Bachelor Informatica

University of Amsterdam

Energy-harvesting policies for heterogeneous computing

Nick Breed

June 18, 2021

Supervisor(s): Dr. ir. Ana-Lucia Varbanescu

Abstract

Graphics processing units (GPUs) are used in every day desktops and laptops, but also for commercial or research purposes in massive supercomputers. As such, the performance of GPUs is constantly improving. Thus, more workloads aim to improve their performance using GPUs. This also means we observe an increase in workloads that use both GPUs and CPUs (the "standard" processors), the so-called heterogeneous workloads. Performance is of course important for these heterogeneous workloads, but their energy impact has also become a concern.

In this project we focus on *energy-harvesting* for GPU-CPU heterogeneous workloads. Specifically, we investigate ways to reduce the energy cost of the workloads without affecting their performance. This research builds upon the master thesis of Bakker [1], which has provided an in-depth analysis framework of the energy consumption of heterogeneous systems. The work from Bakker [1] includes two simplistic policies and three workloads. In this thesis we propose 3 new policies, and add 7 more heterogeneous workloads. Using our single-core policies, we observed energy savings up to 3.5%, with the worst-case scenario showing an increase of energy consumption of 23.5%. For execution time of single-core policies, we observed speedups up to 9.4%, with the worst-case scenario showing a delay of 8.4%. For the energy of multi-core policies, we observed savings up to 34.8%, with the worst-case scenario showing an increase of energy consumption of 27.6%. For execution time of multi-core policies, we observed speedups up to 10%, with the worst-case scenario showing a delay of 27.1%. For some applications, the newly introduced policies are a better solution for energy harvesting.

Contents

CHAPTER 1

Introduction

Graphics Processing Units (GPUs) are used in both every day desktops and laptops (e.g., NVIDIA GeForce RTX 30-series [14] is such a ”consumer card”), but also for commercial or research purposes, in large clusters and supercomputers in the TOP500 [18] (e.g., the NVIDIA Volta GV100 GPU is a ”datacenter” or ”high-performance” card). For both markets, the performance of GPUs is constantly improving. So, more workloads aim to improve their performance using GPUs. This also means we observe an increase in popularity of workloads that use both GPUs and CPUs (the ”standard” processors) [10]. We call such workloads, that use both GPUs and CPUs, heterogeneous.

Performance is of course important for these heterogeneous workloads, but costing less energy has also become important, especially when it does not hurt performance. A lower energy cost for a heterogeneous system also means that it is cheaper to run the system. Moreover, using less energy for the same performance means less unnecessary energy consumption, which is important for our environment. Although GPUs are energy-efficient compared to CPUs, CPU-GPU heterogeneous workloads consume more energy, because of the addition of the GPU. Our research investigates automatically reducing the energy consumption of heterogeneous workloads, at runtime, without user intervention.

Existing research studying the energy consumption of the heterogeneous workloads focuses mostly on modelling the consumption using behavioural patterns of the application and performance variables [3, 8, 6, 11]. Only a couple of papers target the reduction of the energy-consumption dynamically [7, 9], and they focus either on power capping or on GPU-specific analysis.

In this project we research ways to reduce the energy cost of GPU-CPU heterogeneous workloads dynamically, without affecting their performance. This research will build upon the master thesis of Bakker [1], which has provided an in-depth analysis framework of the energy consumption of heterogeneous systems. We use the same framework to analyse many more applications, and we propose new energy saving policies to tackle the diversity of these applications.

1.1 Research Question and Approach

The main research question for this project is:
What policies can improve energy-harvesting for heterogeneous workloads?

The work from Bakker [1] includes two simplistic policies and three workloads. The focus of this thesis is to propose new policies, and to validate them on more heterogeneous workloads. Thus, this research can be split up in three parts:

1. Find new heterogeneous workloads.

2. Design new energy-harvesting policies.
3. Evaluate the energy-harvesting policies for all proposed workloads.

The research to find candidate workloads starts with a literature study. To be eligible, these workloads have to be diverse. Specifically, they need to have a different state pattern according to the classification framework (see Chapter 4). Such diversity ensures that the proposed harvesting policies are thoroughly tested, against a wide spectrum of workloads. We extract these workloads from existing benchmarking suites.

The research to design other energy-harvesting policies also starts with a literature study. For the design and finding of new policies we should research heterogeneous computing, to fully understand the problem. Further, we investigate energy-saving policies that make use of Dynamic Voltage and Frequency Scaling (DVFS) on heterogeneous workloads. Finally, we design new policies by combining workloads characteristics with existing DVFS policies.

To answer the main research question, it is also important to know if the policies actually reduce the energy consumption for the selected workloads. For this phase, we perform an empirical study, where we use the framework to implement promising policies, and we further test those policies on the candidate workloads. Finally, based on these empirical results, we can provide guidelines to match policies to workloads/workload types.

So, the final goal of the research is to use the application characteristics, state patterns, and performance indicators that we obtain from the Heterogeneous Energy Harvesting framework to find new suitable energy harvesting policies for (new) heterogeneous workload applications. In the best case scenario, we aim to find one policy that is suitable for all candidate workloads; it is however more likely that we find a number of policies that are suitable for different kinds of workloads.

1.2 Ethical aspects

We find that there are no ethical concerns with the motivation and results of our the work. One potential ethical concern is that our work on building energy harvesting policies does rely on a significant amount of testing, which consumes energy. We try to build policies that are portable to many large, long-running applications, such that the saved energy consumption for these workloads can outweigh the energy consumption for the development and testing.

1.3 Outline

The thesis is organized as follows. We present background information and previous studies in Chapter 2. Chapter 4 presents the selected candidate workloads we will use for the design and analysis of our energy-harvesting policies. We present the design and implementation of our new energy-harvesting policies in Chapter 5. We will finally present the evaluation of the new policies in Chapter 5. Lastly, we present our conclusion and future work in Chapter 6.

CHAPTER 2

Theoretical background

By "energy-harvesting for computing", we mean minimizing the energy cost without decreasing performance. Heterogeneous computing refers to workloads that use at least two different kind of processors to run, e.g., workloads that use both a GPU and a CPU. In this thesis, we focus specifically on CPU + GPU systems and workloads.

2.1 Energy Manager Framework

Our research builds upon the master thesis of Bakker [1]. This thesis creates a number of frameworks we will use. These frameworks were established in collaboration with SURFsara and the Barcelona Supercomputing Center, as an extension on the Energy Aware Runtime (EAR) library to support CPU-GPU heterogeneous workloads.

The first framework we use is a *monitoring framework* that gathers data about energy consumption, frequency, runtime and events that happen in the workload. The gathered data is processed and visualized into graphs, so the data can be interpreted. The data can be used for testing the impact of changing CPU and GPU frequencies on both the energy consumption and runtime of a heterogeneous workload. With the data from this analysis, it is possible to determine whether energy saving policies that do not negatively impact the runtime are possible.

The second framework, the *workload classification framework*, can classify different applications with respect to their CPU-GPU usage. For example, it is possible to classify CPU-bound or GPU-bound applications. The classification framework focuses on detecting in which state the program is during its execution. The different states are: `IDLE`, `CPU_IDLE`, `GPU_IDLE`, `BUSY` and `BUSY_WAIT`. The `IDLE` state is that both the GPU and CPU have low utilized. The `CPU_IDLE` state has high GPU, but low CPU utilization. The `GPU_IDLE` state has low GPU, but high CPU utilization. The `BUSY` state has low utilization for both the GPU and CPU and there are not any active synchronisation calls. If the GPU and CPU have low utilization, but there is an active synchronisation call it is in the `BUSY_WAIT` state. High utilization means that the utilization rate is above the threshold of 10%. The classification framework has been proven correct on several case-studies [1].

The third framework, *the energy-harvesting framework*, implements the energy harvesting policies. This framework uses the combined functionality of the previous frameworks. The energy-harvesting policies are based on changing the processors' clock frequency to save energy, in a process called DVFS [16]. The current work only proposes two policies: the MinMax policy and System policy.
The MinMax policy always either minimizes or maximizes the CPU and GPU frequencies. This is done based on the five states detected in the workload classification. Specifically, the CPU-frequency is maximized for the states: `GPU_IDLE` and `BUSY`. In the other states the CPU-frequency

is minimized. The GPU-frequency is maximized for the states: `CPU_IDLE`, `BUSY` and `BUSY_WAIT`. In the other, states the GPU-frequency is minimized.
For the System policy the CPU and GPU frequencies will only be minimized or maximized when necessary. Thus, the CPU-frequency is unrestricted and controlled by the operating system for the states: `GPU_IDLE`, `BUSY` and `IDLE`. In the other states the CPU-frequency is minimized. The GPU-frequency is unrestricted and controlled by the operating system for the states: `CPU_IDLE`, `BUSY`, `BUSY_WAIT` and `IDLE`. In the other states the GPU-frequency is minimized. The frequency is only minimized when the CPU and GPU are in different utilization states. Out of these two policies, the MinMax policy performed better when there was energy saved; for one of the three heterogeneous workloads, neither policy performed better than the unrestricted versions of the application (this was the CPU-bound application).

2.2 Related Work

Energy saving is a very important topic in computer science.

For example, a lot of work is done in measuring and modeling the energy consumption of applications [6, 8], or predicting it [3, 11]. However, such studies limit themselves to observation, and do not propose solution to reduce this consumption. Our work focuses on proactively reducing the energy consumption of heterogeneous workloads.

Another large body of work focuses on energy efficiency, where platforms are compared for their ability to perform more work per watt. For example, Green500 [17], this is a list with the best ranking supercomputers in performance, ranked on power efficiency. Similarly, many studies focus on comparing CPUs and GPUs, showing how GPUs are very attractive in terms of work/watt. In most of these studies, authors propose the use of GPUs to reduce energy consumption of given applications. By comparison, our work proposes using the heterogeneity of the system to improve energy efficiency.

A lot of work on DVFS has already been proposed for CPU and GPU applications. For example the EAR library [2] shows how energy monitoring and reduction can be achieved in parallel and distributed CPU-based systems. Similarly, the paper from Mei et al. [9] shows a similar approach for GPU computing. However, this work targets the specifics of each platform, and does not consider the specific aspects of heterogeneous workloads.

DVFS for heterogeneous systems is also the subject of other research work. For example, Komoda et al. [7] also focuses on CPU-GPU heterogeneous systems, and on the fact that reducing frequency for energy-harvesting purposes can lead to comparable performance. Another example is from Mei et al. [9], where the authors focuses on DVFS for GPUs for energy-harvesting. They get an average of 19.28% reduction in energy while only having a loss in performance of 4%. In both papers, the authors agree on the fact that DVFS is an useful mechanism to reduce energy consumption.

In summary, our work focuses on energy-harvesting in heterogeneous workloads, which is a restricted, proactive mode to improve energy consumption. This work is a direct continuation of the work in [1]. Compared to that, we proposed 3 new policies for energy harvesting. These policies are tested on all our candidate workloads, and demonstrate (some) better energy-harvesting results.

CHAPTER 3

Experimental setup

3.1 Hardware and software

All the experiments in this work use a personal computer with an NVIDIA Geforce GTX 960 GPU and an AMD Ryzen 7 3700x CPU.

The system under test runs Linux Ubuntu 20.04.2 LTS. The framework is run in a Docker environment[4] with an NVIDIA extension, so that the GPU can run in this environment. This docker environment was made by Bakker [1], to ensure the right dependencies for the framework.

3.2 Updates in the framework

The framework does not immediately work on the personal computer mentioned in section 3.1, because the framework was initially designed for an Intel processor, but the personal computer uses an AMD processor. Therefore, some modifications were needed. For the energy consumption of the CPU for Intel processors we can use Intel Running Average Power Limit (RAPL) from an virtual file located at /sys/class/powercap/intel-rapl. For the energy consumption for AMD processors we use an interface to read from the model-specific registers (MSRs). This interface is provided in the virtual file located at /dev/cpu/{CPU}/msr. To be able to access this first the driver must be loaded, this can be done with the command: modprobe msr. This interface consists out of multiple bytes of different information. To get the energy consumption we read out the bytes for the package energy and the power unit. Then we calculate the energy unit with the code in listing 3.1. The energy unit is then multiplied with the package energy to calculate the total energy consumption since startup of the interface in Joule. Then the same as with the Intel processors the difference of energy consumption is then calculated with the last call to get the energy consumption in that time frame.

Listing 3.1: Energy unit calculation

```
double energy_unit = pow(0.5, (double)((powerunit>>8) &0x1f));
```

For the GPU there is also a compatibility problem. The *nvmlDeviceSetGpuLockedClocks* and *nvmlDeviceResetGpuLockedClocks* CUDA functions are not supported on the GPU of the personal computer. Instead we used the *setApplicationCoreClockRate* and *resetApplicationCoreClockRate* functions that are delivered in the framework. The *setApplicationCoreClockRate* function without some changes did give errors, because not all core clock rates are accepted with a given memory clock rate. Therefore we use the CUDA function, *nvmlDeviceGetSupportedGraphicsClocks*, this function takes a memory clock rate and gives back an array with the acceptable core clock rates and a length of the array. This array is in descending order and with an iteration through the array the clock rate will be chosen that is either the same as the clock rate that is given in the *setApplicationCoreClockRate* function or is the first clock rate lower. We made the choice to only take a look if the clock rate is lower or equal then the given clock rate from the *setApplicationCoreClockRate* function, instead in looking at the difference between the two clock rates. This choice was made to make the code simple and efficient with the knowledge that the array is in descending order.

On the personal computer there is a bug were the total energy consumption of an application does not start with zero. This bug is fixed by making an extra call to get the energy consumption of the node in the monitor of the node. The node here is the whole platform, including both the GPU and CPU.

3.3 Running the experiments

For the running of the experiments we make an *experiment* function. This function takes the arguments for specifying which core and GPU to use and also the number of iterations to run the application. For the specification for what application to profile, we make use of a template, so we can specify what application profiler class to use with the function call. The function *experiment*:

```
experiment<Application>(arguments, iterations);
```

In the arguments are the flags that are given to the experiment program. Like the number of executions what application is executed and if it the policy should be executed single- or multi-core. The application is then executed once without an energy-harvesting policy and then once for all the different energy-harvesting policies. The function *experimentControl* is used to execute without policy:

```
experimentControl<Application>(arguments, iterations)
```

Then the function *experimentEnergyMonitor* is used to execute with policies:

```
experimentEnergyMonitor<Application>(arguments, iterations, policy, singleCore)
```

singleCore is a Boolean that flips the policies between single-and multi-core.

