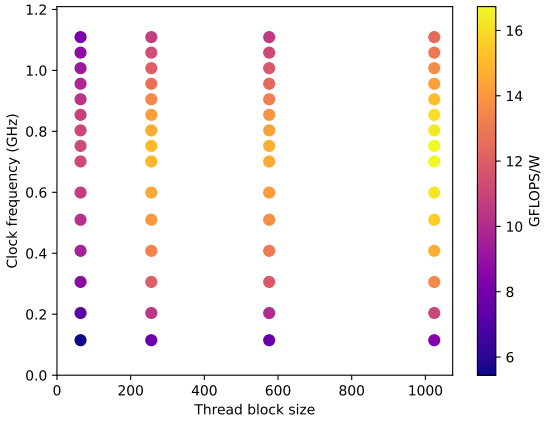

Figure 12: The energy efficiency for all combinations of thread block sizes and clock frequencies.

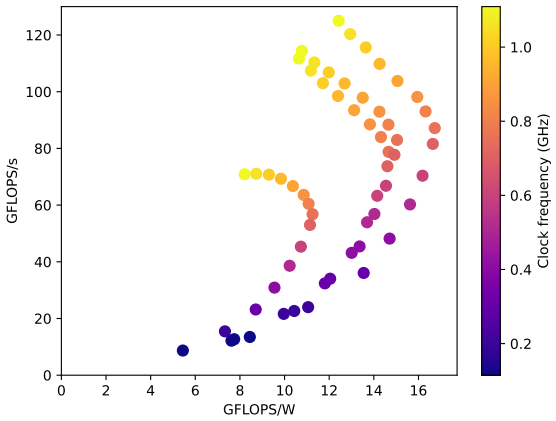

Figure 13: Relationship between performance and energy efficiency.

configurations. Here we can see the trade-off between compute performance and energy efficiency of the fastest and most energy efficient configuration. The most energy efficient configuration shows an improvement of 34.60% in terms of energy efficiency, but is 30.28% slower than the fastest configuration. So achieving better energy efficiency requires a significant compromise in performance.

4.4.3 Tuning strategies

Figure 14 presents the result of the four different tuning strategies, discussed in Section 4.1. Tuning for RaceToIdle results in the highest energy consumption. Compared to GlobalEnergy, RaceToIdle consumes 34.55%/2.84J more energy. Interestingly, it does not matter if we use BlockToFreq, FreqToBlock or GlobalEnergy to find the combination of parameters which is the most energy efficient.

To explain the difference between BlockToFreq and FreqToBlock, we first look at BlockToFreq. BlockToFreq first optimizes the thread block size while keeping the clock frequency constant at 1.10925 GHz. Using Figure 11, we can identify that a thread block size of 1024 is the most energy efficient. With a thread block size of 1024, we can now optimize the clock frequency. Figure 10 indicates a clock frequency of 0.7 GHz as most energy efficient. In the second case, FreqToBlock, we first optimize for clock frequency while keeping the thread block size constant at 256. Figure 10 shows that the best clock frequency is 0.7 GHz in combination with a thread block size of 256. Using a clock frequency of 0.7 GHz to optimize for thread block size, we can see in Figure 11 that this results in a thread block size of 1024. Lastly, GlobalEnergy will return the best configuration for energy efficiency over the complete search space, which again is a thread block size of 1024 in combination with a clock frequency of 7.0 GHz. So all three strategies return the same configuration and energy consumption.

Both BlockToFreq and FreqToBlock only have 19 (15 clock frequencies + 4 thread block sizes) different configurations, while GlobalEnergy has 60 (15 * 4) different configurations. However,

all three strategies return the same optimal configuration. So, we can use either BlockToFreq or FreqToBlock to explore 68.33% less configuration and speed up the tuning process.

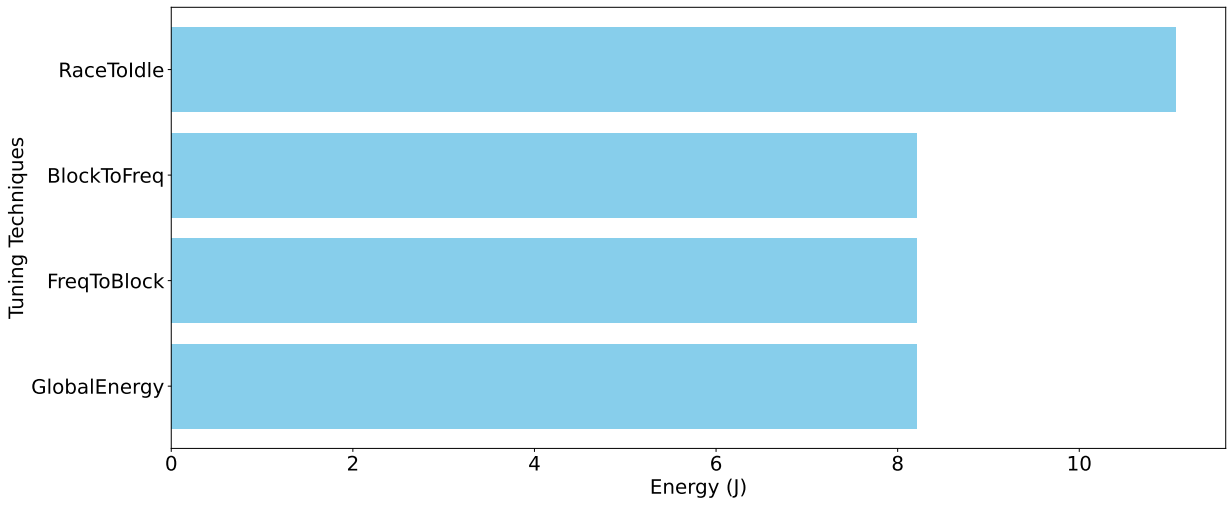

Figure 14: The four different tuning techniques with the total energy used.

4.5 Rgb2gray

To tune the rgb2gray kernel, we use the block size x and block size y. We multiply the block size x and the block size y to get the total thread block size and take the best performing configurations for a thread block size of 32, 64, 128, 256, 512 and 1024 to provide a clearer comparison of their differences.

4.5.1 Comparison of thread block size and clock frequency

To analyze the rgb2gray kernel, let's start with the clock frequency. The same trend can be seen in the rgb2gray kernel as we have seen in the matrix multiplication kernel: higher frequencies consume more power. Figure 17 illustrates that the lowest clock frequency has the poorest performance in terms of GFLOPS/s. When we increase the clock frequency, the difference between clock frequencies decreases. These findings are shown in Figure 19, where we can see the energy efficiency for each clock frequency. Figure 19 shows similarities with Figure 10 (matrix multiplication). In both figures, the middle range of clock frequencies show the best energy efficiency. However, for the rgb2gray kernel, the energy efficiencies 0.11475 GHz, 0.408 GHz and 0.70125 GHz are nearly identical.

Regarding the differences in thread block size, Figure 16 indicates that the thread block size of 32 uses the least power, 64 uses slightly more, but for the remaining thread block sizes, the power usage is nearly identical. The same result can be seen in Figure 18, which shows that a thread block size of 32 again has the poorest performance and 64 performs slightly better. The remaining clock frequencies have nearly identical performance. These findings are summarized in Figure 20, where we can see the energy efficiency of all thread block sizes. We can see that the thread block size of 32 has the worst energy efficiency, which is followed by a thread block size of 64. All other

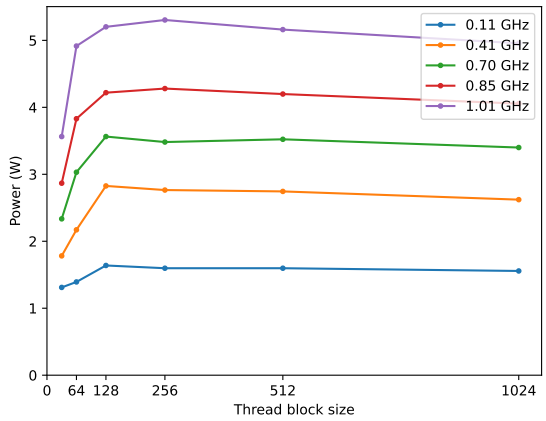

Figure 15: The power consumption of five different clock frequencies.

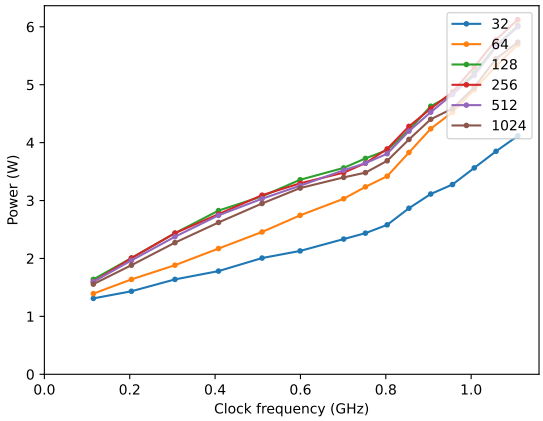

Figure 16: The power consumption of the six different thread block sizes.

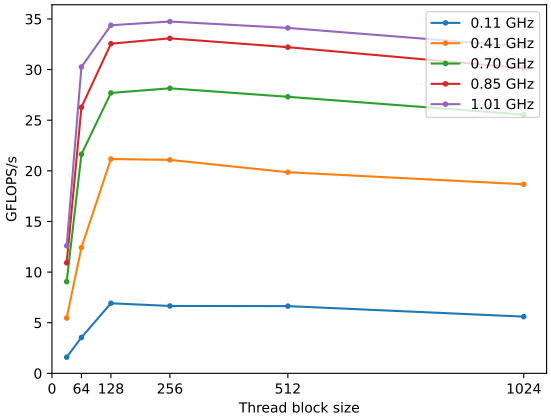

Figure 17: The performance of five different clock frequencies.