We run two sets of experiments: *single-core* and *multi-core*, where we focus the DVFS to either a single core or to all cores. For the single-core experiments, we only change the CPU frequency of the core that is used, and not on the whole CPU. This is done because we can not use the Intel driver (used in the framework) to change the whole CPU frequency directly; as all the applications that we test only use a single core, this experiments would show the case in which energy harvesting does not affect other applications running on the other cores. The multi-core experiments change the frequency of all the cores of the CPU; in this case, we use the same code as for the single core, but then applied for all the cores separately, in a (sequential) for loop. These experiments are more aggressive in their application of DVFS, but could affect other applications which run on these other cores.

CHAPTER 4

Heterogeneous workloads

4.1 Workload classification

In Chapter 1 we mention that the candidate workloads need to be diverse. For the workloads to be diverse, we first need to analyse them, to identify their differences. The only classification that was done in the research from Bakker [1] is if the workload was CPU-bound, GPU-bound or balanced. However, more aspects of workloads may have an impact on their energy consumption.

What also has a proven impact on the energy consumption with DVFS, which is used in the policies, is if the workload is memory-bound or compute-bound [9]. Memory-bound or compute-bound means that the workload performs more computations or memory transfers, respectively. If the workload is compute-bound the frequency has more impact on performance than when the workload is memory-bound. For our energy-harvesting policies, this can mean that when an workload is memory-bound, the compute frequency can be lowered more without slowing down the execution [9]. Another difference in workloads is if the workload is communication light or heavy. Although communication performance does not change with the frequencies [5], it still impacts the energy consumption of the workload.

Different patterns of CPU-GPU interaction are also interesting to look at, because our policies react on these interactions. Different patterns, like one long kernel or multiple short kernels, could, with different policies, show different energy-consumption numbers. Extrapolating, an interesting difference between workloads is their *complexity*, seen here as the maximum number of different kernels, and/or the number of lines of code.

So the workload classification criteria are:

1. CPU-bound, GPU-bound or balanced;
2. Memory-bound or compute-bound for the CPU and GPU;
3. Communication light or heavy;
4. Different patterns of CPU-GPU interactions;
5. Small, medium, or large code-base/number of kernels.

4.2 Workloads testing

We select our workloads from the following benchmarking suites: Rodinia [15] and CUDA SDK samples version 10.1 [12]. We also take a look at a separate implementation of Jacobi, because Jacobi is used in the research from Bakker [1] as a workload, and is often used as a benchmark for heterogeneous systems.

For the classification of the workload applications of the benchmarks, we use the NVIDIA Profiler from the CUDA toolkit 10.1 [13]. To get data, on whether the workload CPU-bound, GPU-bound or balanced we use the Dependency Analysis table in the visual version of nvprof. This table shows metrics collected from a dependency analysis of the program execution. The data here is summarized per function type and gives a percentage of the total execution time. So the larger the percentage the impact on the total application runtime. In the execution timeline of nvprof you can see which function is executed on the CPU or GPU. With the assumption that `<Other>` in the table is CPU-computation, since the profiler only reads CUDA calls on the CPU. When we look at the impacts on the execution time for CPU and GPU, we can estimate if the application is CPU-bound, GPU-bound or balanced. Then, we use the same table to distinguish the memory functions from the computation functions, and assess whether the workloads for the CPU and GPU are memory- or compute-bound.

The nvprof execution timeline from the visual version is also used to characterize the pattern of CPU-GPU interactions. Interesting to look at is the length and repetition of the kernel, but also the memory copies that can be intertwined with the kernel. We also used nvprof to look at the number of different kernels and also the usages of these different kernels, so if the kernels are used equally or if some are only executed ones. This is for the complexity classification. We use the differences in classification from the nvprof to choose from the workloads application a set of candidate workloads.

Table 4.1: Candidate workloads from Rodinia [15]

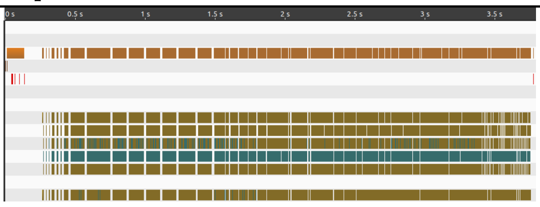
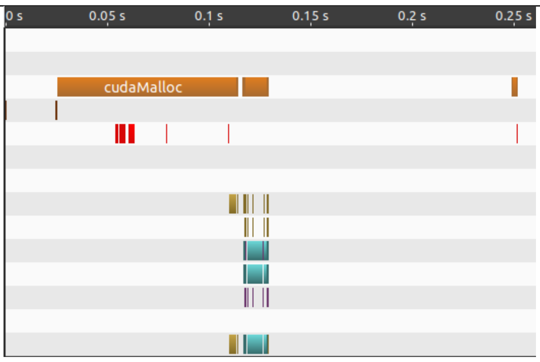
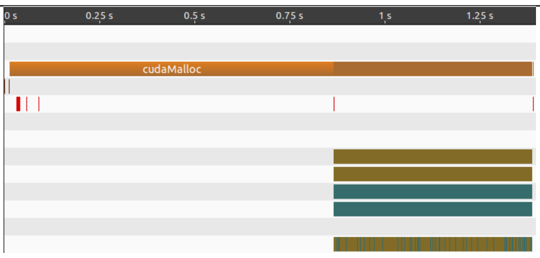
Id	**Name**	**nvprof**	**Description**	**Dataset**
1	Streamcluster	 0 s 0.5 s 1 s 1.5 s 2 s 2.5 s 3 s 3.5 s	For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center.	Minimum centers: 10, Maximum centers: 20 Dimension: 256, Number of points: 65536, Chunk size: 65536, Cluster size: 1000
2	BFS	 0 s 0.05 s 0.1 s 0.15 s 0.2 s 0.25 s cudaMalloc	Breadth-First Search	The input file is a large graph file, 64.2 MB
3	Myocyte	 0 s 0.25 s 0.5 s 0.75 s 1 s 1.25 s cudaMalloc	Myocyte application models cardiac myocyte (heart muscle cell) and simulates its behavior.	Time: 100ms Number of instances: 1 Method of parallelization: 0

Table 4.1: Candidate workloads from Rodinia [15]

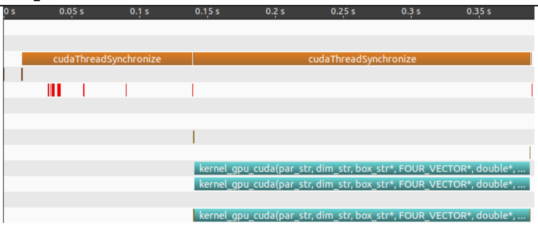
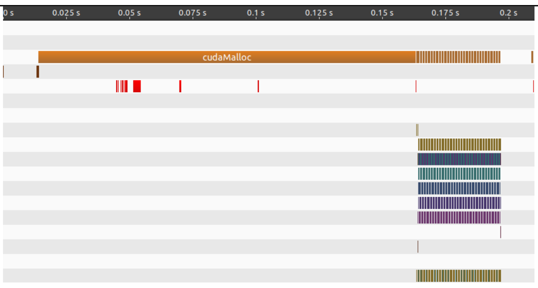
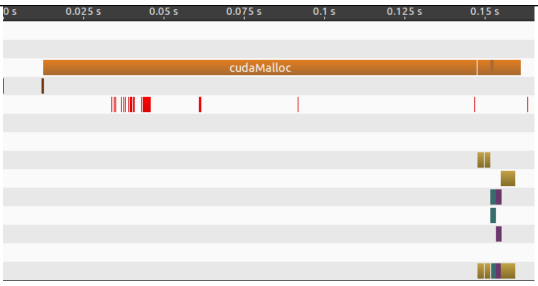
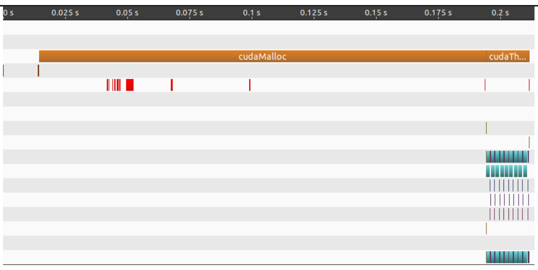
Id	Name	nvprof	Description	Dataset
4	LavaMD	 0 s 0.05 s 0.1 s 0.15 s 0.2 s 0.25 s 0.3 s 0.35 s cudaThreadSynchronize cudaThreadSynchronize kernel_gpu_cuda(par_str, dim_str, box_str*, FOUR_VECTOR*, double*, ...	It calculates particle potential and relocation due to mutual forces between particles within a large 3D space.	Number of boxes in one dimension: 10 and the image is an grayscale image, 458 pixels by 502 pixels.
5	SRAD-V1	 0 s 0.025 s 0.05 s 0.075 s 0.1 s 0.125 s 0.15 s 0.175 s 0.2 s cudaMalloc	SRAD (Speckle Reducing Anisotropic Diffusion) is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations.	Number of iterations: 100, Saturation coefficient 0.5, Number of rows: 502, Number of columns: 458
6	NW	 0 s 0.025 s 0.05 s 0.075 s 0.1 s 0.125 s 0.15 s cudaMalloc	Needleman-Wunsch is a nonlinear global optimization method for DNA sequence alignments.	Length of sequences: 2048, Penalty value: 10
7	Particlefilter-float	 0 s 0.025 s 0.05 s 0.075 s 0.1 s 0.125 s 0.15 s 0.175 s 0.2 s cudaMalloc cudaTh...	The particle filter is statistical estimator of the location of a target object given noisy measurements of that target's location and an idea of the object's path in a Bayesian framework.	Dimension X: 128, Dimension Y: 128, Number of frames: 10, Number of particles: 1000

Table 4.1: Candidate workloads from Rodinia [15]

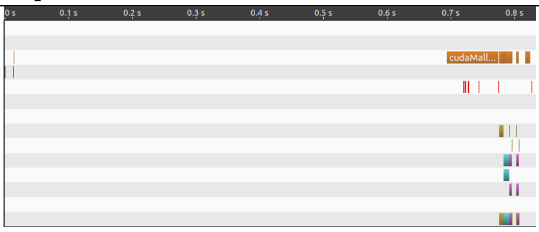
Id	Name	nvprof	Description	Dataset
8	Kmeans	 0 s 0.1 s 0.2 s 0.3 s 0.4 s 0.5 s 0.6 s 0.7 s 0.8 s cudaMall...	Dividing a cluster of data objects into K sub-clusters.	Output cluster center coordinates: On, and input file is 494020 lines long with data.

Table 4.2: Candidate workloads from NVIDIA Corporation [12]

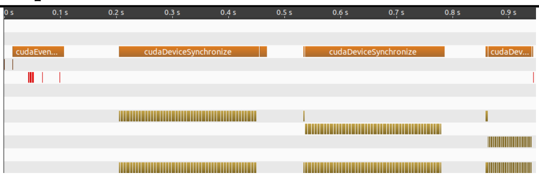
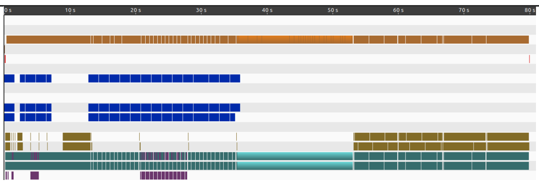
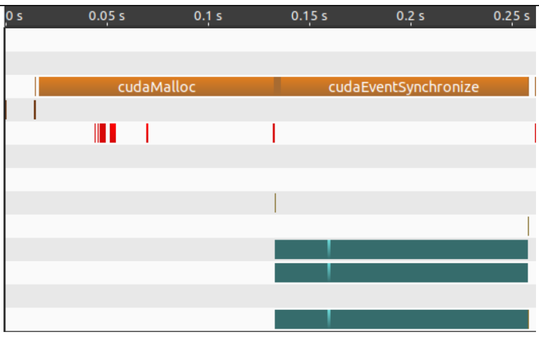
Id	Name	nvprof	Description	Dataset
9	Bandwidth	 0 s 0.1 s 0.2 s 0.3 s 0.4 s 0.5 s 0.6 s 0.7 s 0.8 s 0.9 s cudaEven... cudaDeviceSynchronize cudaDeviceSynchronize cudaDev...	It measures the memcopy bandwidth of the GPU and memcpy bandwidth across PCI-e.	All three memory types are tested.
10	Unified-Memory-Perf	 0 s 10 s 20 s 30 s 40 s 50 s 60 s 70 s 80 s	It demonstrates the performance differences between different types of communication and memory calls, using a matrix multiplication kernel.	
11	matrixMul	 0 s 0.05 s 0.1 s 0.15 s 0.2 s 0.25 s cudaMalloc cudaEventSynchronize	Implements matrix multiplication	

Table 4.3: Candidate workloads for the Jacobi application[1]

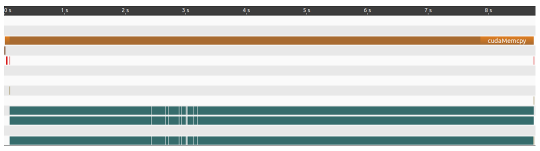
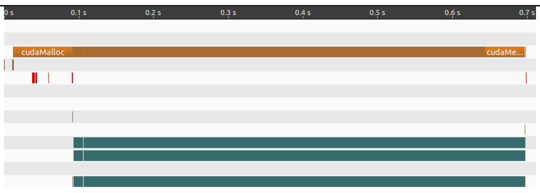
Id	Name	nvprof	Description	Dataset
12	Jacobi unoptimized	 0 s 1 s 2 s 3 s 4 s 5 s 6 s 7 s 8 s cudaMemcpy	Jacobi is used to find approximate numerical solutions for systems of linear equations of the form $Ax = b$. This version has an unoptimized CUDA kernel.	Number of elements (X, Y): (1024, 1024), Iterations: 10000, Kernel: 1 (unoptimized), File is 1024x1024 coefficient matrix and a 1024x1 vector
13	Jacobi optimized	 0 s 0.1 s 0.2 s 0.3 s 0.4 s 0.5 s 0.6 s 0.7 s cudaMalloc cudaMe...	Jacobi is used to find approximate numerical solutions for systems of linear equations of the form $Ax = b$. This version has an optimized CUDA kernel.	Number of elements (X, Y): (1024, 1024), Iterations: 10000, Kernel: 2 (optimized), Tilesize: 4, File is 1024x1024 coefficient matrix and a 1024x1 vector

Table 4.1, 4.2 and Table 4.3 consist of candidate workload applications from different sources. Every workload has a name, a short description, and the dataset and parameters used for its execution.

The tables also show the NVIDIA Profiler profile for these workloads. These graphs are timelines of execution, in which different lines correspond to different functions executing in the heterogeneous system. For exmple, the third row from the top (in orange) indicates the CPU calls to CUDA functions. The fifth row from the top (in red), shows the overhead from the profiling on the CPU. Finally, rows seven and below, show the GPU activity: in blue. communication, in brown, memory copies, and different colors for different kernel executions.

4.3 States with current policies

To further analyse these workloads, we proceed to benchmark their performance and energy consumption. For representative data, we aim for the execution time of the workloads to be either around the 60 seconds, and have at least 3 repetitions of the application. The 60 seconds is to better compare the applications to each other, while the 3 repetitions is to better compare patterns of the states.

In the coming sections we discuss the characteristics of each application individually.

[1] This implementation is from: `https://github.com/MMichel/CudaJacobi`.

4.3.1 Streamcluster

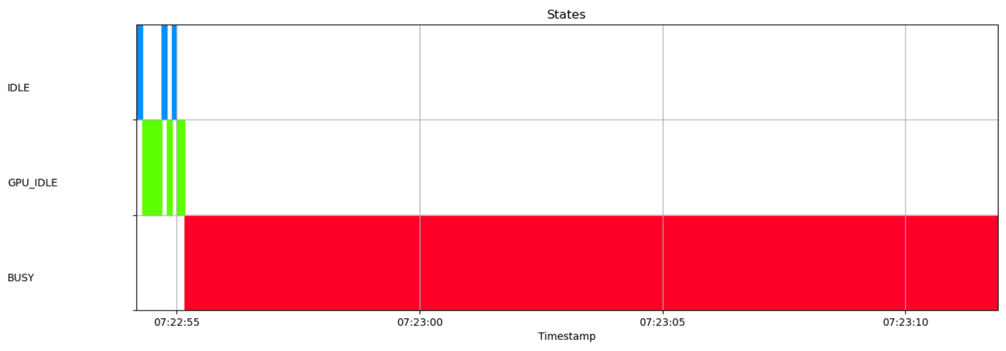


Figure 4.1: Streamcluster states

The Streamcluster from Table 4.1 is executed 10 times in the framework on a single CPU core. The framework gives the states for those 10 repetitions in Figure 4.1. The time span of the Figure 4.1 is shorter then the actual execution, because the GPU reads fail and stop earlier. This is most likely because of the high number of CUDA calls. Even after an update by Bakker [1], the framework was still not able to read all the CUDA calls. So we drop this application from this research.

4.3.2 BFS

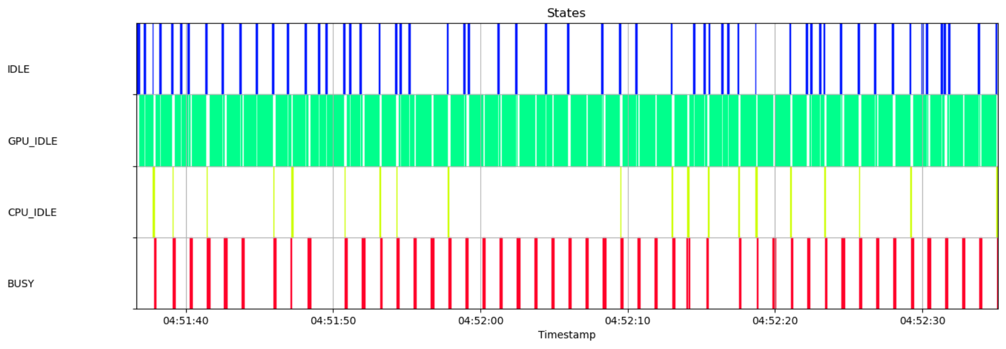


Figure 4.2: BFS states

The BFS from Table 4.1 is executed 50 times in the framework on one CPU core. The framework gives the states for those 50 repetitions in Figure 4.2. If we compare the states from the framework with the NVIDIA Profiler in the Table 4.2, we observe that the kernel execution is mainly in the BUSY state and a little bit in the CPU_IDLE state with before it an IDLE state. The execution around the kernel is mostly in the GPU_IDLE state. The CPU_IDLE state and IDLE state is either from the operating system, hardware or is a measurement error, in the utilization for the CPU it drops to zero. It does not seem to be part of the application.

The BFS application is a CPU heavy workload, that has CPU computation before and after the kernel execution. The application also has some communication, and is mostly compute-bound.

4.3.3 Myocyte

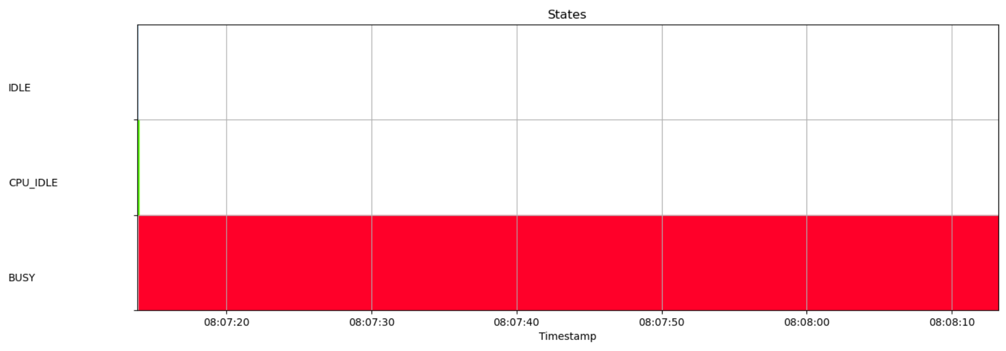


Figure 4.3: Myocyte states

The Myocyte from Table 4.1 is executed 75 times in the framework on one CPU core. The framework gives the states for those 75 repetitions in Figure 4.3. The myocyte application has the same problem as the streamcluster application, in that the GPU reads fail. The timespan in Figure 4.3 also does not line up here with the actual execution time. So we drop this application from this research.

4.3.4 LavaMD

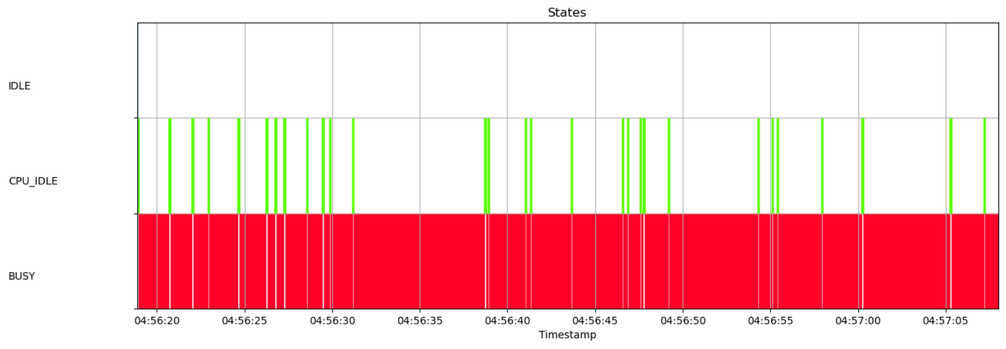


Figure 4.4: LavaMD states

The LavaMD from Table 4.1 is executed 150 times in the framework on one CPU core. The framework gives the states for those 150 repetitions in Figure 4.5. If we compare the states from the framework with the NVIDIA Profiler in the Table 4.2, we observe that the kernel with synchronisation and the synchronisation before it mostly fall in the BUSY state, the CPU_IDLE state is either from the operating system, hardware or is a measurement error, in the utilization for the CPU it drops to zero. It does not seem to be part of the application.

The LavaMD application is in the BUSY state the whole time, it makes use of a lot of thread synchronisation and has one big kernel in one execution. The application also has low communication and is compute-bound.

4.3.5 SRAD_V1

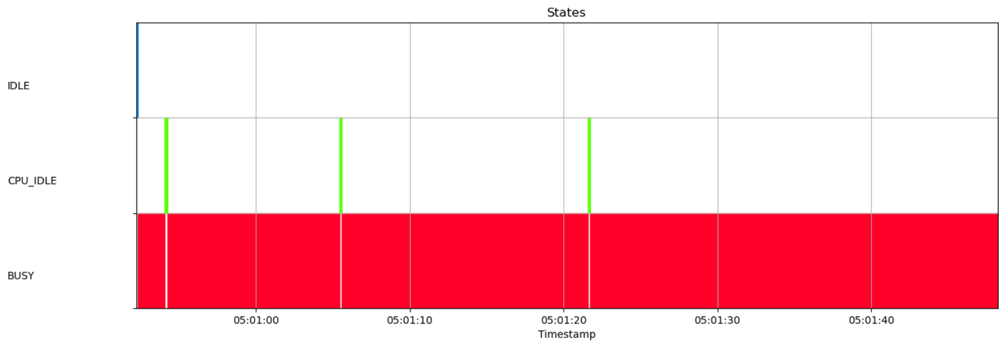


Figure 4.5: SRAD_V1 states

The SRAD_V1 from Table 4.1 is executed 350 times in the framework on one CPU core. The framework gives the states for those 350 repetitions in Figure 4.5. If we compare the states from the framework with the NVIDIA Profiler in the Table 4.2, we observe that the kernels with memory copies and the setup before it mostly fall in the BUSY state, the CPU_IDLE state is either from the operating system, hardware or is a measurement error, in the utilization for the CPU it drops to zero. It does not seem to be part of the application.

This program does have a problem that when it is run multiple times the execution time and energy consumption increases with every execution. As we could not understand the reasons for this behaviour, we also drop this application from this research.

The SRAD_V1 application is also in the BUSY state, but has different kernels with communication, memory copies, in between. So the application has high communication and also more complexity since there are 6 different kernels.

4.3.6 NW

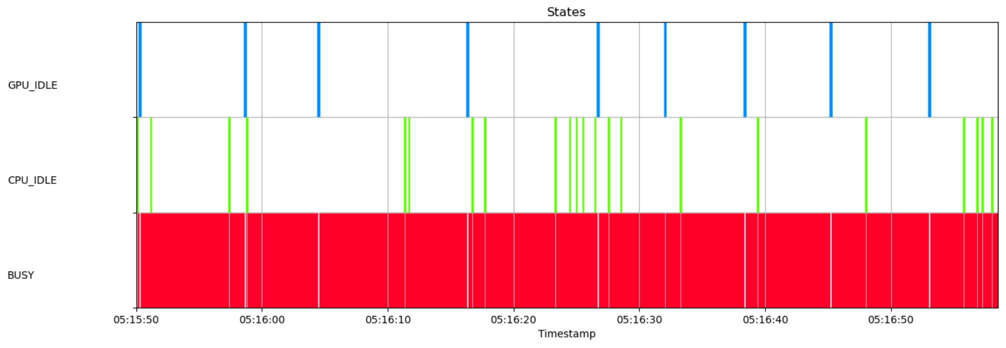


Figure 4.6: NW states

The NW from Table 4.1 is executed 400 times in the framework on one CPU core. The framework gives the states for those 400 repetitions in Figure 4.6. If we compare the states from the framework with the NVIDIA Profiler in the Table 4.2, we observe that the whole program is in the BUSY state, this is interesting since the cudaMalloc in Figure 4.2 is most likely in the GPU_IDLE state. The GPU_IDLE in Figure 4.6, because the utilization of the GPU is barely active, sometimes, the utilization just drops a bit and is then in the GPU_IDLE state. The CPU_IDLE state is either from the operating system, hardware or is a measurement error, in the utilization for the CPU it drops to zero. It does not seem to be part of the application.

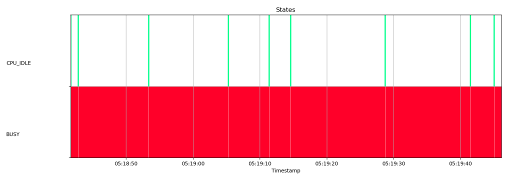


Figure 4.7: NW states with MinMax policy

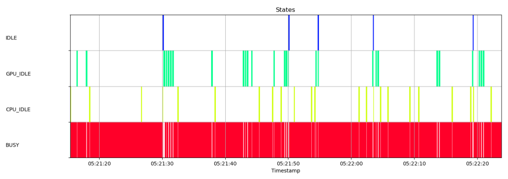


Figure 4.8: NW states with System policy

The figures 4.7 and 4.8 where the states are given with the policies. We notice how the MinMax policy removes idle states, while the system policy only adds more idle states. The overall pattern stays the same and this application was one the most different between policies.

The NW application is mostly in the BUSY state, but has a relatively low utilization of the GPU, likely because the kernel computation is relatively short compared to the communication between CPU and GPU. This application is mostly CPU-bound and has compared to the GPU execution a lot of communication.