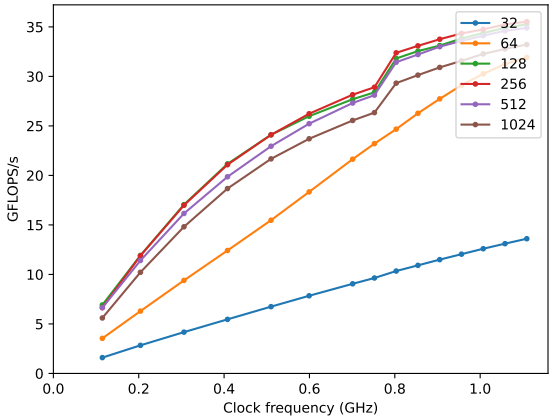

Figure 18: The performance of the six different thread block sizes.

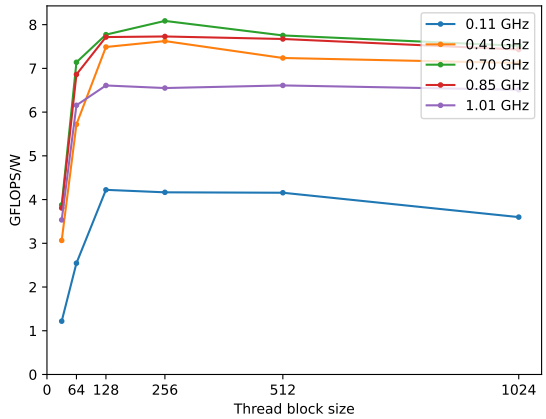

Figure 19: The energy efficiency of five different clock frequencies.

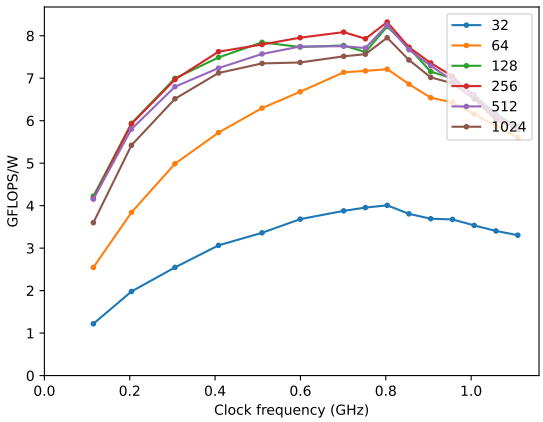

Figure 20: The energy efficiency of the six different thread block sizes.

thread block sizes show minimal differences in energy efficiency.

The reason why the rgb2gray kernel prefers specific configurations, starting with the clock frequency. When we compare Figure 19 (rgb2gray) to Figure 10 (matrix multiplication), we can see the same results: the middle range of clock frequencies performs the best in terms of energy efficiency. As explained in Section 4.4.1, the upper range consumes almost 35% more power than the middle range, but is only 22% faster. The difference in power consumption and performance means that in total, the upper range will use more energy than the middle range. On the other hand, the lower range consumes around 30% less power, but takes almost five times longer to execute, which results in the middle range to be the most energy efficiency. Secondly, the thread block size. The rgb2gray is highly parallelizable, which was the case with the matrix multiplication kernel as well. As explained in Section 4.4.1, kernels which are highly parallelizable prefer a higher thread block size. We can see that the rgb2gray kernel prefers a higher thread block in Figure 16, Figure 18 and Figure 20 as well. All three figures show that a higher thread block size performs better.

4.5.2 Energy efficiency and performance

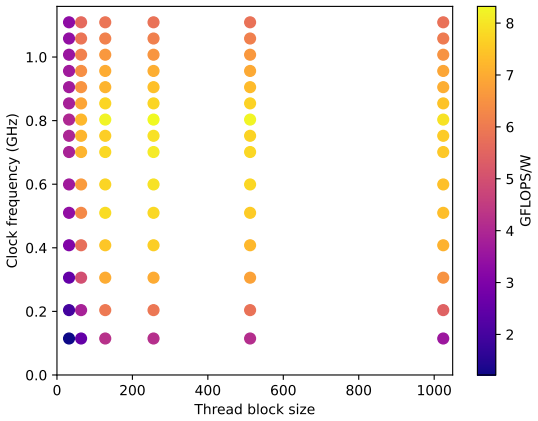

Figure 21: The energy efficiency for all combinations of thread block sizes and clock frequencies.

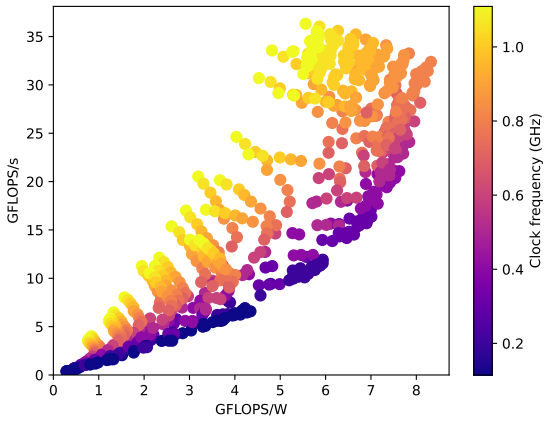

Figure 22: Relationship between performance and energy efficiency.

Figure 21 shows the energy efficiency for all configurations for the rgb2gray kernel. The difference between Figure 21 (rgb2gray) and Figure 12 (matrix multiplication) is that in Figure 21, we can see a lot more configurations which have almost the same energy efficiency. To understand why, we look at both Figure 19 and Figure 20. Figure 19 illustrates that the middle range of possible frequencies all show minimal difference in energy efficiency. Figure 20 illustrates that thread block sizes ranging from 128 to 1024 all shows minimal differences in energy efficiency as well. Combining both findings, we get a larger range of excellent performing configuration in terms of energy efficiency.

Figure 22 shows the relationship between performance and energy efficiency for all possible configurations. Here we can see the trade-off between compute performance and energy efficiency of the fastest and most energy efficient configuration. We can see that the most energy efficient configuration is 49.64% more energy efficient and only 11.12% slower than the fastest configuration.

This is significant, while it means that with only a small reduction in performance, we can achieve a remarkable improvement in energy efficiency.

4.5.3 Tuning strategies

Figure 23 shows the result of the four different tuning strategies, discussed in Section 4.1. First of all, RaceToIdle again uses the most energy, as was the case with the matrix multiplication kernel. For the rgb2gray kernel, RaceToIdle consumed 50%/0.06J more energy than GlobalEnergy. When we compare BlockToFreq, FreqToBlock and GlobalEnergy to the matrix multiplication kernel, shown in Figure 14, we can see a noteworthy difference. Interestingly, BlockToFreq consumes more energy than FreqToBlock and GlobalEnergy.

BlockToFreq keeps the frequency constant at 1.10925 GHz when first tuning the thread block size. For this clock frequency, the best possible thread block size is 128. Using a thread block size of 128, the best possible clock frequency is 0.80325 GHz. FreqToBlock, when first optimizing the clock frequency, finds a clock frequency of 0.80325 GHz as well. But for this clock frequency, the best thread block size is 256 and not 128. This shows the difference between BlockToFreq and FreqToBlock. The best possible configurations, found by GlobalEnergy, is the combination of 0.80325 GHz and a thread block size of 256, as was the case with FreqToBlock. BlockToFreq is not able to find the best optimal configuration, whereas FreqToBlock is. BlockToFreq consumes 1.31% more energy compared to FreqToBlock and GlobalEnergy.

In total, there are 48 different combinations of block size x and block size y, resulting in total thread block sizes ranging from 32 to 1024. To determine the best configuration using GlobalEnergy, we need to test all 720 (15 * 48) possible configurations. However, for both FreqToBlock and BlockToFreq, there are only 63 (15 + 48) possible configurations, representing a significant reduction of 91.25% in possible configurations. As explained, BlockToFreq does not result in the best possible configuration, but consumes 1.31% more energy. However, in contrast to a significant reduction of 91.25%, the small reduction in performance represents only a minor trade-off.

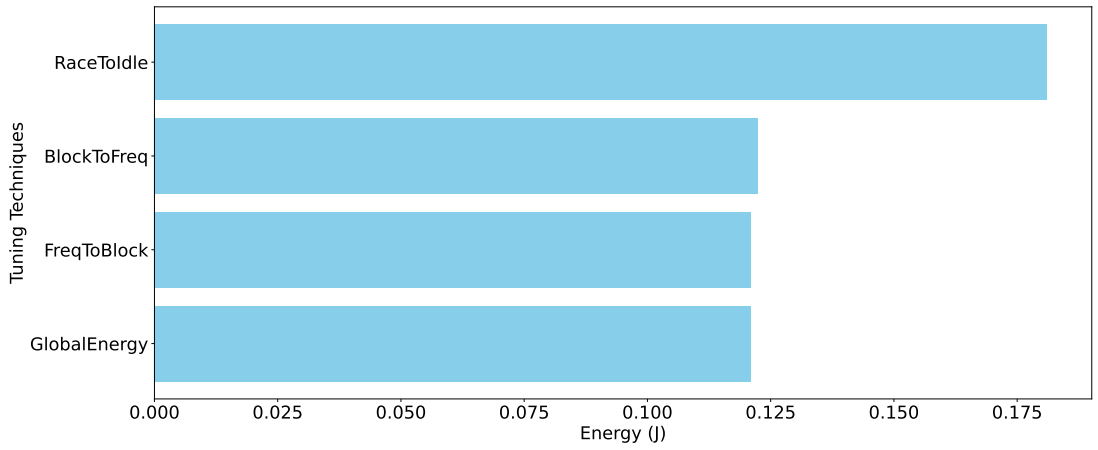

Figure 23: The four different tuning techniques with the total energy used.

4.6 Raycasting

To optimize the raycasting kernel, we use 6 different thread block sizes. These thread block sizes all vary along the block size x dimension.