4.3.7 Particlefilter-float

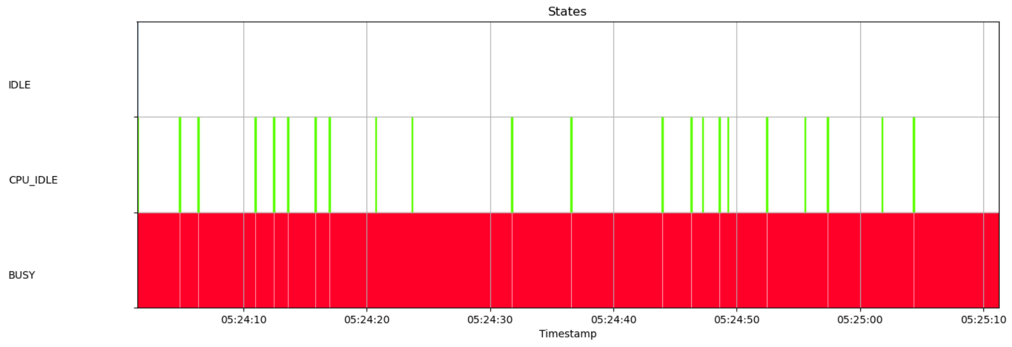


Figure 4.9: Particlefilter-float states

The Particlefilter-float from Table 4.1 is executed 500 times in the framework on one CPU core. The framework gives the states for those 500 repetitions in Figure 4.9. If we compare the states from the framework with the NVIDIA Profiler in the Table 4.2, we observe that the whole program, so the cudaMalloc and the kernels, are in the BUSY state. The CPU_IDLE state is either from the operating system, hardware or is a measurement error, in the utilization for the CPU it drops to zero. It does not seem to be part of the application.

The Particlefilter-float application is in the BUSY state the whole time, but has thread synchronisation with multiple kernels. This application has not much communication. The utilization of the GPU Particlefilter-float is low around 30%, but it does stay the whole time above the threshold.

4.3.8 Kmeans

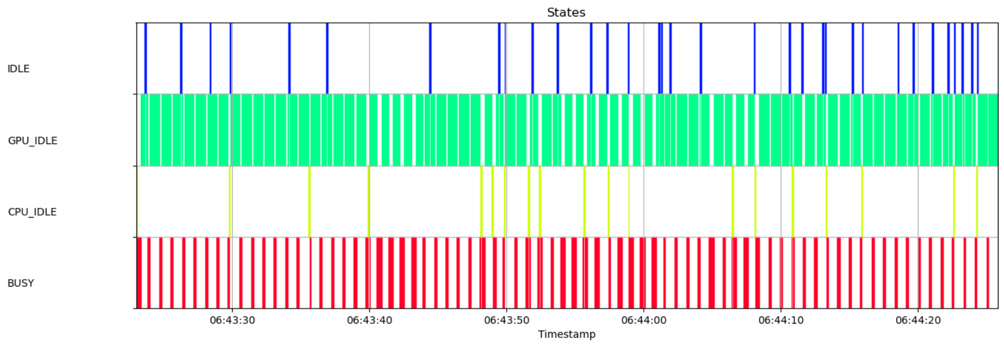


Figure 4.10: Kmeans states

The Kmeans from Table 4.1 is executed 75 times in the framework on one CPU core. The framework gives the states for those 75 repetitions in Figure 4.10. If we compare the states from the framework with the NVIDIA Profiler in the Table 4.2, we observe that the memory allocation with kernels and communication is in the BUSY state while the rest of the application in the GPU_IDLE state is. The CPU_IDLE state and IDLE state is either from the operating system, hardware or is a measurement error, in the utilization for the CPU it drops to zero. It does not seem to be part of the application.

The Kmeans application is an application that is CPU-bound. The application is compute-bound with low communication and complexity.

4.3.9 Bandwidth

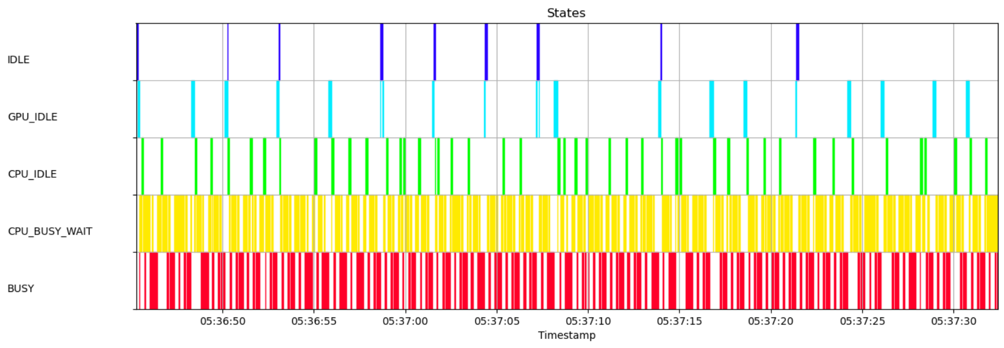


Figure 4.11: Bandwidth states

The Bandwidth from Table 4.2 is executed 50 times in the framework on one CPU core. The framework gives the states for those 50 repetitions in Figure 4.11. If we compare the states from the framework with the NVIDIA Profiler in the Table 4.2, we observe that there are a lot of different states. We can not really observe an correlation, but there is some sort of pattern although it looks more random.

The bandwidth application is an application with a lot of communication differentiated in three blocks from CPU to GPU, GPU to CPU and inside the GPU.

4.3.10 UnifiedMemoryPerf

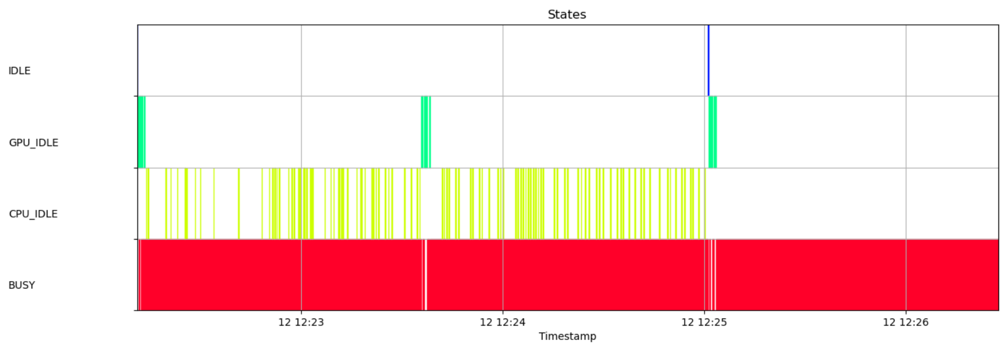


Figure 4.12: UnifiedMemoryPerf states

The UnifiedMemoryPerf from Table 4.2 is executed 3 times in the framework on one CPU core. The framework gives the states for those 3 repetitions in Figure 4.9. With the original framework there were to many CUDA events for the framework to keep up, so these reads are done with an updated version of the framework, by Bakker [1]. If we compare the states from the framework with the NVIDIA Profiler in the Table 4.2, we observe that only the start for a short time has only a active CPU while the rest of the matrix multiplication kernels are int the BUSY state. The CPU_IDLE state is either from the operating system, hardware or is a measurement error, in the utilization for the CPU it drops to zero. It does not seem to be part of the application.

The UnifiedMemoryPerf application is an application with the same kernel, but with different means of communication and memory calls. The application is mostly in the BUSY state, but with different utilization rates throughout the execution. So the complexity of this program is high and the communication is also high.

4.3.11 MatrixMul

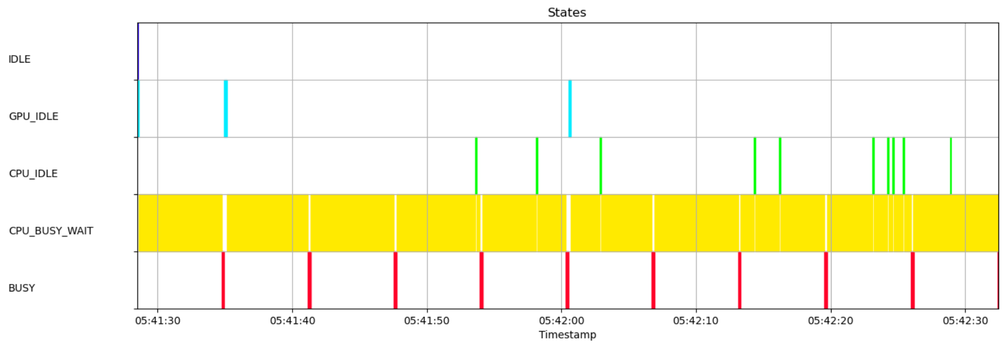


Figure 4.13: MatrixMul states

The MatrixMul from Table 4.2 is executed 50 times in the framework on one CPU core. The framework gives the states for those 50 repetitions in Figure 4.13. If we compare the states from the framework with the NVIDIA Profiler in the Table 4.2, we observe that the application start and ends with a BUSY state to do setup and finalization, besides the first run, because everything has to startup. The rest is in CPU_BUSY_WAIT, because of the synchronization for the kernel.

The MatrixMul is GPU heavy, although it is not the only candidate workload application with synchronisation it is one of few with the CPU_BUSY_WAIT state. Communication is low and the complexity is also low.

4.3.12 Jacobi

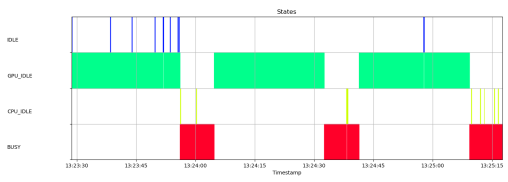


Figure 4.14: Jacobi unoptimized kernel states

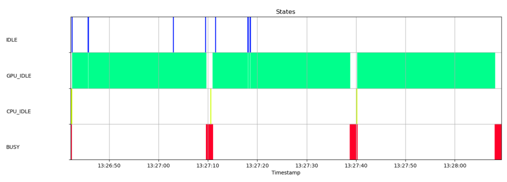


Figure 4.15: Jacobi optimized kernel states

The Jacobi application from Table 4.3 is executed 3 times in the framework on one CPU core. The framework gives the states for those 3 repetitions in Figure 4.14 and Figure 4.15. With the original framework there were to many CUDA events for the framework to keep up for the optimized version, so these reads are done with an updated version of the framework, by Bakker [1]. If we compare the states from the framework with the NVIDIA Profiler in the Table 4.2, we observe that the nvprof only shows with this program the execution of the kernel and everything outside the kernel is only CPU. So the nvprof shows only what is in the BUSY state. The CPU_IDLE and IDLE state is either from the operating system, hardware or is a measurement error, in the utilization for the CPU it drops to zero. It does not seem to be part of the application. A difference between the optimized and unoptimized is the execution duration of the kernel, so the optimized version is shorter in the BUSY state. Another difference is that the unoptimized version gives a more precise answer. The unoptimized version gives an answer with relative error of 0.0 while the optimized version gives an answer with relative error 0.000628.

4.3.13 Summary

Table 4.4 presents a characterization of our 10 workloads. Specifically, for each application, we show the cummulated time the application spends in each state of execution.

4.4 Energy harvesting with current policies

To calculate the average energy consumption, each workload is run 5 times, and an average of these runs is calculated and reported. This methodology is needed to reduce the impact of noise on the measurements. Such noise can appear because we run all experiments on a personal computer, and background tasks can change the energy consumption. We also do this because the CPU energy consumption sometimes overflows and becomes negative, so we filter these negative energy consumption out.

Table 4.5 shows the energy consumption (and gain/loss) of the applications when using the three original policies: NoAction, System, and MinMax. We make the following observations.

The System energy policy has, on most applications, a negative impact on the energy consumption. Exceptions are BFS, Kmeans and matrixMul. With those three application there is a significant difference in energy consumption between MinMax and System, with System winning. The BFS and kmeans application do share quite a lot of similarities (see figures 4.2 and 4.10), while matrixMul is quite different. For 5 applications, the MinMax policy is better for energy harvesting when compared to NoAction, because the energy consumption is lower, as is the runtime. The Table 4.5 does not show a clear pattern that can be used to create one single, best policy for all applications.

Table 4.4: Time spent in state for the NoAction policy*

Id	Applications	IDLE	CPU_IDLE	GPU_IDLE	CPU_BUSY_WAIT	BUSY
2	BFS	9.1%	1.4%	77.6%	0.0%	11.9%
4	LavaMD	0.0%	5.9%	0.0%	0.0%	94.1%
5	SRAD_V1	0.2%	0.8%	0.0%	0.0%	99.1%
6	NW	0.0%	3.5%	2.2%	0.0%	94.3%
7	Particlefilter-float	0.1%	3.1%	0.0%	0.0%	96.8%
8	Kmeans	4.6%	2.5%	69.7%	0.0%	23.3%
9	Bandwidth	1.5%	9.5%	4.9%	46.1%	38.0%
10	UnifiedMemoryPerf	0.2%	6.4%	1.7%	0.0%	91.7%
11	matrixMul	0.2%	1.7%	0.6%	94.9%	2.7%
12	Jacobi unoptimized	0.8%	0.9%	75.4%	0.0%	22.9%
13	Jacobi optimized	0.9%	0.3%	94.1%	0.0%	4.6%

*The time spent in state does change per policy, like we show in Figure 4.7 and 4.8, about the NW application. NW one of the applications that the state changes to most per policy. The overall pattern do not change.

Table 4.5: Time and energy consumption for original three policies

Id	Applications	Policy					
		NoAction		MinMax		System	
		Energy	Time	Energy	Time	Energy	Time
2	BFS	5248.7 J	60.5 s	6099.5 J (16.2%)	61.3 s (1.2%)	5752.3 J (9.6%)	62.4 s (3.1%)
4	LavaMD	7454.3 J	52.1 s	7430.8 J (-0.3%)	51.8 s (-0.5%)	7751.4 J (4.0%)	52.9 s (1.5%)
6	NW	6103.3 J	64.9 s	6886.8 J (12.8%)	65.0 s (0.1%)	7083.4 J (16.1%)	67.6 s (4.2%)
7	Particlefilter-float	8540.8	89.5 s	8906.8 J (4.3%)	86.1 s (-3.7%)	9437.0 J (10.5%)	89.3 s (-0.2%)
8	Kmeans	5729.4 J	66.2 s	6922.7 J (20.8%)	66.0 s (-0.3%)	5956.9 J (4.0%)	66.1 s (-0.1%)
9	Bandwidth	6337.7 J	50.4 s	6321.7 J (-0.3%)	50.4 s (-0.0%)	6417.0 J (1.3%)	50.9 s (1.1%)
10	UnifiedMemoryPerf	33188.3 J	266.1 s	32685.0 J (-1.5%)	255.9 s (-3.8%)	33723.8 J (1.6%)	257.6 s (-3.2%)
11	matrixMul	9295.6 J	66.6 s	11111.1 J (19.5%)	66.6 s (-0.0%)	9675.8 J (4.1%)	66.7 s (0.1%)
12	Jacobi unoptimized	10980.4 J	118.1 s	10590.7 J (-3.5%)	109.4 s (-7.4%)	10901.6 J (-0.7%)	109.6 s (-7.2%)
13	Jacobi optimized	7697.2 J	95.3 s	7486.3 J (-2.7%)	86.3 s (-9.4%)	8267.5 J (7.4%)	97.0 s (1.8%)

CHAPTER 5

New Energy-harvesting policies

5.1 Original policies

Something that was added later to the framework from Bakker [1], that was not yet implemented in the thesis, is that the frequency does not immediately change, but is changed gradually. This was done to prevent large energy spikes. There were already two policies implemented in the framework, these are explained in Section 2. These policies were based on the changes of the utilization states. We also have results of the effectiveness of the two policies in Table 4.5.