4.6.1 Comparison of thread block size and clock frequency

To analyze the raycasting kernel, we start with the clock frequency. Figure 24 shows the same trend we have seen in both matrix multiplication and rgb2gray: higher clock frequencies consume more power. Figure 26 illustrates that the lowest clock frequency is significantly worse than the other frequencies. However, when we increase the clock frequency, the total difference between two clock frequencies reduces. The highest frequency shown, 1.01 GHz, performs just a little better than 0.85 GHz. The energy efficiency of the clock frequencies is shown in Figure 28. Here we can see that the middle range of clock frequencies perform almost identical in terms of energy efficiency and are better in terms of energy efficiency than the upper and lower range.

To examine the impact of different thread block size, we start with Figure 25. Figure 25 shows a difference when compared to Figure 7 (matrix multiplication) and Figure 16 (rgb2gray). In Figure 25, all thread block sizes consume nearly the same amount of power. Furthermore, Figure 27 shows that almost all thread block sizes perform the same, with only the thread block size of 32 performing slightly better. Lastly, Figure 29 shows that while all thread block frequencies perform the same in the case of energy efficiency, the thread block size of 32 is marginally better due to its better performance.

The reason why the middle range of clock frequencies has better energy efficiency than the upper and lower range, is explained before in Section 4.4.1. The raycasting kernel again shows that, while the upper range is faster, the middle range has a better balance between execution time and power consumption. The raycasting kernel shows a significant difference compared to the matrix multiplication and rgb2gray kernels. While the matrix multiplication kernel and the rgb2gray kernel preferred a higher thread block size, the raycasting kernel prefers the lowest possible thread block size. The reason why the raycasting kernel prefers a smaller thread block size, is that first of all, the raycasting kernel is not as parallelizable as the matrix multiplication and rgb2gray kernel. The raycasting kernel can have varying workloads per thread, while the raycasting kernel contains conditional statements, which in turn can vary the amount of work each thread performs. Varying workloads per thread can cause imbalances between threads, while threads can have different execution paths (thread divergence). When thread divergence happens, the GPU serializes their execution. Serialization of the execution means that only one path of the conditional statement is processed at a time. While the thread block is only executing one path, threads that follow a different path are idle, waiting for their turn [ZJGS10]. This decreases the performance of the GPU significantly. To address the issue of idle threads, smaller thread block sizes are preferred. Smaller thread block sizes allow the imbalances to be localized, while fewer threads have to remain inactive while waiting for other threads to finish.

4.6.2 Energy efficiency and performance

Figure 30 shows the energy efficiency for all configurations for the rgb2gray kernel. This figure shows, unsurprisingly, that all best configurations for energy efficiency have a thread block size of

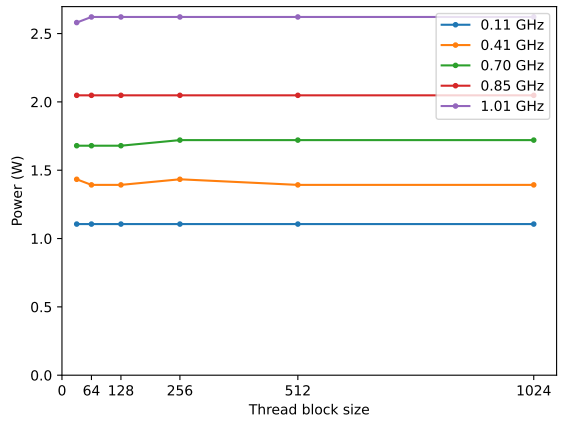

Figure 24: The power consumption of five different clock frequencies.

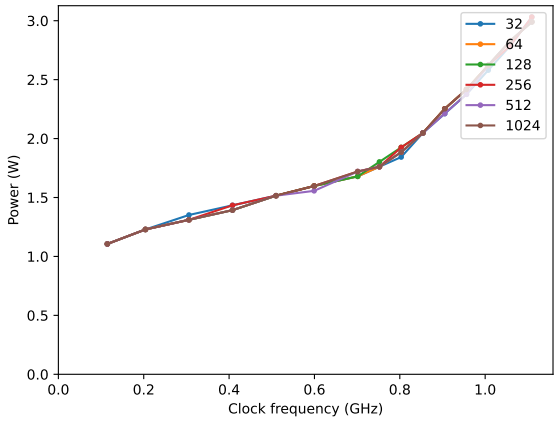

Figure 25: The power consumption of the six different thread block sizes.

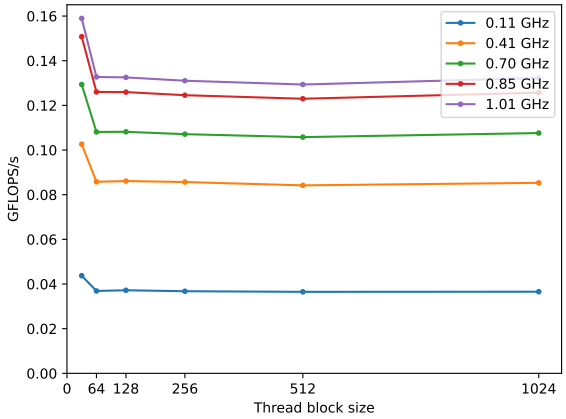

Figure 26: The performance of five different clock frequencies.

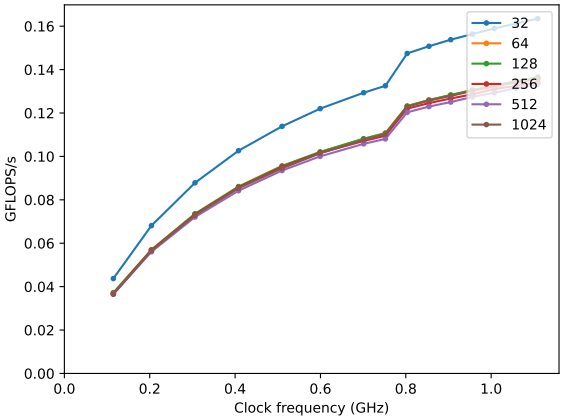

Figure 27: The performance of the six different thread block sizes.

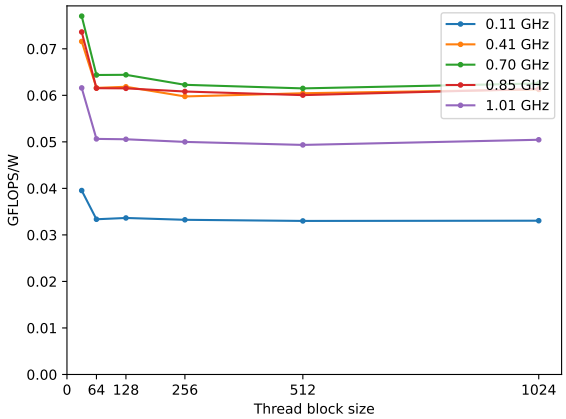

Figure 28: The energy efficiency of five different clock frequencies.

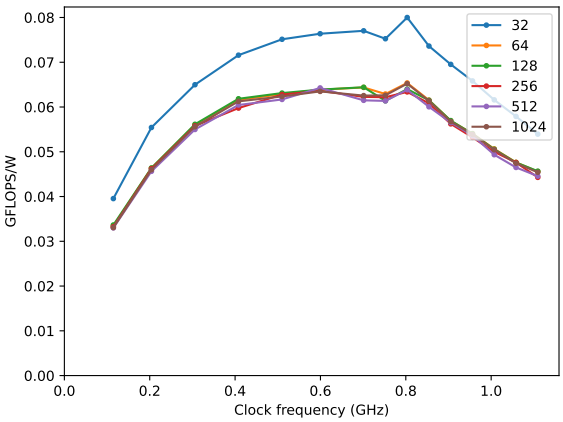

Figure 29: The energy efficiency of the six different thread block sizes.

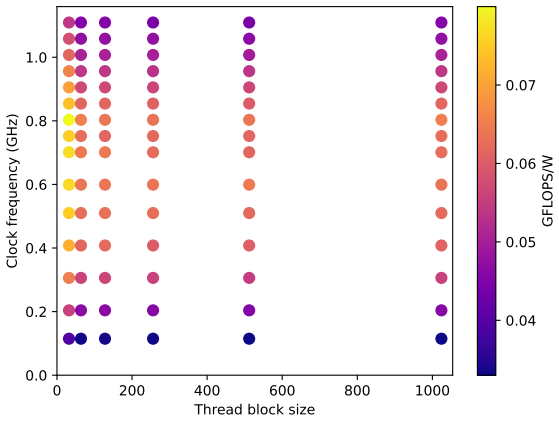

Figure 30: The energy efficiency for all combinations of thread block sizes and clock frequencies.

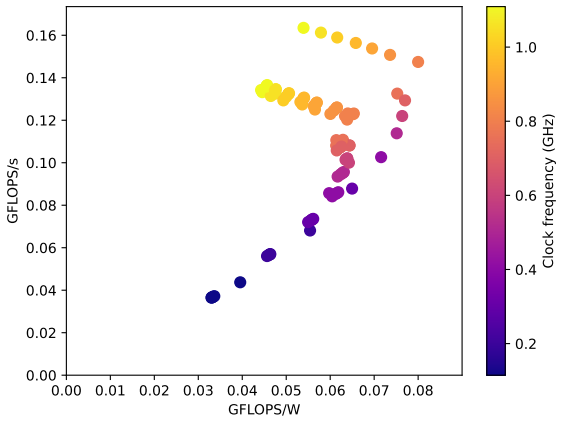

Figure 31: Relationship between performance and energy efficiency.

32, which was explained in Section 4.6.1. The middle range of clock frequencies perform almost the same, which we can see in Figure 28 as well.

Figure 31 shows the relationship between performance and energy efficiency for all possible configurations. Here we can see the trade-off between compute performance and energy efficiency of the fastest and most energy efficient configuration. We can see that the most energy efficient configuration is 48.15% more energy efficient and only 9.80% slower than the fastest configuration. This is a notable result, as we can substantially improve energy efficiency with minimal loss in performance.

4.6.3 Tuning strategies

Figure 32 shows the result of the four different tuning strategies, discussed in Section 4.1. Figure 32 shows that RaceToIdle uses the most amount of energy, in total RaceToIdle consumed 48.32%/0.91J more energy than GlobalEnergy. However, when we compare the results of BlockToFreq, FreqToBlock and GlobalEnergy to the other kernels, we can see a difference. For the raycasting kernel, FreqToBlock uses more energy compared to BlockToFreq and GlobalEnergy.

To explain the difference between BlockToFreq and FreqToBlock, we first examine FreqToBlock. FreqToBlock keeps the thread block size constant at 256 when first optimizing the clock frequency. This results in 0.59925 GHz as the best performing frequency. Using 0.59925 GHz, a thread block size of 32 is the best option. This configuration consumes in total 1.98J. On the other hand, BlockToFreq discovers that 32 is the best thread block size as well. The difference here is that BlockToFreq then discovers that 0.80325 GHz is the best option in combination with a thread block size of 32. This results in energy consumption of only 1.89J. So FreqToBlock consumes 4.76% more energy compared to BlockToFreq. GlobalEnergy finds the combination of a thread block size of 32 and clock frequency of 0.80325 GHz as well, showing that BlockToFreq did find the best configuration.

In total, there are 6 different thread block sizes and 15 clock frequencies. This results in 90 different configurations. To determine the best configuration using GlobalEnergy, we need to test

all 90 possible configurations. However, for both FreqToBlock and BlockToFreq, there are only 21 possible configurations. This represents a reduction of 76.67% in possible configurations. With that knowledge, we can save a lot of time tuning the raycasting kernel by using BlockToFreq, while FreqToBlock consumes more power.

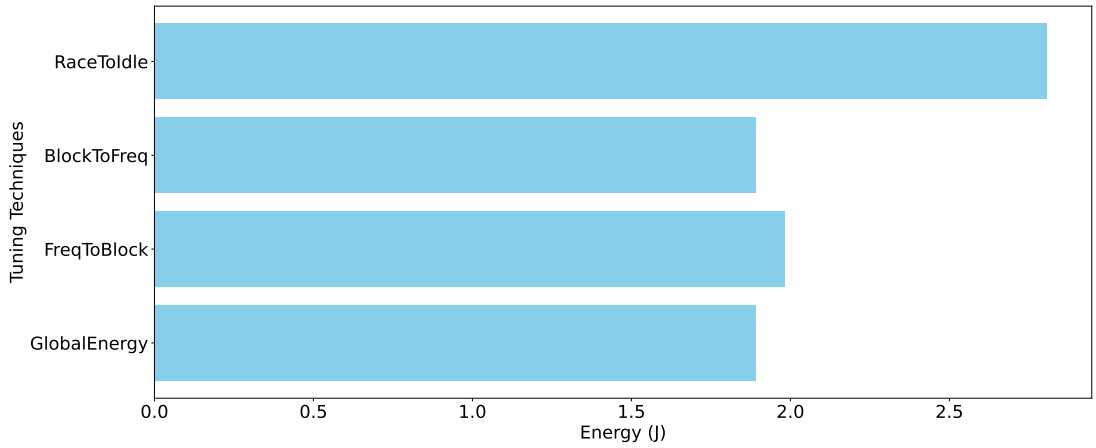

Figure 32: The four different tuning techniques with the total energy used.

4.7 Bandpass correction

To optimize the bandpass correction kernel, we use both block size x and block size y. We use the exact same method we used for the rgb2gray kernel in Section 4.5. We multiply block size x and block size y to get the total thread block size and take the best performing configurations for a thread block size of 32, 64, 128, 256, 512 and 1024 to provide a clearer comparison of their differences.

4.7.1 Comparison of thread block size and clock frequency

The bandpass correction kernel shows some remarkable results. Starting with the clock frequency, Figure 33 the same trend all other kernels have shown: a higher clock frequency consumes more power. Secondly, Figure 35 shows that the lowest clock frequency performs significantly worse than all other frequencies. The frequency 0.41 GHz shows already an improvement, being slightly worse than the other three clock frequencies. The energy efficiency of the clock frequencies is shown in Figure 37, which illustrates that a clock frequency of 0.41 GHz performs the best for all thread block sizes. The only exception is a thread block size of 32, where 0.70 GHz is the best option for clock frequency.

Moving on to the thread block sizes, Figure 34 shows that all thread block sizes consume almost the same amount of power, with a slight exception of 32, which consumes just a little more. Secondly, Figure 36 shows the performance of each thread block size. A notable aspect of Figure 36 is that each thread block size performs almost the same, except the 512 which is a little better and 32, whose performance increases significantly more than all other when we increase the clock

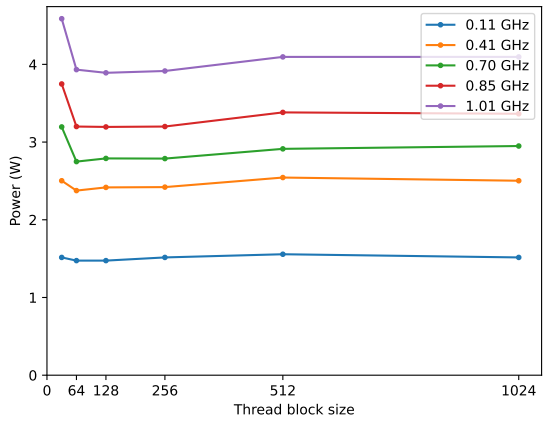

Figure 33: The power consumption of five different clock frequencies.

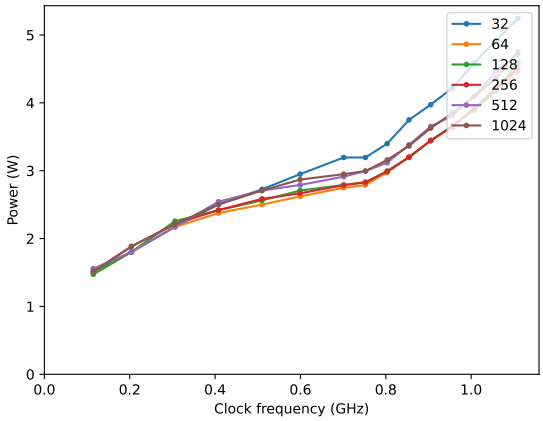

Figure 34: The power consumption of the six different thread block sizes.

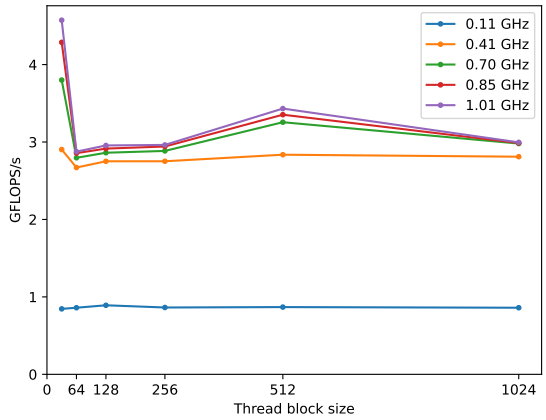

Figure 35: The execution time of five different clock frequencies.

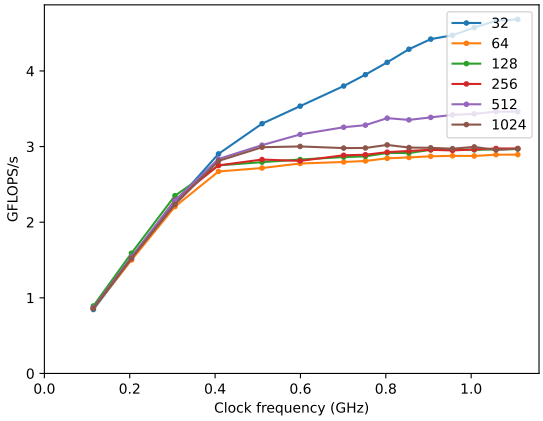

Figure 36: The execution time of the six different thread block sizes.

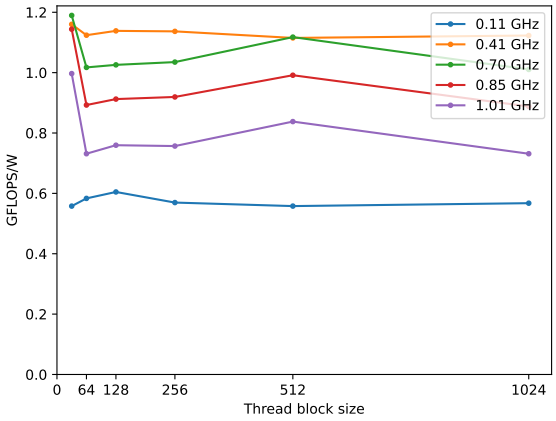

Figure 37: The energy efficiency of five different clock frequencies.

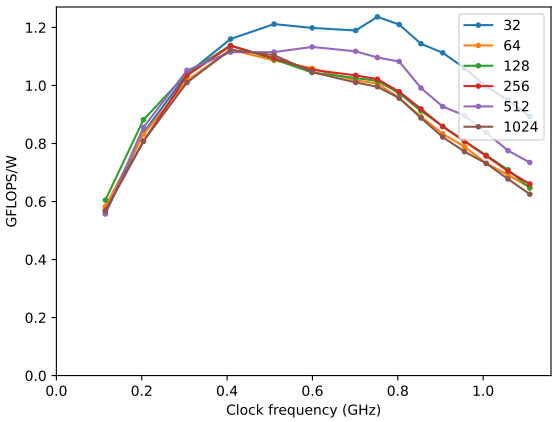

Figure 38: The energy efficiency of the six different thread block sizes.

frequency. The energy efficiency of the thread block sizes, shown in Figure 37, depicts that 32 is the performs the best in terms of energy efficiency, mostly due to its increasingly better performance.

Now that we have the results, we can explain why certain configurations are preferred above others. The reason behind the better energy efficiency for the middle range clock frequencies is described before in Section 4.4.1. In the bandpass correction kernel, we can see similarities to the raycasting kernel, while the bandpass correction kernel prefers a lower thread size too. The reason is the same for the bandpass correction kernel compared to the raycasting kernel, while the bandpass correction kernel contains conditional statements (for and if) as well.