5.2 Static Frequencies

Before we try policies, we first take a look at setting static frequencies for a workload to see if there are frequencies that can reduce energy consumption without impacting the execution time negatively. We test 5 different static frequencies, compared to the maximum. We test 10%, 25%, 50%, 75% and 100% compared to the maximum frequency. Although this maximum frequency that the CPU gives back is not necessarily the effective maximum frequency, because this maximum frequency is only sustainable for a short period of time. So the 100% is in reality closer to the 75% than the arithmetic difference would indicate. We implemented the static frequency in the framework by creating a staticFreq policy and adding an additional parameter for the percentage. This policy sets the frequency with equation 5.1.

$$maximum\_frequency \cdot (percentage/100) \tag{5.1}$$

The results for the single-core static frequency changes are in Table 5.1, while the impact of the static frequency changes is visualised in Figure 5.1. The static frequencies 10% and 25% are fairly similar with most applications, especially the execution time. The energy consumption is mostly lower for the static frequency of 10%, compared to the 25%, but most of the application have significant longer execution time and significant higher energy consumption then NoAction. There are exceptions like matrixMul that get similar results for all 5 static frequencies. The 75% and 100% have for most applications a faster execution time, then NoAction. The static frequency 75% seems to do better with energy consumption then NoAction and the 100% static frequency. The 50% is mostly in between with the results for most applications, but looks to be closer to the 10% and 25% static frequency. Lower frequencies does not mean lower energy consumption necessarily, mostly it does mean that the execution time becomes longer. So with dynamic policies it would we possible to find the balance, and only lower the frequency of the parts of the application that do not need energy for the performance. The goal of the dynamic policies is to find those parts of the application and also how much the frequency is lowered. The 75% static frequency shows that minimizing and maximizing frequency does not necessarily give the best results for energy harvesting.

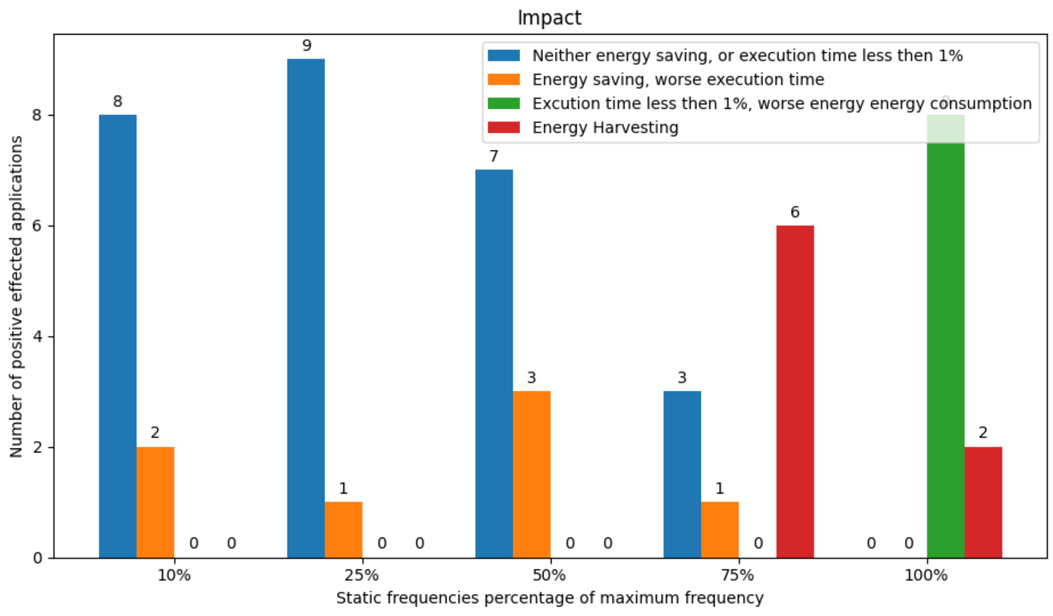


Figure 5.1: Impact of single-core static frequency changes on applications

The results for the multi-core static frequency changes are presented in Table 5.2, while the impact of the static frequency changes is visualised in Figure 5.2. Most results are similar to results of the single-core static frequency changes, especially the execution time. There are a two main differences between the single-core and multi-core. If we look at the lower frequencies, like 10%, 25% and 50%, the energy consumption is lower compared to the single-core static frequency changes and for the 100% the execution time seems to do worse with some applications. Although the 100% frequency for multi-core was done on a different run that did get some different results even for the NoAction. The execution time for NW and Particlefilter_float are for the 100% static frequency both higher then they are compared to the NoAction of that run. Also the energy consumption of the Jacobi applications are for the 100% static frequency lower then they are compared to the NoAction of that run.

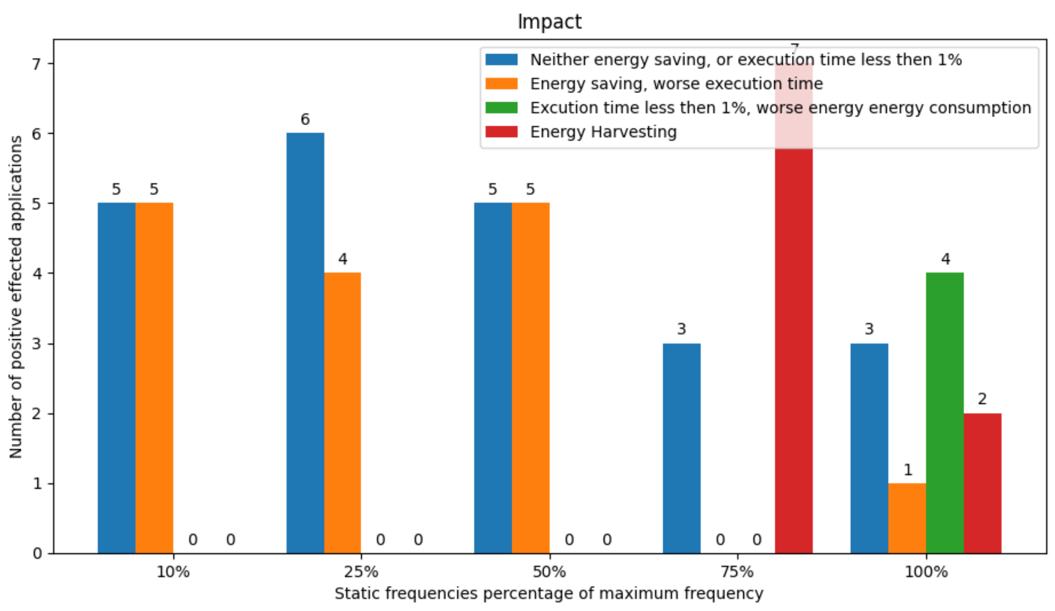


Figure 5.2: Impact of mutli-core static frequency changes on applications

5.3 New policies

5.3.1 New policies implementation

The framework from Bakker [1] does not support more then 2 policies without changes. This is because which policy to use is determined by a Boolean. We changed this Boolean and the references of this Boolean to an enumerate with the policies. Then for the enforcement of the policies in the energy monitor we changed the if-statements for what policy is used to a switch-case statement for all the policies in the enumeration.

5.3.2 Maximum Frequency

We have a Maximum Frequency policy to have a more stable policy to compare the different policies on different applications. This policy is basically the same as setting the frequency statically on 100%, as mentioned in Section 5.2. This policy sets the frequencies of both the CPU and GPU to the maximum frequency of both devices, and basically ignoring the states. This policy can possibly be good in energy harvesting for applications that get good results from the MinMax policy and are mostly in the BUSY state like LavaMD, these results are in Table 4.5. This is because the MinMax policy that is in the BUSY state also has the maximum frequency for both the CPU and GPU.

5.3.3 Ranked MinMax

We observed that a lot of the applications in the Table 4.5 have a lot of BUSY states, but do differ in the utilization of the CPU and GPU. The MinMax policy only looks at busy or idle, not at how busy. That is the idea of the ranked MinMax, to look at the utilization and determine the frequency on this. So instead of maximising the CPU frequency and GPU frequency if it is not idle. It will look at the utilization and will 'rank' it in 4 different frequencies. The ranking is done based on the utilization percentage: 25%, 50% and 75% are the thresholds for different frequencies. The frequencies caps are linearly created, with the minimum frequency at the idle

Table 5.1: Time and energy consumption for single-core static frequency changes

Id	Applications	Static frequency											
		NoAction		10%		25%		50%		75%		100%	
		Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
2	BFS	5248.7 J	60.5 s	8180.9 J (55.9%)	113.3 s (87.3%)	8663.6 J (65.1%)	111.2 s (83.7%)	6486.4 J (23.6%)	91.7 s (51.6%)	5073.5 J (-3.3%)	63.7 s (5.3%)	6300.7 J (20.0%)	60.8 s (0.5%)
4	LavaMD	7454.3 J	52.1 s	6540.7 J (-12.3%)	58.4 s (12.1%)	7478.5 J (0.3%)	58.3 s (11.9%)	7222.2 J (-3.1%)	55.6 s (6.7%)	7264.8 J (-2.5%)	51.5 s (-1.2%)	7400.2 J (-0.7%)	52.0 s (-0.1%)
6	NW	6103.3 J	64.9 s	6979.6 J (14.4%)	102.9 s (58.6%)	7831.6 J (28.3%)	104.2 s (60.5%)	6152.1 J (0.8%)	93.9 s (44.7%)	6609.5 J (8.3%)	76.8 s (18.4%)	7355.4 J (20.5%)	66.4 s (2.2%)
7	Particlefilter-float	8540.8	89.5 s	8961.0 J (4.9%)	128.8 s (43.9%)	9465.9 J (10.8%)	129.9 s (45.1%)	8493.8 J (-0.6%)	120.7 s (34.9%)	8812.1 J (3.2%)	103.8 s (16.0%)	9508.8 J (11.3%)	86.5 s (-3.3%)
8	Kmeans	5729.4 J	66.2 s	8257.2 J (44.1%)	112.3 s (69.7%)	8352.3 J (45.8%)	112.7 s (70.2%)	7127.6 J (24.4%)	93.0 s (40.5%)	5321.2 J (-7.1%)	64.1 s (-3.1%)	7074.4 J (23.5%)	66.1 s (-0.2%)
9	Bandwidth	6337.7 J	50.4 s	6359.2 J (0.3%)	62.3 s (23.6%)	6496.7 J (2.5%)	62.6 s (24.2%)	5973.3 J (-5.8%)	56.7 s (12.5%)	6071.3 J (-4.2%)	49.2 s (-2.3%)	6192.5 J (-2.3%)	50.4 s (0.1%)
10	UnifiedMemoryPerf	33188.3 J	266.1 s	34613.8 J (4.3%)	290.3 s (9.1%)	34617.4 J (4.3%)	299.9 s (12.7%)	33425.8 J (0.7%)	278.1 s (4.5%)	32597.9 J (-1.8%)	261.0 s (-1.9%)	33221.0 J (0.1%)	256.7 s (-3.5%)
11	matrixMul	9295.6 J	66.6 s	8608.1 J (-7.4%)	69.4 s (4.3%)	9016.5 J (-3.0%)	69.4 s (4.3%)	9629.2 J (3.6%)	69.2 s (4.0%)	9318.6 J (0.2%)	68.5 s (2.8%)	9619.8 J (3.5%)	66.6 s (-0.0%)
12	Jacobi unoptimized	10980.4 J	118.1 s	14975.5 J (36.4%)	181.8 s (53.9%)	15708.3 J (43.1%)	181.9 s (54.0%)	12644.7 J (15.2%)	151.4 s (28.2%)	10049.7 J (-8.5%)	109.3 s (-7.5%)	12186.3 J (11.0%)	109.2 s (-7.6%)
13	Jacobi optimized	7697.2 J	95.3 s	12384.5 J (60.9%)	161.1 s (69.1%)	12678.0 J (64.7%)	161.0 s (69.0%)	9946.1 J (29.2%)	129.0 s (35.3%)	7067.5 J (-8.2%)	86.3 s (-9.4%)	8042.1 J (4.5%)	86.5 s (-9.3%)