4.7.2 Energy efficiency and performance

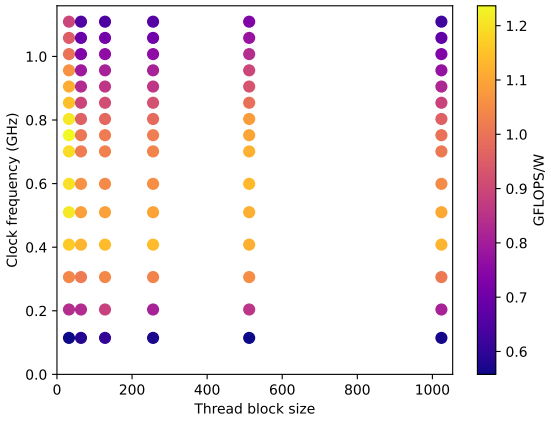

Figure 39: The energy efficiency for all combinations of thread block sizes and clock frequencies.

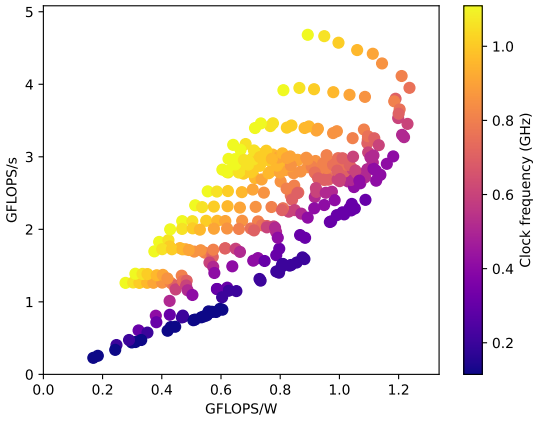

Figure 40: Relationship between performance and energy efficiency.

Figure 39 shows the relationship between performance and energy efficiency across all configurations for the bandpass correction kernel. Here we can see that all best configurations use a thread block size of 32, which we saw in Section 4.6 as well. The middle range of clock frequencies perform almost the same, as observed previously in Figure 37.

Figure 40 shows the relationship between performance and the energy efficiency for all possible configurations. Here we can see the trade-off between compute performance and energy efficiency of the fastest and most energy efficient configuration. We can see that the most energy efficient configuration is 38.47% more energy efficient and is only 15.60% slower than the fastest configuration. This is a significant result, while we can increase the energy efficiency and with a minimal reduction in performance.

4.7.3 Tuning strategies

Figure 41 shows the result of the four different tuning strategies, discussed in Section 4.1. As with all the other kernels, we can again see that RaceToIdle consumes the most power. For the bandpass correction kernel, RaceToIdle consumed 38.64%/0.017J more energy than GlobalEnergy. As for

the other techniques, we see a similarity compared to the raycasting kernel. Figure 41 shows that FreqToBlock consumes more power than both BlockToFreq and GlobalEnergy.

By first examining FreqToBlock, we can understand the difference between FreqToBlock and BlockToFreq. FreqToBlock initially optimizes for the clock frequency, determining that 0.408 GHz is the best option in combination with a thread block size of 256. Thereafter, it optimizes the thread block size, while keeping the clock frequency constant at 0.408 GHz. Discovering that a thread block size of 32 is the best option. This specific configuration consumes a total of 0.046J. BlockToFreq first discovers that a thread block size32 is the best option as well. However, when optimizing the clock frequency, it discovers that not 0.408 GHz is the best option, but 0.75225 GHz. This configuration only consumes 0.44J. So FreqToBlock consumes 4.55% more energy opposed to BlockToFreq. GlobalEnergy give the same configuration as BlockToFreq, showing that BlockToFreq finds the optimal configuration for energy efficiency.

There are 21 different combinations of block size x and block size y, resulting in total thread block sizes ranging from 32 to 1024. To determine the best configuration using GlobalEnergy, we need to test all 315 (15 * 21) possible configurations. However, for both FreqToBlock and BlockToFreq, there are only 36 possible configurations (15 + 21). This represents a significant reduction of 88.57% in possible configurations. Using BlockToFreq, we can significantly reduce the time to tune the bandpass correction kernel, while BlockToFreq technique finds the best configuration as well.

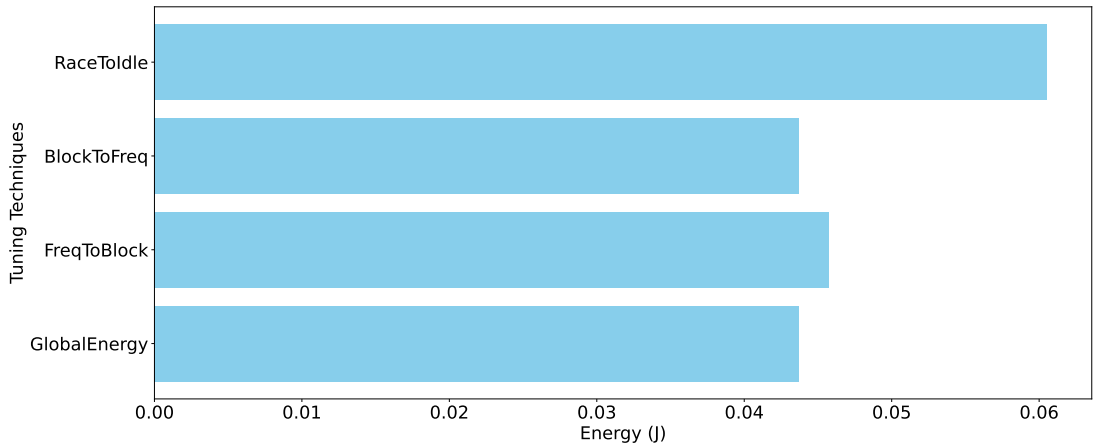

Figure 41: The four different tuning techniques with the total energy used.

4.8 Correlator

For the correlator kernel, there are only 4 different possible thread block sizes possible. The possible thread block sizes are 128, 192, 448 and 832. The thread block size is determined by computing the number of stations per thread, as explained in Section 4.2. The resulting valid configurations are:

- 1 thread per station with a block size x of 128.
- 2 threads per station with a block size x of 192.

- 3 threads per station with a block size x of 448.
- 4 threads per station with a block size x of 832.

4.8.1 Comparison of thread block size and clock frequency

On to the last kernel, Figure 42 shows that for the correlator kernel, as was the case with all other kernels, a higher clock frequency consumes more power. Figure 44 illustrates that a higher clock frequency performs better, but the total difference between frequencies decreases. These findings are summarized in Figure 46, where can see that in all cases, a clock frequency of 0.70 GHz has the best energy efficiency.

Looking at the difference in thread block size, Figure 43 shows that a higher thread block size uses more power. Secondly, Figure 45 shows that almost all thread block sizes perform the same, with only minimal differences. In Figure 47, we can see that in all cases a thread block size of 448 has the best energy efficiency.

Now that we have the results, we can explain why certain configurations are preferred above other. The reason behind the better energy efficiency is described before in Section 4.4.1. In the correlator kernel, we can see similarities to the raycasting kernel, while the bandpass correction kernel prefers a lower thread size as well. The reason is the same for the correlator kernel compared to the raycasting kernel, while the correlator kernel contains conditional statements (for and if) as well.

4.8.2 Energy efficiency and performance

Figure 48 shows the energy efficiency for all configurations for the correlator kernel. Figure 48 shows that all the best configurations for energy efficiency are in the middle of the scatterplot, meaning configuration with a thread block size of 192 or 448 and where the clock frequency falls in the middle of the range perform the best in terms of energy efficiency.

Figure 49 shows the relationship between the performance and the energy efficiency for all possible configurations. Here we can see the trade-off between compute performance and energy efficiency of the fastest and most energy efficient configuration. We can see that the most energy efficient configuration is 46.8% more energy efficient, but is 32.44% slower than the fastest configuration. This means that we can almost double the energy efficiency, at the cost of a substantial reduction in performance.

4.8.3 Tuning strategies

Figure 50 shows the result of the four different tuning strategies, discussed in Section 4.1. This figure shows that RaceToIdle uses the most amount of energy, 46.84%/0.27J more than GlobalEnergy. Again, we can see that BlockToFreq, FreqToBlock and GlobalEnergy all use the same amount of energy.

Figure 46 shows that a clock frequency of 0.70125 GHz always performs the best in the case of energy efficiency, so this clock frequency will always be chosen when first optimizing for clock frequency. Secondly, in Figure 47 we can see that a thread block size of 448 is always the best option in case of energy efficiency. Combining both findings, no matter if we first optimize for thread block size or clock frequency, we will always and up with the configuration of a thread

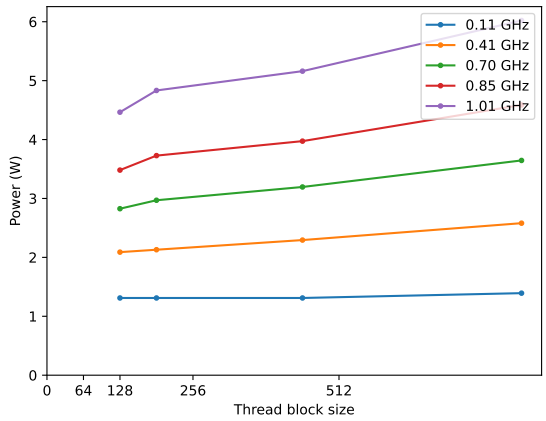

Figure 42: The power consumption of five different clock frequencies.

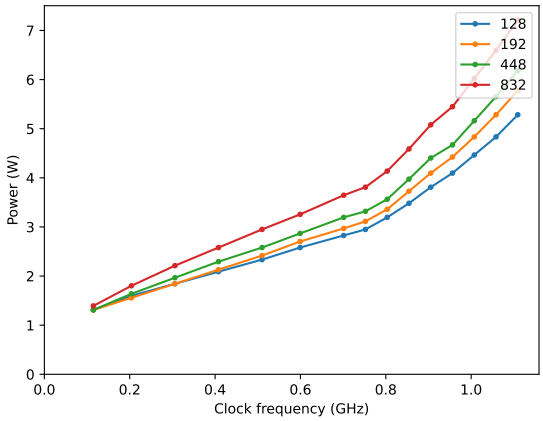

Figure 43: The power consumption of the four different thread block sizes.