Table 5.2: Time and energy consumption for multi-core static frequency changes

Id	Applications	Static frequency											
		NoAction		10%		25%		50%		75%		100%	
		Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
2	BFS	5248.7 J	60.5 s	8767.8 J (67.0%)	116.1 s (91.9%)	8830.1 J (68.2%)	115.4 s (90.7%)	7155.3 J (36.3%)	91.8 s (51.7%)	5554.7 J (5.8%)	63.4 s (4.8%)	6276.2 J (19.6%)	60.2 s (-0.5%)
4	LavaMD	7454.3 J	52.1 s	6916.0 J (-7.2%)	57.7 s (10.7%)	6878.6 J (-7.7%)	57.8 s (10.9%)	7119.5 J (-4.5%)	54.3 s (4.3%)	7230.5 J (-3.0%)	49.7 s (-4.6%)	7473.5 J (0.3%)	51.0 s (-2.1%)
6	NW	6103.3 J	64.9 s	6077.0 J (-0.4%)	103.8 s (59.9%)	6703.2 J (9.8%)	105.9 s (63.1%)	5346.8 J (-12.4%)	89.8 s (38.4%)	6405.1 J (4.9%)	68.8 s (6.0%)	7787.6 J (27.6%)	70.4 s (8.5%)
7	Particlefilter-float	8540.8	89.5 s	7573.6 J (-11.3%)	128.3 s (43.3%)	7868.6 J (-7.9%)	129.8 s (45.1%)	6146.1 J (-28.0%)	112.8 s (26.0%)	8538.6 J (-0.0%)	88.8 s (-0.7%)	10301.2 J (20.6%)	91.5 s (2.2%)
8	Kmeans	5729.4 J	66.2 s	7600.2 J (32.7%)	115.1 s (73.9%)	7830.2 J (36.7%)	115.6 s (74.6%)	6508.0 J (13.6%)	s (13.6%)	5491.9 J (-4.1%)	64.6 s (-2.4%)	6633.3 J (15.8%)	66.5 s (0.5%)
9	Bandwidth	6337.7 J	50.4 s	5485.0 J (-13.5%)	63.2 s (25.4%)	6020.6 J (-5.0%)	63.5 s (26.0%)	5934.0 J (-6.4%)	56.8 s (12.7%)	5773.5 J (-8.9%)	49.0 s (-2.8%)	6165.4 J (-2.7%)	51.0 s (1.2%)
10	UnifiedMemoryPerf	33188.3 J	266.1 s	26469.8 J (-20.2%)	293.8 s (10.4%)	27620.3 J (-16.8%)	296.1 s (11.3%)	26818.9 J (-19.2%)	282.2 s (6.1%)	33156.6 J (-0.1%)	262.8 s (-1.2%)	34542.5 J (4.1%)	258.4 s (-2.9%)
11	matrixMul	9295.6 J	66.6 s	10538.7 J (13.4%)	69.7 s (4.7%)	9372.0 J (0.8%)	69.7 s (4.7%)	10274.4 J (10.5%)	69.1 s (3.8%)	10320.3 J (11.0%)	67.9 s (2.0%)	10086.7 J (8.5%)	66.5 s (-0.2%)
12	Jacobi unoptimized	10980.4 J	118.1 s	15176.7 J (38.2%)	186.9 s (58.2%)	13585.5 J (23.7%)	186.9 s (58.3%)	11359.5 J (3.5%)	152.4 s (29.1%)	10899.2 J (-0.7%)	109.1 s (-7.6%)	8039.1 J (-26.8%)	109.0 s (-7.7%)
13	Jacobi optimized	7697.2 J	95.3 s	11071.3 J (43.8%)	166.7 s (75.0%)	10521.2 J (36.7%)	166.6 s (74.9%)	8631.8 J (12.1%)	131.1 s (37.5%)	7332.9 J (-4.7%)	86.4 s (-9.3%)	5021.9 J (-34.8%)	85.8 s (-10.0%)

threshold of 10%, and the maximum frequency if it is above 75% utilization. So the formula is:

$$(maximum\_frequency - minimum\_frequency) \cdot utilization + minimum\_frequency \quad (5.2)$$

The utilization is between 0 and 1, and will be rounded up to the next threshold divided by 100. This enforces the frequency to be from minimum to maximum. The utilization is used from the moment the state changes.

5.3.4 Scaled MinMax

We observed that some applications have a utilization that changes while it stays in the same state. Then the ranked MinMax might fix it self into a frequency that is only good for the start of the applications, like the utilization is low for the startup of the application, while still being in the same state. The scaled MinMax will not only change the frequency on state changes, but also every iterations of the framework when they check for a state change, when the CPU or GPU are not in the IDLE state, the CPU or GPU frequency will be changed based on the utilization with the same formula 5.2 as the ranked MinMax, but for the scaled MinMax there are no thresholds for utilization. There are no threshold, because there are many frequency updates so smaller patterns can be utilized. With smaller patterns we mean small fluctuations in the utilization.

5.4 Results

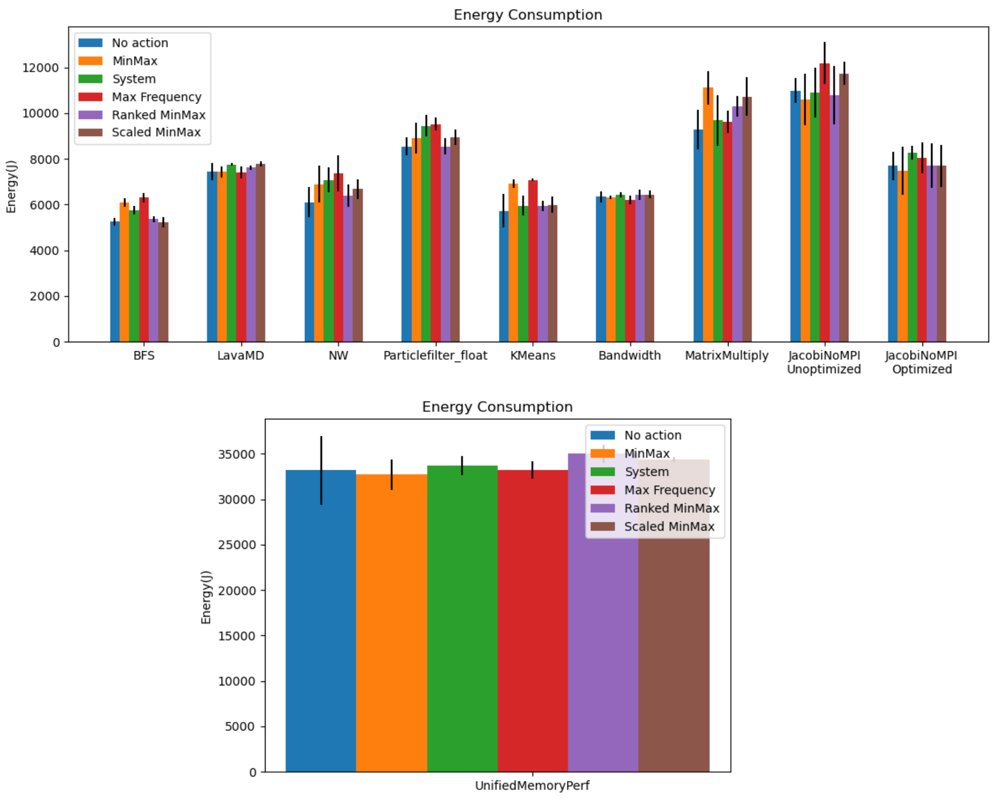


Figure 5.3: Energy Consumption of all the single-core energy harvesting policies

The bandwidth application seems to be the least impacted by the different policies, this can be because of what we talked about in Section 4.3.9 that the state pattern is all over the place and does not seem to match a pattern in the application. It is also interesting that the best policy, not by much, is actually the Max Frequency policy, so ignoring the states seem to do a little bit better.

Matrix Multiply is mostly in the CPU_BUSY_WAIT state, as can be seen in Figure 4.4. The best policy for reducing energy consumption as seen in Figure 5.3 is the Max Frequency policy. This might suggest that minimizing the CPU frequency in the CPU_BUSY_WAIT state, might not be the best solution. It is not better than NoAction, but maybe a policy that differentiate between the GPU_IDLE state and CPU_BUSY_WAIT state can be more effective then NoAction.

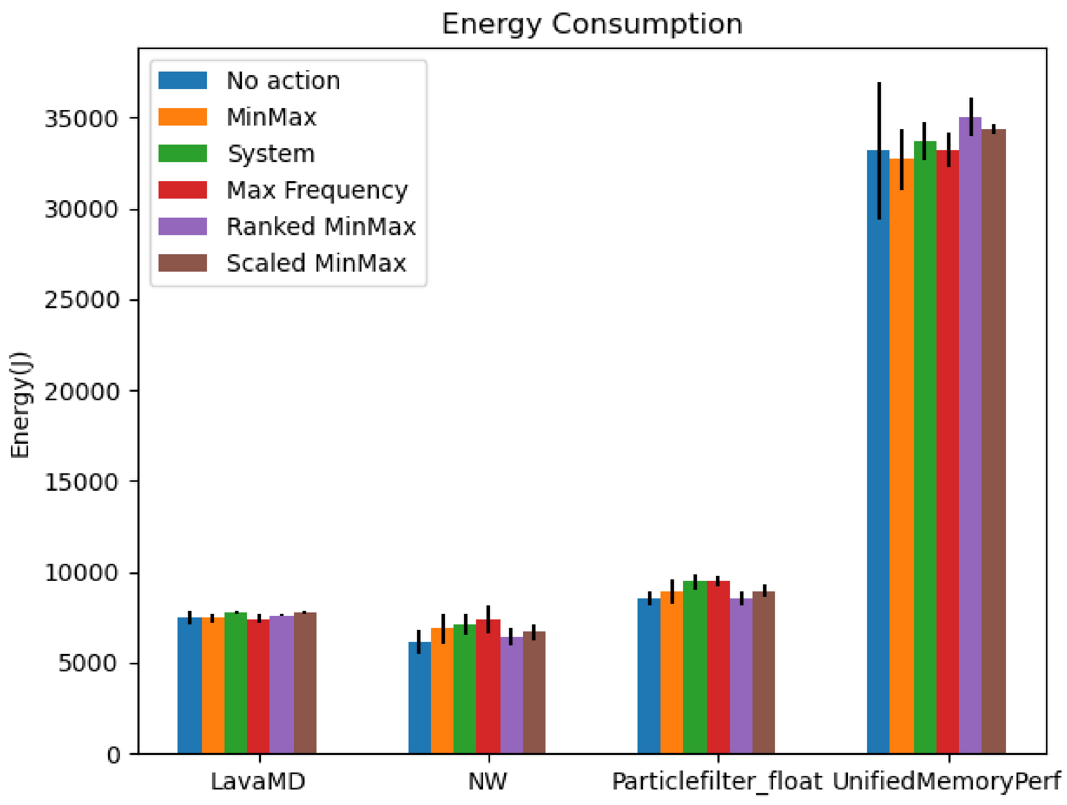


Figure 5.4: Comparing energy consumption of the policies of the applications with mostly BUSY states

If we compare the 4 applications that are mostly in the busy state, as can be seen in Figure 4.4. The applications are LavaMD, Particlefilter-float, NW and UnifiedMemoryPerf. If we compare them with the energy consumption of all the policies in Figure 5.4, we notice that we can split them up in two groups, a group with low and a group with high utilization of the GPU, both have high CPU utilization. The Particlefilter-float and NW have low utilization of the GPU, like we mention in Section 4.3.6 and Section 4.3.7. We notice here that the policies that specifically take the utilization in account like Scaled and Ranked MinMax do a lot better for Particlefilter-float and NW then the other policies, while Maximum Frequency is the worst option for lowering energy consumption. For the LavaMD and UnifiedMemoryPerf, the policies that set the frequency to the maximum, seem to do the best in lowering the energy consumption, so in this case MinMax and Max Frequency. All the other policies that try to be more nuanced, end up hurting the energy harvesting process. Although the differences in energy consumption is for LavaMD and UnifiedMemoryPerf smaller then for the Particlefilter-float and NW.

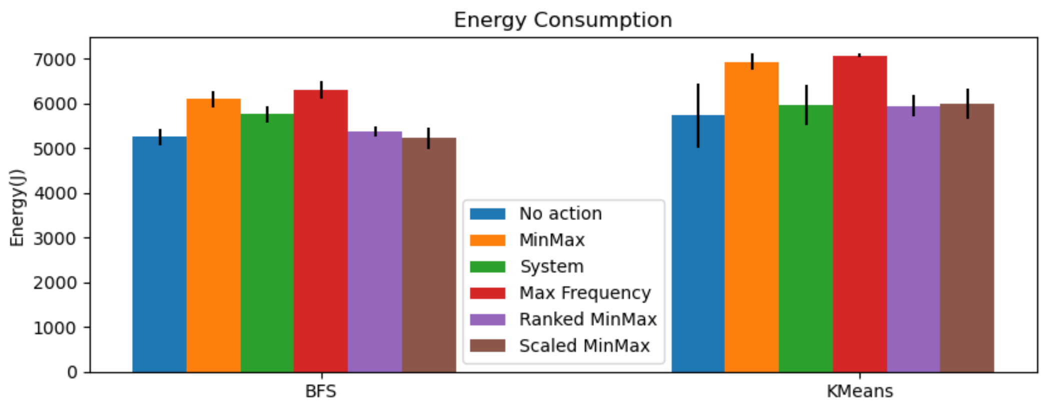


Figure 5.5: Comparing energy consumption of the policies of the BFS and Kmeans applications

The BFS and Kmeans are comparable in states as can be seen in Figure 4.2 and Figure 4.10. The BFS and Kmeans do react similar on the MinMax and Max Frequency policies, as can be seen in Figure 5.5, for both applications they negatively impact the energy consumption the worst compared to the other policies, with the Max Frequency doing even worse then MinMax. The more nuanced policies do better. The difference between BFS and Kmeans is for the energy consumption with the System, Ranked MinMax and Scaled MinMax policies. While BFS has a clear differences between those three, with System as the worst of the three and Scaled MinMax as the best, the Kmeans application all three policies have around the same energy consumption.