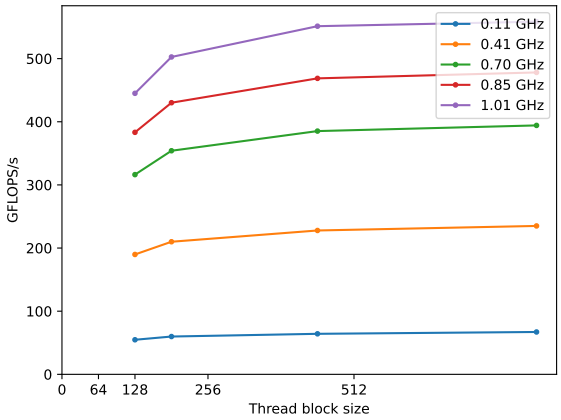

Figure 44: The execution time of five different clock frequencies.

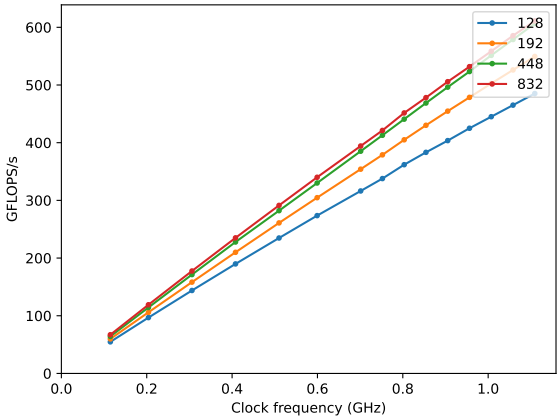

Figure 45: The execution time of the four different thread block sizes.

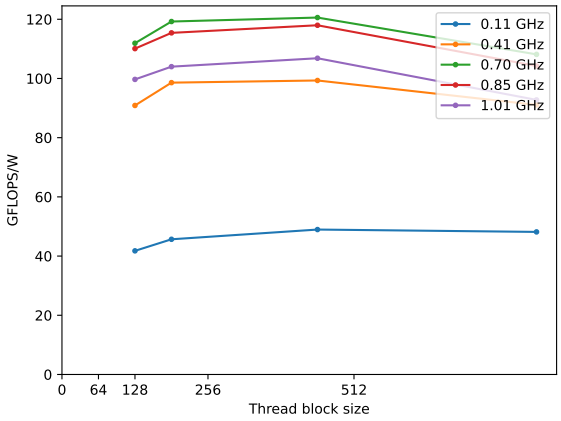

Figure 46: The energy efficiency of five different clock frequencies.

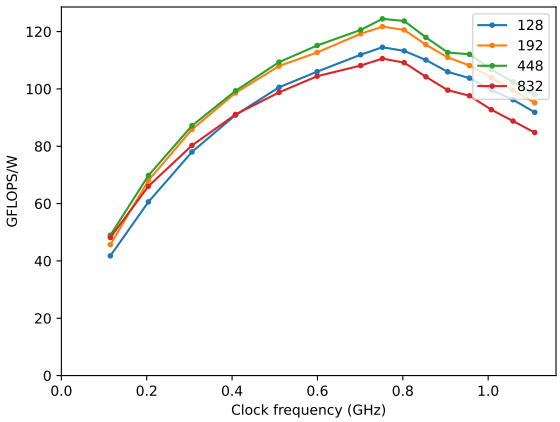

Figure 47: The energy efficiency of the four different thread block sizes.

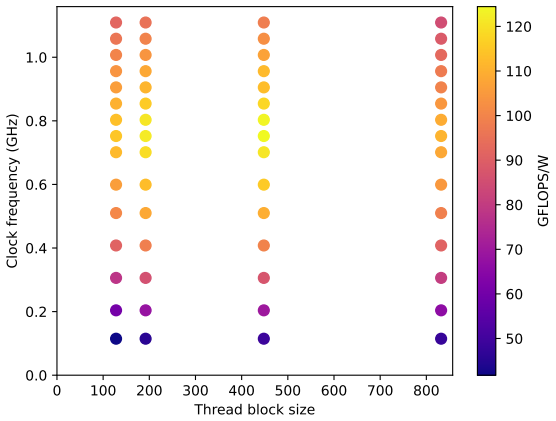

Figure 48: The energy efficiency for all combinations of thread block sizes and clock frequencies.

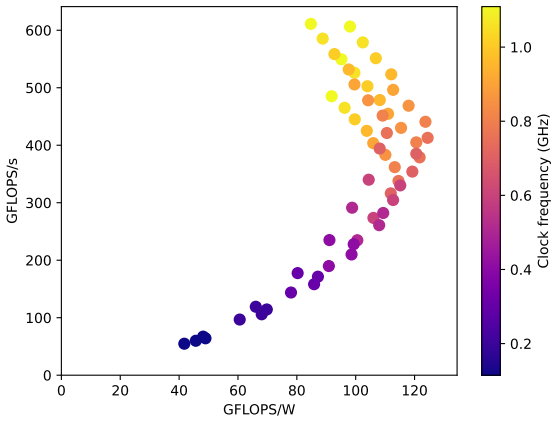

Figure 49: Relationship between performance and energy efficiency.

block size of 448 and a clock frequency of 0.70125 GHz. Figure 47 confirms these findings, while BlockToFreq, FreqToBlock and GlobalEnergy all have the same total energy consumption.

There are 4 different thread block sizes and 15 different clock frequencies, which results in a total of 60 possible configurations. To determine the best configuration using GlobalEnergy, we need to test all 60 possible configurations. However, for both FreqToBlock and BlockToFreq, there are only 19 possible configurations. This represents a significant reduction of 68.33% in possible configurations. Knowing that both BlockToFreq and FreqToBlock return the same configuration as GlobalEnergy, we can use either one to speed up the tuning process.

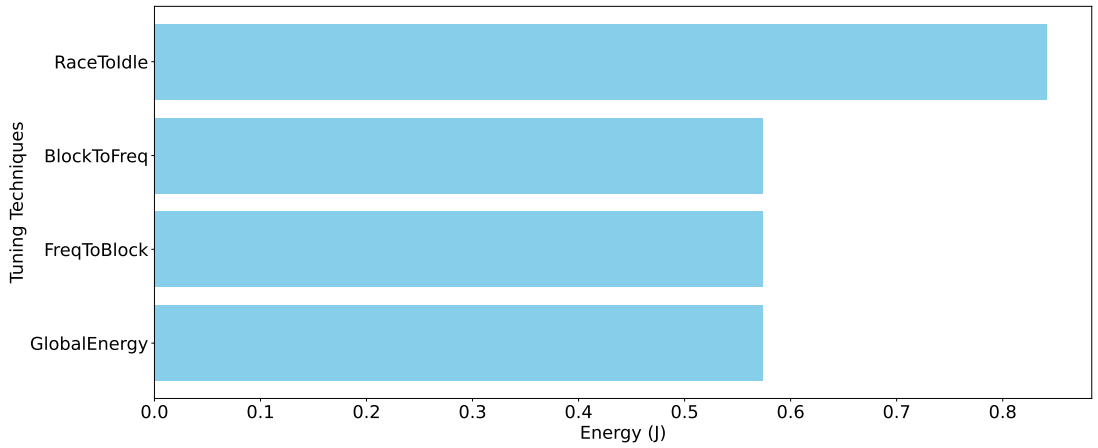

Figure 50: The four different tuning techniques with the total energy used.

5 Discussion

In the previous sections, we gathered and discussed the findings of the various benchmarks individually. This section will discuss and compare the finding of all benchmarks.

5.1 Clock frequencies

Starting with the first parameter, the clock frequency. All five benchmarks have shown a similarity in the power consumption of different clock frequencies; higher clock frequencies consume more power. Another similarity is seen in the performance, while higher frequencies tend to have better performance in terms of GFLOPS/s. However, all benchmarks have shown that despite a better performance, the middle range of clock frequencies is better in terms of energy efficiency. The total energy used is given by the formula $E = P * t$, where E is the energy in Joules, P the power in Watt and t the execution time in seconds. Figure 51, Figure 52 and Figure 53 illustrate the amount of power, time and energy the lowest, middle and highest clock frequency use for the matrix multiplication kernel. Firstly, in Figure 51 we can see that the highest frequency uses by far the most energy, the lowest frequency uses the least amount of energy and the middle frequency sits in between. However, Figure 52 shows the most significant difference. The lowest frequency is more than 5 times slower than the middle and higher frequencies, while only using 4 times less energy than the other frequencies, resulting in the highest energy consumption. On the other hand, the highest is about as fast as the middle frequency, while using a lot more power. This explains why the middle frequency is the most energy efficient.

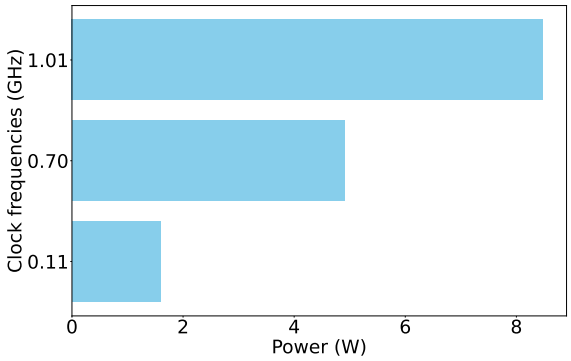

Figure 51: Power consumption of the highest, middle and lowest clock frequencies.

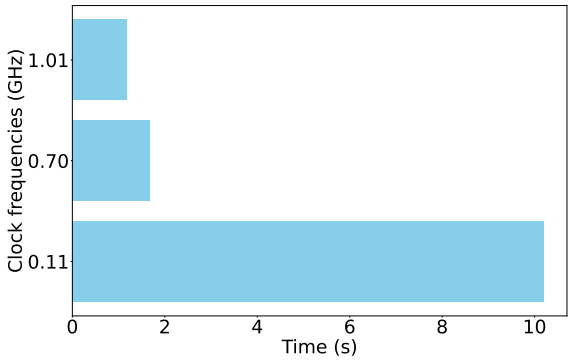

Figure 52: Execution time of the highest, middle and lowest clock frequencies.