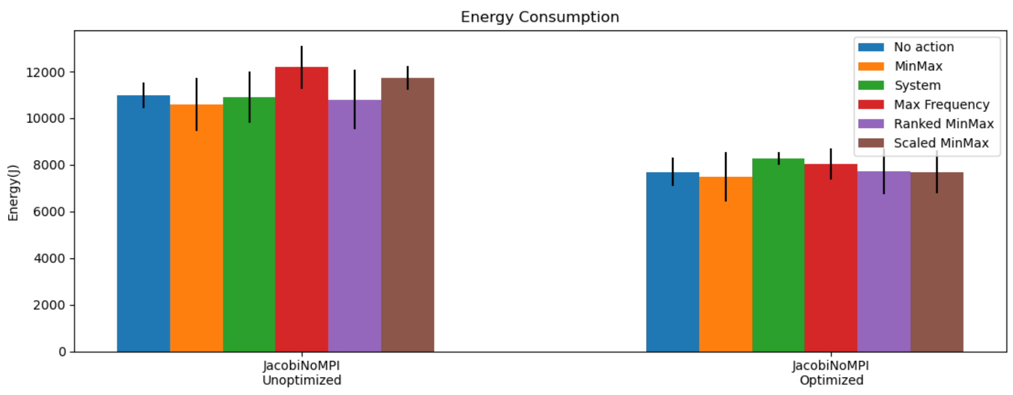
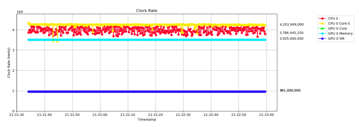


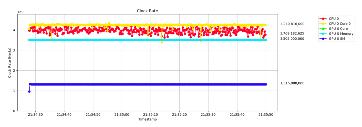
Figure 5.6: Comparing energy consumption of the policies of the Jacobi versions

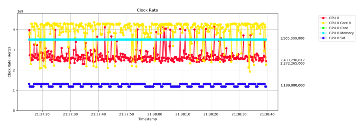
The Jacobi application has the two versions the unoptimized and optimized kernel version. These differ in states a little, like we mentioned in Section 4.3.12, in the fact that the optimized version has a smaller BUSY state. Both of the Jacobi version have some of the highest energy saving with the MinMax policy, the unoptimized version saves 3.5% energy and the optimized version saves 2.7% energy, from Figure 4.5. If we look at the percentages of the Maximum Frequency policy in Table 5.3 we see that the unoptimized version does worse with the Maximum Frequency policy then the optimized version. This is not really logical since the BUSY state is larger with the unoptimized version, so the part that the unoptimized version overlaps with the MinMax is larger than the overlap with the MinMax of the optimized version.

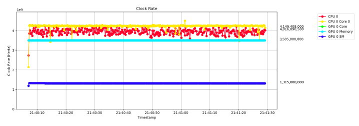
Table 5.3: Time and energy consumption for new policies

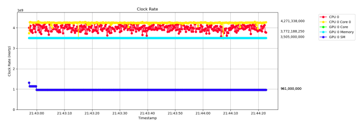
Id	Applications	Policy							
		NoAction		Maximum Frequency		Ranked MinMax		Scaled MinMax	
		Energy	Time	Energy	Time	Energy	Time	Energy	Time
2	BFS	5248.7 J	60.5 s	6300.7 J (20.0%)	60.8 s (0.5%)	5365.8 J (2.2%)	60.6 s (0.2%)	5220.2 J (-0.5%)	60.6 s (0.2%)
4	LavaMD	7454.3 J	52.1 s	7400.2 J (-0.7%)	52.0 s (-0.1%)	7611.8 J (2.1%)	52.1 s (-0.0%)	7785.7 J (1.9%)	53.1 s (1.9%)
6	NW	6103.3 J	64.9 s	7355.4 J (20.5%)	66.4 s (2.2%)	6394.4 J (4.8%)	67.8 s (4.4%)	6675.9 J (8.4%)	70.4 s (8.4%)
7	Particlefilter-float	8540.8	89.5 s	9508.8 J (11.3%)	86.5 s (-3.3%)	8537.6 J (-0.0%)	90.8 s (1.5%)	8928.5 J (4.5%)	94.2 s (5.3%)
8	Kmeans	5729.4 J	66.2 s	7074.4 J (23.5%)	66.1 s (-0.2%)	5941.1 J (3.7%)	66.6 s (0.6%)	5992.8 J (4.6%)	66.9 s (1.1%)
9	Bandwidth	6337.7 J	50.4 s	6192.5 J (-2.3%)	50.4 s (0.1%)	6428.0 J (1.4%)	50.6 s (0.5%)	6439.0 J (1.6%)	51.3 s (1.9%)
10	UnifiedMemory-Perf	33188.3 J	266.1 s	33221.0 J (0.1%)	256.7 s (-3.5%)	35015.6 J (5.5%)	257.9 s (-3.1%)	34376.9 J (3.6%)	250.8 s (-5.7%)
11	matrixMul	9295.6 J	66.6 s	9619.8 J (3.5%)	66.6 s (-0.0%)	10302.0 J (10.8%)	66.7 s (0.1%)	10715.9 J (15.3%)	66.8 s (0.3%)
12	Jacobi unoptimized	10980.4 J	118.1 s	12186.3 J (11.0%)	109.2 s (-7.6%)	10786.9 J (-1.8%)	109.5 s (-7.3%)	11731.6 J (6.8%)	109.6 s (-7.2%)
13	Jacobi optimized	7697.2 J	95.3 s	8042.1 J (4.5%)	86.5 s (-9.3%)	7709.9 J (0.2%)	86.3 s (-9.4%)	7695.1 J (-0.0%)	86.8 s (-8.9%)


NoAction


MinMax


System


Max Frequency


Ranked MinMax

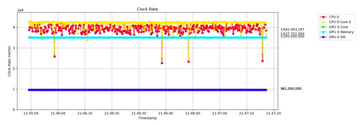

Scaled MinMax

Figure 5.7: Particlefilter-float clock rate

5.5 Policy Analysis

The particlefilter-float application spent most time in the BUSY state, as can be seen in Table 4.4, but has low GPU utilization as we discussed in Section 4.3.7. This means that MinMax and Max Frequency should be very similar, this is the case as we see in Figure 5.7. We can also see in the figure that NoAction also lowers the clock rate of the GPU as do the Scaled MinMax and Ranked MinMax which both look at the low utilization. The Ranked MinMax takes longer to get to the lower frequency, mainly because it updates the frequency less then the Scaled MinMax. On the other side we see that the Scaled MinMax is more influenced by the drops in utilization that as we discussed in Section 4.3.7 is not part of the program, we can see it is influenced since there are a few drops in clock rate for the CPU. We only see them here since there are more frequency updates. The most different clock rate graph is the graph from the System policy. The System policy seems to have more freedom then NoAction in the BUSY state. The Scaling MinMax and Ranked MinMax are based on the theoretical maximum and minimum. The theoretical maximum is higher then the actual frequency the CPU can sustain, so even if the utilization drops a little bit this will not change the frequency, because it is still above the frequency the CPU can actually sustain.

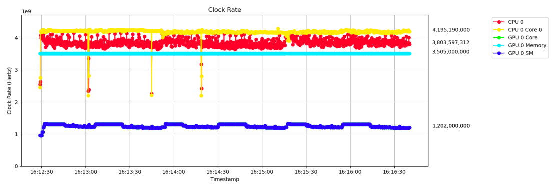


Figure 5.8: UnifiedMemoryPerf Scaled MinMax clock rate

The UnifiedMemoryPerf application also spends most time in the BUSY state, as can be seen in Table 4.4, but the utilization of the GPU in this BUSY state changes quite a bit. This is also what we can see in the clock rate figure of the scaled MinMax in Figure 5.8.

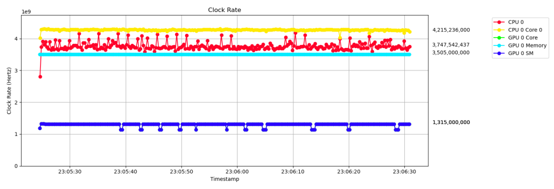


Figure 5.9: Kmeans MinMax clock rate

The Kmeans application does have different states as can be seen in the Figure 4.10. The

GPU has busy and idle states. We can also see that back in the Figure 5.9, although the frequency probably should be more down then up. This can happen when the changes in the states happen to fast after each other, because it takes some time to change the frequency.

5.6 Multi-core policies results

The results in Section 5.4 were all made by changing only the single-core the application was run on. In this section we take a look what happens when we change the frequency of all the CPU cores.

The NW application did look to run slower with this run, in Table 5.4, then the previous results in Section 5.4. We used the NoAction from the previous section, to compare them but the experiment was ran with NoAction, and even this NoAction had a slower execution time of 77.1 seconds. The Particlefilter_float has a similar problem with the execution time as the NoAction here is 102.9 seconds. Both of these application do have similar energy consumption, the NoActions have even a lower energy consumption, 5680.1 and 7725.8 Joule respectively. The Jacobi applications have a similar problem, but here the energy consumption is significantly lower overall, even for NoAction with 8449.8 Joule for unoptimized and 5425.1 Joule for optimized.

The biggest difference between the changing of one core compared to the changing of all cores of the CPU is in the range of values. For the energy of multi-core policies, we observed savings up to 34.8%, with the worst-case scenario showing an increase of energy consumption of 27.6%. For execution time of multi-core policies, we observed speedups up to 10%, with the worst-case scenario showing a delay of 27.1%. In Table 5.4 the Ranked MinMax and Scaled MinMax do extremely well for energy consumption, only 2 for Ranked MinMax and 3 for Scaled MinMax have a negative impact on the energy. For the execution time it is the other way round. There are only 3 for Ranked MinMax and 1 for Scaled MinMax were these policies have a negative impact on the execution time. A reason for the negative impact on execution time and positive impact on energy consumption is that the policy is more effective, but it creates more overhead for the execution time.

Table 5.4: Time and energy consumption for multi-core policies

Id	Applications	Policy											
		NoAction		MinMax		System		Maximum Frequency		Ranked MinMax		Scaled MinMax	
		Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
2	BFS	5248.7 J	60.5 s	6499.7 J (23.8%)	70.6 s (16.7%)	5669.3 J (8.0%)	69.7 s (15.2%)	6276.2 J (19.6%)	60.2 s (-0.5%)	5294.3 J (0.9%)	61.2 s (1.2%)	5496.3 J (4.7%)	70.8 s (17.0%)
4	LavaMD	7454.3 J	52.1 s	6962.4 J (-6.6%)	52.6 s (1.0%)	7024.6 J (-5.8%)	52.3 s (0.4%)	7473.5 J (0.3%)	51.0 s (-2.1%)	6951.1 J (-6.8%)	52.9 s (1.5%)	7125.0 J (-4.4%)	53.8 s (3.3%)
6	NW	6103.3 J	64.9 s	6465.5 J (5.9%)	77.0 s (18.6%)	7132.7 J (16.9%)	74.1 s (14.2%)	7787.6 J (27.6%)	70.4 s (8.5%)	5619.0 J (-7.9%)	78.5 s (21.0%)	5635.6 J (-7.7%)	82.5 s (27.1%)
7	Particlefilter-float	8540.8	89.5 s	9245.1 J (8.2%)	99.6 s (11.3%)	10028.8 J (17.4%)	96.9 s (8.3%)	10301.2 J (20.6%)	91.5 s (2.2%)	7666.4 J (-10.2%)	102.8 s (14.8%)	7578.4 J (-11.3%)	107.6 s (20.2%)
8	Kmeans	5729.4 J	66.2 s	6248.0 J (9.1%)	77.0 s (16.3%)	6303.4 J (10.0%)	74.4 s (12.4%)	6633.3 J (15.8%)	66.5 s (0.5%)	5514.4 J (-3.8%)	68.9 s (4.1%)	5932.2 J (3.5%)	77.9 s (17.7%)
9	Bandwidth	6337.7 J	50.4 s	5957.7 J (-6.0%)	54.0 s (7.1%)	6128.0 J (-3.3%)	52.3 s (3.8%)	6165.4 J (-2.7%)	51.0 s (1.2%)	6029.5 J (-4.9%)	53.5 s (6.2%)	6004.9 J (-5.3%)	54.7 s (8.5%)
10	UnifiedMemoryPerf	33188.3 J	266.1 s	28612.8 J (-13.7%)	263.1 s (-1.1%)	32491.1 J (-2.1%)	257.5 s (-3.2%)	34542.5 J (4.1%)	258.4 s (-2.9%)	27956.7 J (-15.8%)	262.5 s (-1.4%)	27810.9 J (-16.2%)	258.6 s (-2.8%)
11	matrixMul	9295.6 J	66.6 s	10442.3 J (12.3%)	67.6 s (1.5%)	10962.8 J (17.9%)	67.0 s (0.6%)	10086.7 J (8.5%)	66.5 s (-0.2%)	10913.3 J (17.4%)	67.5 s (1.4%)	10264.3 J (10.4%)	68.0 s (2.1%)
12	Jacobi unoptimized	10980.4 J	118.1 s	7802.1 J (-28.9%)	124.6 s (5.5%)	8192.6 J (-25.4%)	128.0 s (8.4%)	8039.1 J (-26.8%)	109.0 s (-7.7%)	8958.9 J (-18.4%)	109.3 s (-7.5%)	8440.3 J (-23.1%)	124.8 s (5.7%)
13	Jacobi optimized	7697.2 J	95.3 s	5467.1 J (-29.0%)	101.9 s (6.9%)	5280.8 J (-31.4%)	101.4 s (6.4%)	5021.9 J (-34.8%)	85.8 s (-10.0%)	6090.9 J (-20.9%)	86.6 s (-9.1%)	5400.4 J (-29.8%)	102.1 s (7.1%)