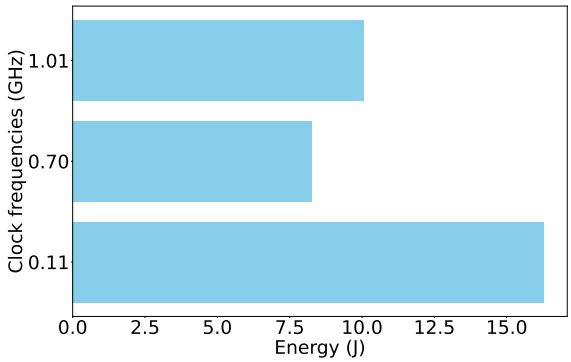

Figure 53: The energy consumption of the highest, middle and lowest clock frequencies.

5.2 Thread block size

Analyzing the energy efficiency of different thread block sizes, we observed that the results varied between the kernels. The matrix multiplication and rgb2gray kernels showed a high thread block size as the best option for energy efficiency, while the other kernels preferred a middle to low thread block size. Matrix multiplication and rgb2gray are kernels which are highly parallelizable. Both can calculate and store independent elements, which means that each thread can operate on separate data without waiting for others. High thread block sizes allow the streaming multiprocessors (SMs) to be fully utilized. The other kernels preferred smaller thread block sizes. The reason why, explained earlier in Section 4.6.1, lies in the fact that these kernels can have varying workloads. Unlike the matrix multiplication and rgb2gray kernel these kernels contain conditional statements such as if, for and/or while loops. Conditional statements can cause thread divergence, meaning that threads will follow different execution paths. When thread divergence happens, the GPU will serialize the execution, meaning that only one path is executed at a time, which will cause the threads which do not follow that path to remain idle [ZJGS10]. Using smaller thread block sizes, we can localize and mitigate these imbalances while fewer threads remain inactive while waiting for other threads to finish. Using smaller thread block sizes improve utilization of the SM and less inactive threads,

which lowers the overall execution time and increases the performance and energy efficiency.

5.3 Energy efficiency and performance

On to the energy discussion of the overall efficiency and performance. Each kernel has showed that an improvement in energy efficiency comes at the cost of reduced performance. However, the ratio of gain in energy efficiency and decrease in performance was not the same for all kernels. The rgb2gray, raycasting and bandpass correction kernel showed a remarkable improvement in energy efficiency with only a small reduction in performance. However, this was not the case for the matrix multiplication and correlator kernel. Both needed to decrease their performance by the same proportion that they gain in energy efficiency. First of all, both the matrix multiplication kernel and the correlation kernel were only tested with 60 different configurations, while e.g. the rgb2gray kernel was tested with 720 configurations. Due to the difference in number of tested configurations, rgb2gray had more opportunities to find a better configuration. Secondly, in Figure 8 (matrix multiplication) and Figure 44 (correlator) we observe that the frequency optimized for energy efficiency performs nearly 30% worse than the frequency optimized for optimal performance. Figure 17 (rgb2gray), Figure 26 (raycasting) and Figure 35 (bandpass correction) showed a much smaller performance difference between the two frequencies.

5.4 Tuning strategies

Lastly, we will discuss the results of the tuning strategies. Across the various benchmarks, we have seen interesting results. Firstly, RaceToIdle consistently showed the highest energy consumption across all benchmarks. On average, RaceToIdle consumed 43.67% more energy compared to GlobalEnergy, highlighting the significant increase in energy efficiency we can achieve when tuning for GlobalEnergy compared to RaceToIdle. Secondly, variations were observed between GlobalEnergy, BlockToFreq and FreqToBlock. While matrix multiplication and correlator kernels showed consistent configurations and energy consumption across all three strategies, the rgb2gray, raycasting and bandpass correction kernel showed some differences between GlobalEnergy and either BlockToFreq or FreqToBlock. In these kernels, either BlockToFreq or FreqToBlock failed to find the optimal configuration for energy efficiency. However, the maximal increase in energy consumption using either BlockToFreq or FreqToBlock was only 4.55% compared to GlobalEnergy. Conversely, both BlockToFreq and FreqToBlock demonstrated on average a 78.30% decrease in possible configuration. While BlockToFreq or FreqToBlock showed a slight increase in energy consumption, this is only minimal compared to the reduction in possible configuration and the time required to tune a kernel.

6 Conclusion and further research

In this thesis we extended Kernel Tuner to enable energy measurements and clock frequency tuning on Nvidia Jetson devices. The objective of this extension was to make Kernel Tuner able to optimize applications on Nvidia Jetson devices for energy efficiency. To evaluate the effectiveness of the extension, we performed experiments with five different kernels. In these benchmarks, we focused primarily on tuning the clock frequency and thread block size of the GPU.

In these benchmarks we discovered that each kernel had a unique configuration to maximize the energy efficiency. This shows once again the importance of tuning each kernel separately. We discovered that the best clock frequencies for all fell within the middle range of possible clock frequencies. This is interesting, because neither a high nor low clock frequency maximizes the energy efficiency, but rather a more balanced frequency. For the thread block size, the benchmarks indicated that for each kernel, the optimal thread block size varied between all five. Highly parallelizable kernels prefer higher thread block sizes, whereas kernels with potential workload imbalances among threads, due to e.g. conditional statements, prefer lower thread block sizes.

The extension to Kernel Tuner and the finding of this thesis can be beneficial to users and researchers of the Nvidia Jetson series. By using this research, developers can optimize applications on Nvidia Jetson devices for energy efficiency and prolong the battery life of these devices.

There are numerous possible research directions to further improve the energy efficiency of Nvidia Jetson devices. Possible further research could involve optimizing memory bandwidth. This could reduce power consumption by improving the efficiency of data transfer. Another possibility is to study and implement Dynamic Voltage and Frequency Scaling (DVFS), which could reduce power consumption by adjusting voltage levels and core frequencies dynamically based on workload demands.

References

[AYM22] Büşra Aslan and Ayse Yilmazer-Metin. A study on power and energy measurement of nvidia jetson embedded gpus using built-in sensor. In *2022 7th International Conference on Computer Science and Engineering (UBMK)*, pages 1–6. IEEE, 2022.

[BCVMFB20] Théo Benoit-Cattin, Delia Velasco-Montero, and Jorge Fernández-Berni. Impact of thermal throttling on long-term visual inference in a cpu-based edge device. *Electronics*, 9(12):2106, 2020.

[BDG+18] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106(11):2068–2083, November 2018.

[BIM16] Robert A Bridges, Neena Imam, and Tiffany M Mintz. Understanding gpu power: A survey of profiling, modeling, and simulation methods. *ACM Computing Surveys (CSUR)*, 49(3):1–27, 2016.

[BPD21] David Bol, Thibault Pirson, and Rémi Dekimpe. Moore's law and ict innovation in the anthropocene. In *2021 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 19–24, 2021.

[BZZ14] Martin Burtscher, Ivan Zecena, and Ziliang Zong. Measuring gpu power with the k20 built-in sensor. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, pages 28–36, 2014.

[CLMS20] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE access*, 8:85714–85728, 2020.

[CLZ+11] Jianmin Chen, Bin Li, Ying Zhang, Lu Peng, and Jih-kwon Peir. Statistical gpu power analysis using tree-based methods. In *2011 International Green Computing Conference and Workshops*, pages 1–6. IEEE, 2011.

[DGDD+19] F De Gasperin, TJ Dijkema, A Drabent, M Mevius, D Rafferty, R Van Weeren, M Brüggen, JR Callingham, KL Emig, G Heald, et al. Systematic effects in lofar data: A unified calibration strategy. *Astronomy & Astrophysics*, 622:A5, 2019.

[FCX+17] Yuling Fang, Qingkui Chen, Neal N Xiong, Deyu Zhao, and Jingjuan Wang. Rgca: a reliable gpu cluster architecture for large-scale internet of things computing based on effective performance-energy optimization. *Sensors*, 17(8):1799, 2017.

[FYK+17] Mariza Ferro, André Yokoyama, Vinicius Klôh, Gabrieli Silva, Rodrigo Gandra, Ricardo Bragança, Andre Bulcao, and Bruno Schulze. Analysis of gpu power consumption using internal sensors. In *Anais do XVI Workshop em Desempenho de Sistemas Computacionais e de Comunicação*. SBC, 2017.

[Gee05] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[GGXS+12] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 innovative parallel computing (InPar)*, pages 1–10. Ieee, 2012.

[GPKB12] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. Gpgpu processing in cuda architecture. *arXiv preprint arXiv:1202.4347*, 2012.

[GVM+13] Rong Ge, Ryan Vogt, Jahangir Majumder, Arif Alam, Martin Burtscher, and Ziliang Zong. Effects of dynamic voltage and frequency scaling on a k20 gpu. In *2013 42nd International Conference on Parallel Processing*, pages 826–833. IEEE, 2013.

[HCDP19] Sam Hatfield, Matthew Chantry, Peter Düben, and Tim Palmer. Accelerating high-resolution weather models with deep-learning hardware. In *Proceedings of the platform for advanced scientific computing conference*, pages 1–11, 2019.

[HGS19] Yanhui Huang, Bing Guo, and Yan Shen. Gpu energy consumption optimization with a global-based neural network method. *IEEE Access*, 7:64303–64314, 2019.

[HK10] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 280–289, 2010.

[ISAHA22] Muhammad Jawad Ikram, Mostafa Elsayed Saleh, Muhammad Abdulhamid Al-Hashimi, and Osama Ahmed Abulnaja. Investigating the effect of varying block size on power and energy consumption of gpu kernels. *The Journal of Supercomputing*, 78(13):14919–14939, 2022.

[JMSN05] Hailin Jiang, Malgorzata Marek-Sadowska, and Sani R Nassif. Benefits and costs of power-gating technique. In *2005 International conference on computer design*, pages 559–566. IEEE, 2005.

[JN16] Lennart Johnsson and Gilbert Netzer. The impact of moore's law and loss of dennard scaling: Are dsp socs an energy efficient alternative to x86 socs? In *Journal of Physics: Conference Series*, volume 762, page 012022. IOP Publishing, 2016.

[JOL+23] Mathilde Jay, Vladimir Ostapenco, Laurent Lefèvre, Denis Trystram, Anne-Cécile Orgerie, and Benjamin Fichel. An experimental comparison of software-based power meters: focus on cpu and gpu. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 106–118. IEEE, 2023.

[KC12] Christopher Kanan and Garrison W Cottrell. Color-to-grayscale: does the method matter in image recognition? *PloS one*, 7(1):e29740, 2012.

[KTg] Kernel tuner github. https://github.com/KernelTuner/kernel_tuner. Accessed: 2024-07-19.

[LLCP20] Sanghoon Lee, Dongkyu Lee, Pyung Choi, and Daejin Park. Accuracy–power controllable lidar sensor system with 3d object recognition for autonomous vehicle. *Sensors*, 20(19):5706, 2020.

[LZR+19] Pei Li, Shihao Zhou, Bingqing Ren, Shuman Tang, Ting Li, Chang Xu, and Jiageng Chen. Efficient implementation of lightweight block ciphers on volta and pascal architecture. *Journal of Information Security and Applications*, 47:235–245, 2019.

[Mit19] Sparsh Mittal. A survey on optimized implementation of deep learning models on the nvidia jetson platform. *Journal of Systems Architecture*, 97:428–442, 2019.

[MK15] Ashish Mishra and Nilay Khare. Analysis of dvfs techniques for improving the gpu energy efficiency. *Open Journal of Energy Efficiency*, 4(4):77–86, 2015.

[MV14] Sparsh Mittal and Jeffrey S Vetter. A survey of methods for analyzing and improving gpu energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):1–23, 2014.

[Obs] Observers - kernel tuner 1.0 documentation. https://kerneltuner.github.io/kernel_tuner/stable/observers.html. Accessed: 2024-05-20.

[OHL+08] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[PFG+22] Mohit Pandey, Michael Fernandez, Francesco Gentile, Olexandr Isayev, Alexander Tropsha, Abraham C Stern, and Artem Cherkasov. The transformational role of gpu computing and deep learning in drug discovery. *Nature Machine Intelligence*, 4(3):211–221, 2022.

[RBMvN10] John W. Romein, P. Chris Broekema, Jan David Mol, and Rob V. van Nieuwpoort. The lofar correlator: implementation and performance analysis. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 169–178, New York, NY, USA, 2010. Association for Computing Machinery.

[S+09] Reiji Suda et al. Accurate measurements and precise modeling of power dissipation of cuda kernels toward power optimized high performance cpu-gpu computing. In *2009 international conference on parallel and distributed computing, applications and technologies*, pages 432–438. IEEE, 2009.

[Sch97] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[SDŞ20] Ahmet Ali Süzen, Burhan Duman, and Betül Şen. Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn. In *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, pages 1–5. IEEE, 2020.

[SLS+20] Sergio Márquez Sánchez, Francisco Lecumberri, Vishwani Sati, Ashish Arora, Niloufar Shoeibi, Sara Rodríguez, and Juan M Corchado Rodríguez. Edge computing driven smart personal protective system deployed on nvidia jetson and integrated with ros. In *Highlights in Practical Applications of Agents, Multi-Agent Systems, and Trust-worthiness. The PAAMS Collection: International Workshops of PAAMS 2020, L'Aquila, Italy, October 7–9, 2020, Proceedings 18*, pages 385–393. Springer, 2020.

[spe] Nvidia jetson xavier. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/. Accessed: 2024-05-23.

[Sto10] Andrew James Stothers. On the complexity of matrix multiplication. 2010.

[SVVWB22] Richard Schoonhoven, Bram Veenboer, Ben Van Werkhoven, and K Joost Batenburg. Going green: optimizing gpus for energy efficiency through model-steered auto-tuning. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 48–59. IEEE, 2022.

[SvWB22] Richard Arnoud Schoonhoven, Ben van Werkhoven, and Kees Joost Batenburg. Benchmarking optimization algorithms for auto-tuning gpu kernels. *IEEE Transactions on Evolutionary Computation*, 27(3):550–564, 2022.

[UAR+21] Md Imran Uddin, Md Shahriar Alamgir, Md Mahabubur Rahman, Muhammed Shahnewaz Bhuiyan, and Md Asif Moral. Ai traffic control system based on deepstream and iot using nvidia jetson nano. In *2021 2nd International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST)*, pages 115–119. IEEE, 2021.

[vNR10] Rob van Nieuwpoort and John W Romein. Building correlators with many-core hardware. *IEEE Signal Processing Magazine*, 27(2):108–117, 2010.

[WPW00] Qing Wu, Massound Pedram, and Xunwei Wu. Clock-gating and its application to low power design of sequential circuits. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(3):415–420, 2000.

[WWR+16] Bartosz D Wozniak, Freddie D Witherden, Francis P Russell, Peter E Vincent, and Paul HJ Kelly. Gimmik—generating bespoke matrix multiplication kernels for accelerators: Application to high-order computational fluid dynamics. *Computer Physics Communications*, 202:12–22, 2016.

[YYAS11] W. K. Yeo, David F. W. Yap, D. P. Andito, and M. K. Suaidi. Grayscale mri image compression using feedforward neural networks. In *7th International Conference on Broadband Communications and Biomedical Applications*, pages 39–42, 2011.

[ZJGS10] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining gpu applications on the fly: thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 115–126, 2010.

[ZLT+19] Hadi Zamani, Yuanlai Liu, Devashree Tripathy, Laxmi Bhuyan, and Zizhong Chen. Greenmm: energy efficient gpu matrix multiplication through undervolting. In *Proceedings of the ACM International Conference on Supercomputing*, pages 308–318, 2019.

A Tune example

```
dimensions = 4096
problem_size = (dimensions, dimensions)
size = numpy.prod(problem_size)
A = numpy.random.randn(*problem_size).astype(numpy.float32)
B = numpy.random.randn(*problem_size).astype(numpy.float32)
C = numpy.zeros_like(A)
n = numpy.int32(dimensions)
args = [C, A, B, n]

tegraObserver = TegraObserver(["gpu_freq", "gpu_temp", "gpu_power", "
                                        gpu_energy"])

tune_params = dict()
tune_params["tegra_gr_clock"] = list(tegraobserver.tegra.
                                        supported_gr_clocks)
tune_params["block_size_x"] = [32, 24, 16, 8]
tune_params["block_size_y"] = [32, 24, 16, 8]
restrict = ["block_size_x==block_size_y"]

metrics = dict()
metrics["GFLOPS/W"] = lambda p: (((2*(total**3)) / 1e9) / (p["time"] /
                                        1e3)) / p["gpu_power"]

result, env = tune_kernel(kernel_name="matmul_kernel", kernel_source=
                                        matmul_kernel,
problem_size=problem_size, arguments=args, tune_params=tune_params,
                                        observers=[tegraObserver],
                                        metrics=metrics, restrictions=
                                        rest, objective="gpu_temp",
                                        objective_higher_is_better=True
                                        )
```

Figure 54: Python code to start the tuning process.

B Temperature code

```
    def get_temp_path(self):
        """Find the file which holds the GPU temperature"""
        for zone in Path("/sys/class/thermal").iterdir():
            with open(zone / Path("type")) as fp:
                name = fp.read().strip()
            if name == "GPU-therm":
                gpu_temp_path = zone + "/"
                break
        else:
            raise FileNotFoundError("No GPU sensor for temperature found")

        return gpu_temp_path
```

Figure 55: Python code to find the file which hold the information of the GPU temperature.

```
    def read_gpu_temp(self):
        """Read GPU temperature"""
        with open(self.gpu_temp_path + "temp") as fp:
            temp = int(fp.read())
        return temp / 1000
```

Figure 56: Python code to read the temperature on Nvidia Jetson devices.

C Power code

```
    def get_power_path(self, start_path="/sys/bus/i2c/drivers/ina3221"):
        """Recursively search for a file which holds power readings
        starting from start_path."""
        for entry in os.listdir(start_path):
            path = os.path.join(start_path, entry)
            if os.path.isfile(path) and entry == "curr1_input":
                return start_path + "/"
            elif entry in start_path:
                continue
            elif os.path.isdir(path):
                result = self.get_power_path(path)
                if result:
                    return result
        return None
```

Figure 57: Python code to find the directory that holds the power readings of all channels.

```
    def get_gpu_channel(self):
        """Get the channel number of the sensor which measures the GPU power"""

        # Iterate over all channels in the of_node dir of the power path
        # to find the channel which holds GPU power information
        for channel_dir in Path(self.gpu_power_path + "of_node/").iterdir():
            if("channel@" in channel_dir.name):
                with open(channel_dir / Path("label")) as fp:
                    channel_label = fp.read().strip()
                if "GPU" in channel_label:
                    return str(int(channel_dir.name[-1])+1)

        # If this statement is reached, no channel for the GPU was found
        raise FileNotFoundError("No channel found with GPU power readings")
```

Figure 58: Python code to find the channel that holds the power readings of the GPU.

```
    def read_gpu_power(self):
        """Read the current and voltage to calculate and return the power in
                                                Watt"""
        result_cur = subprocess.run(["sudo", "cat", f"{self.gpu_energy_path}
                                                curr{self.gpu_channel}_input"],
                                                 capture_output=True, text=True
                                                )
        current = int(result_cur.stdout.strip()) / 1000
        result_vol = subprocess.run(["sudo", "cat", f"{self.gpu_energy_path}in{
                                                self.gpu_channel}_input"],
                                                capture_output=True, text=True)
        voltage = int(result_vol.stdout.strip()) / 1000

        return current * voltage
```

Figure 59: Python code to read the current and voltage of the GPU.