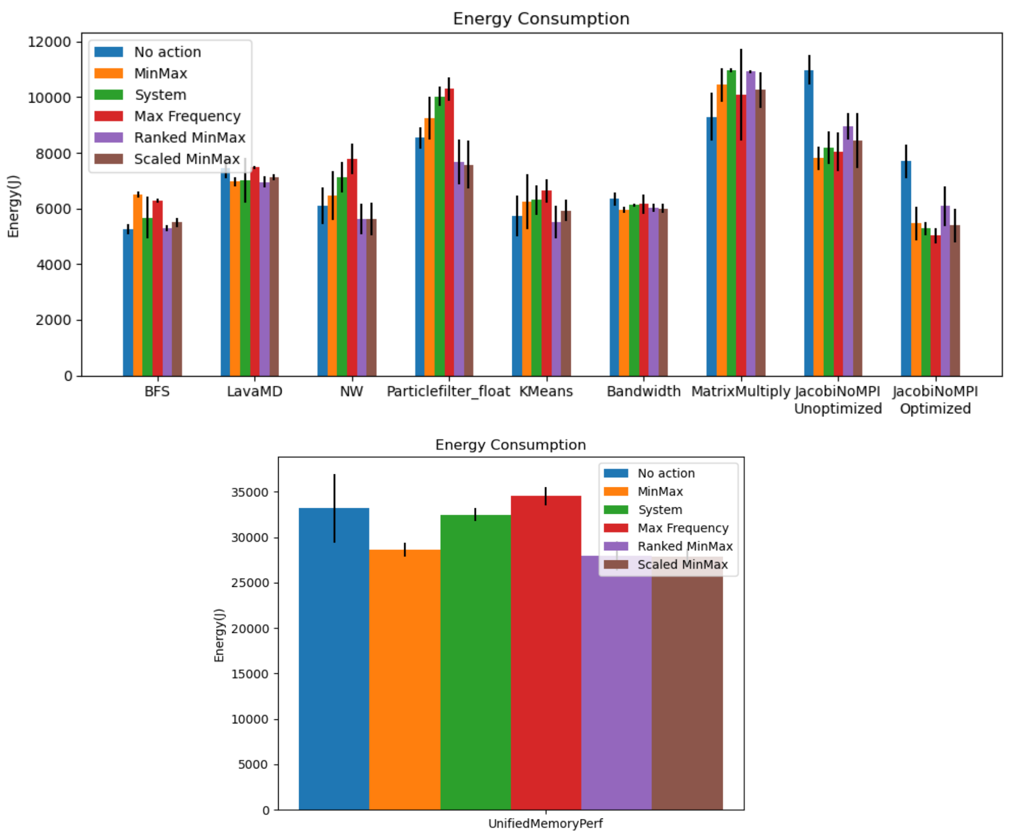


Figure 5.10: Energy consumption of all the multi-core energy harvesting policies

The applications BFS, Kmeans, Bandwidth, Matrix Multiply, NW and Particlefilter_float have somewhat the same pattern for energy consumption with the policies in Figure 5.3 and Figure 5.10. For the Bandwidth application the policies still do not differ much, but the Max Frequency is no longer the best. For Kmeans and BFS the Scaled MinMax and Ranked MinMax are still the best policies, but the System Policy is not comparable anymore. For the NW and Particlefilter_float the Scaled MinMax and Ranked MinMax, are still the best solutions for energy consumption. For the Matrix Multiply the best policy is still the Max Frequency, although the error bar here is quite significant. The four applications that are quite different are LavaMD, UnifiedMemoryPerf, Jacobi Unoptimized and Jacobi Optimized. LavaMD and UnifiedMemoryPerf were similar in pattern before and they still are only with the Figure 5.4, the Scaled MinMax and the Ranked MinMax were at first not good for the energy consumption, but as you can see in the Figure 5.3, these policies are now comparable to the MinMax. The Maximum Frequency was for one core frequency changes comparable to the MinMax, for all cores frequency changes it is the worst option for energy consumption. For the Jacobi application with the results from Figure 5.10 the policies react similar on the unoptimized and optimized version. For both versions the Scaled MinMax and Ranked MinMax both interact worse with the application for energy consumption then the rest of the policies.

5.7 In Practice

In practice, when deciding which policy to use, it is handy to have a process users can go through to choose the best policy. This process needs to be easy to understand and as short as possible. We note that, our recommendations are based purely on the results we got, so recommendations might not work on a different system or on applications radically different than the applications we covered. We also split our recommendation into single-core and multi-core recommendations.

5.7.1 Single-core policies

The System policy is never the best solution, which we can see in Tables 4.5 and 5.3, so we exclude that policy. First the energy harvesting of the MinMax policy needs to be tested on the given application. If it reduces energy consumption, the Maximum Frequency policy should be tried to see if it is better. If it increases energy consumption, the Ranked MinMax and Scaled MinMax policies should be tried. Then the policy with the best energy consumption decrease should be chosen. If there are no policies with energy consumption decreases, it is better to use NoAction.

There is one exception: if one knows whether the program has busy waits for the CPU, and the MinMax policy increases the energy consumption, it might be interesting to also take a look at the maximum frequency.

5.7.2 Multi-core policies

The System policy is again never the best policy to choose, which we can see in Table 5.4. For these policies, looking at the execution time is also very important. For the multi-core policies, we have a similar process as the single-core, but we also have to look at the execution time. So when the MinMax policy has a negative energy consumption and execution time lower or equal to 1.0% increase compared to NoAction, test the Scaled MinMax and Ranked MinMax. Otherwise test the Maximum Frequency policy. Then choose the best policy from the tested policies with the highest energy consumption decrease and an execution time lower or equal to 1.0% increase. If none of the policy do not have any energy consumption decrease or/and an increase in execution time larger then 1.0%, none of the policies would fit the application.

If the goal in not energy harvesting, but lowering the energy consumption, the Ranked MinMax and Scaled MinMax will always be interesting to test.

CHAPTER 6

Conclusion and Future Work

In the context of increasing energy consumption of ICT systems, among which CPU-GPU machines gain popularity, the main topic of this project is to define and evaluate energy harvesting policies for heterogeneous workloads. To this end, we evaluate a diverse set of workloads and policies; this chapter summarizes our findings and contributions, and suggests further research directions.

6.1 Main findings and contributions

The research question driving this research is:
What policies can improve energy-harvesting for heterogeneous workloads?

To answer this question we first updated our instrumentation by updating the existing energy-harvesting framework such that it can run on the personal computer used for the experiments.

The first part of the research focused on finding new heterogeneous workloads. We selected 10 diverse heterogeneous workloads, that differ in CPU- or GPU-boundness, memory- or compute-boundness, communication volume (light or heavy), and patterns of CPU-GPU interactions and complexity. The selection is presented in Chapter 4.

We further tested the two existing policies (System and MinMax), on these 10 heterogeneous workloads. These results represent a baseline for our future policies. We continued with the design of new energy harvesting policies. We created three new policies: Maximum Frequency, Ranked MinMax and Scaled MinMax, and tested these policies on the selected workloads. The results are presented in Table 4.5.

Finally, for our empirical evaluation, we benchmarked all workloads with all policies using two different runtime configurations: single-core and multi-core. The results are presented in Chapter 5, in Tables 5.3 and 5.4, respectively.

To summarize the results of our energy-harvesting approaches, we present the best possible policy for the single- and multi-core variants in Table 6.1. The table clearly indicates that different applications benefit the most from different policies. Moreover, not all workloads have a policy that improves energy consumption, and there is no policy that improves energy harvesting for all applications. However, every single-core policy has a heterogeneous application were it improves energy-harvesting. So MinMax, System, Maximum Frequency, Ranked MinMax and Scaled MinMax, for one single core, are all policies that can improve energy-harvesting for heterogeneous workloads. For the multi-core policies, this is not always the case; however, there are multi-core policies that are better at energy harvesting for some specific applications.

We found that applications with different policies give varying results for energy harvesting. For the energy of single-core policies, we observed savings up to 3.5%, with the worst-case scenario showing an increase of energy consumption of 23.5%. For execution time of single-core policies, we observed speedups up to 9.4%, with the worst-case scenario showing a delay of 8.4%. For the energy of multi-core policies, we observed savings up to 34.8%, with the worst-case scenario showing an increase of energy consumption of 27.6%. For execution time of multi-core policies, we observed speedups up to 10%, with the worst-case scenario showing a delay of 27.1%. For most applications the multi-core policies have a higher overhead in execution time. The applications react differently to every policy, even when the states look similar and looking at utilization can for some application be a solution.

Finally, this work proposes the following conceptual and technical contributions to the energy harvesting framework:

- Three new policies to the framework: Maximum Frequency, Ranked MinMax and Scaled MinMax.
- Single-core and many-core versions of all policies.
- A through evaluation of the framework on a set of 10 diverse workloads.
- Support for AMD Ryzen processors, for energy consumption reads.
- Support for GPUs that do not support *nvmlDeviceSetGpuLockedClocks*.

Table 6.1: Time and energy consumption for the best policy for energy harvesting per application

Applications	Best Policy					
	Single Core			Multi Core		
	Name	Energy	Time	Name	Energy	Time
BFS	Scaled MinMax	-0.5%	0.2%	Ranked MinMax	0.9%	1.2%
LavaMD	Maximum Frequency	-0.7%	-0.1%	MinMax	-6.6%	1.0%
NW	Ranked MinMax	4.8%	4.4%	Ranked MinMax	-7.9%	21.0%
Particlefilter-float	Ranked MinMax	-0.0	1.5%	Ranked MinMax *	-10.2%	14.8%
Kmeans	Ranked MinMax	3.7%	0.6%	Ranked MinMax	-3.8%	4.1%
Bandwidth	Maximum Frequency	-2.3%	0.1%	Maximum* Frequency	-2.7%	1.2%
UnifiedMemoryPerf	MinMax	-1.5%	-3.8%	Scaled MinMax	-16.2%	-2.8%
matrixMul	Maximum Frequency	3.5%	-0.0%	Maximum Frequency	8.5%	-0.2%
Jacobi unoptimized	MinMax	-3.5%	-7.4%	Maximum Frequency	-26.8%	-7.7%
Jacobi optimized	MinMax	-2.7%	-9.4%	Maximum Frequency	-34.8%	-10.0%

*This application has an policy that gets higher energy savings, but hurts the execution time more.

6.2 Limitations and threats to validity

All the results presented in this thesis are collected on a personal computer, because energy measurement and DVFS for the CPU and GPU require administrative OS rights. However, when using a personal computer, interference from other running tasks during measurement can be significant. To reduce the impact of such noise in our measurements, we run experiments multiple times, and present average and standard deviation measures. More details on the different challenges related to measurements are presented in Section 4.4. Additionally, all our gain/loss analysis for energy and execution time uses as reference the so-called NoAction policy. However, this policy is non-deterministic and can show variations, between runs, in both time and energy consumption. To alleviate both limitations, and provide more credibility to the presented numbers, more analysis needs to be done on more stable systems.

6.3 Future Work

Future work should focus first and foremost on collecting more data for different machines. It is important to get as much data as possible to make definitive conclusions. Also, there is room for researching more advanced energy-harvesting policies based on utilization, or policies that also take memory frequency into account. Additionally, our framework can provide a good environment for further research into building energy consumption models. Finally, these policies and models should be applied for much larger applications and computing systems.

Acronyms

CPU Central Processing Unit. 7, 9–14, 17–24, 27, 29, 32, 33, 36, 37, 40–42

CUDA Compute Unified Device Architecture. 11, 13, 14, 17, 18, 23, 24

DVFS Dynamic Voltage and Frequency Scaling. 8–10, 13, 42

GPU Graphics Processing Unit. 7, 9–14, 17–22, 24, 29, 32, 33, 36, 37, 41, 42

nvprof NVIDIA Profiler. 14, 17–24

Bibliography

[1] Q. Bakker. Saving energy in heterogeneous workloads. Master's thesis, Vrij Universiteit Amsterdam and Universiteit van Amsterdam, 2020. URL `https://gitlab.qub1.com/education/university/vrije-universiteit/master-project/thesis/-/blob/master/Source/Report.pdf`.

[2] Barcelona Supercomputing Center. EAR_team / EAR · GitLab. URL `https://gitlab.bsc.es/ear{_}team/ear`.

[3] J. Chen, B. Li, Y. Zhang, L. Peng, and J. K. Peir. Statistical GPU power analysis using tree-based methods. In *2011 International Green Computing Conference and Workshops, IGCC 2011*, 2011. ISBN 9781457712203. doi:10.1109/IGCC.2011.6008582.

[4] Docker. Docker. URL `https://docs.docker.com/get-docker/`.

[5] M. Gadou, S. R. Mogili, T. Banerjee, and S. Ranka. Multi-objective optimization on dvfs based hybrid systems. 2019 Tenth International Green and Sustainable Computing Conference (IGSC), 2019. doi:10.1109/IGSC48788.2019.8957181.

[6] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and performance characterization of computational kernels on the GPU. In *Proceedings - 2010 IEEE/ACM International Conference on Green Computing and Communications, GreenCom 2010, 2010 IEEE/ACM International Conference on Cyber, Physical and Social Computing, CPSCom 2010*, pages 221–228, 2010. ISBN 9780769543314. doi:10.1109/GreenCom-CPSCom.2010.143. URL `https://research.cs.vt.edu/synergy/pubs/papers/yang-perf-power-GPU-DVFS-greencom10.pdf`.

[7] T. Komoda, S. Hayashi, T. Nakada, S. Miwa, and H. Nakamura. Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping. International Conference on Computer Design (ICCD), 2013. doi:10.1109/ICCD.2013.6657064.

[8] X. Ma and L. Zhong. Statistical Power Consumption Analysis and Modeling for GPU-based Computing. *Proceedings of the SOSP Workshop on Power Aware Computing and Systems (HotPower '09)*, page None, 2009. URL `https://www.yecl.org/publications/ma09hotpower.pdfhttp://www.sigops.org/sosp/sosp09/hotpower.html`.

[9] X. Mei, L. S. Yung, K. Zhao, and X. Chu. A measurement study of gpu dvfs on energy conservation. *Proceedings of the Workshop on Power-Aware Computing and Systems*, pages 1–5, 2013. doi:10.1145/2525526.2525852.

[10] S. Mittal and J. S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys*, 47(4), 2015. ISSN 15577341. doi:10.1145/2788396. URL `https://dl.acm.org/doi/abs/10.1145/2788396`.

[11] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka. Statistical power modeling of GPU kernels using performance counters. In *2010 International Conference on Green Computing, Green Comp 2010*, pages 115–122, 2010. ISBN 9781424476138. doi:10.1109/GREENCOMP.2010.5598315. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.232.4758`.

[12] NVIDIA Corporation. Samples for cuda developers which demonstrates features in cuda toolkit, . URL `https://github.com/NVIDIA/cuda-samples`.

[13] NVIDIA Corporation. Cuda toolkit 10.1, . URL `https://developer.nvidia.com/cuda-10.1-download-archive-base`.

[14] NVIDIA Corporation. GeForce RTX 30-serie, . URL `https://www.nvidia.com/nl-nl/geforce/graphics-cards/30-series/`.

[15] Rodinia. Rodinia. URL `http://www.cs.virginia.edu/rodinia/doku.php`.

[16] E. L. Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. *Proceedings of the Workshop on Power-Aware Computing and Systems*, 2010. URL `https://www.usenix.org/legacy/events/hotpower/tech/full_papers/LeSueur.pdf`.

[17] TOP500. Green500 november 2020. URL `https://www.top500.org/lists/green500/2020/11/`.

[18] TOP500. TOP500 November 2020. URL `https://www.top500.org/lists/top500/2020/11/`.