Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Investigation of the energy consumption of different GPUs with respect to the used software stack

Erik Zeiske

Course of Study:	Informatik
Examiner:	Prof. Dr. Dirk Pflüger
Supervisor:	Marcel Breyer, Karlo Kraljic
Commenced:	September 29, 2021
Completed:	March 29, 2022

Abstract

Within the last couple of years, Graphical Processing Units (GPUs) have become one of the main drivers of the increase of the compute power of the TOP500 list. Next to High Performance Computing (HPC) use cases GPUs were also the main driver enabling the wide spread use of neural networks for machine learning. However, this increase of compute power comes at the price of an ever-increasing amount of energy consumption, with all the negative effects of associated environment and operational costs.

At the moment, there are 2 main vendors for HPC GPUs: Advanced Micro Devices (AMD) and NVIDIA. Both offer different software stack. This paper will discuss the impact of the choice of framework on the power draw and energy usage of the running kernels compared to the "native" software stack (i.e. CUDA and HIP for NVIDIA and AMD respectively). The goal is to gain an understanding whether the choice has a large enough impact on energy consumption to be significant for the choice of framework when aiming for green computing.

As matrix multiplication is a common operation for a fast array of work loads including machine learning a simple matrix multiplication kernel is used and rudimentarily optimized for this paper. The comparison explores the impact of single and double precision, explicit synchronisation, and shared memory.

The analysis has shown that whilst the choice of framework can have an impact of up to 15 % on the energy consumption, that difference often vanishes when compared to the savings caused by other factors that could easily reach 60 % e.g. for utilizing shared memory.

In den letzten Jahren sind graphische Recheneinheiten (GPU) ein starker Treiber für den Anstieg an Rechenleistung in der TOP500 gewesen. Des Weiteren sind GPUs auch einer der Hauptgründe für die breite Nutzung von neuronalen Netzen im maschinellen Lernen. Leider kommt dieser starke Anstieg in Rechenleistung mit einem erheblichen Anstieg an Energieverbrauch, mit allen seinen negativen Folgen für die Umwelt und die Kosten für den Betrieb.

Aktuell gibt es 2 große Anbieter von GPUs: AMD und NVIDIA. Beide bieten die Möglichkeit verschiedene Bibliotheken zu nutzen. Diese Arbeit diskutiert den Einfluss der gewählten Bibliothek auf die Leistungsaufnahme und Stromverbrauch der GPU im Vergleich zu der nativen Bibliotheken für die GPU (CUDA/HIP für NVIDIA/AMD). Das Ziel ist es eine Empfehlung auszuarbeiten, welche Bibliothek sich am meisten anbieten, um einen stromsparenden Informatik Betrieb zu ermöglichen.

Da Matrix Multiplikationen ein stets verwendetes Werkzeug sind in einer Vielzahl an Problemstellung wie zum Beispiel maschinelles Lernen, basiert diese Arbeit auf einem rudimentären Matrix-Multiplikation Kernel. Dabei wird der Einfluss von Fließkommapräzision, expliziter Synchronisation und geteiltem Speicher diskutiert.

Die Arbeit hat gezeigt, das es zwar definitiv einen Unterschied macht welche Bibliothek verwendet wird, aber die bis zu 15 % Unterschied die dadurch entstehen nicht wirklich relevant sind, wenn andere Faktoren, wie z. B. geteilter Speicher bis zu 60 % Energie sparen.

Contents

List of Figures

List of Listings

Acronyms

ACE	Asynchronous Compute Engine
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
API	Application Programmable Interface
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
CU	Compute Unit
DVFS	Dynamic Voltage and Frequency Scaling
FPGA	Field-programmable Gate Array
GCD	Graphics Compute Die
GeMM	General Matrix Multiplication
GPU	Graphical Processing Unit
GPC	GPU Processing Cluster
HPC	High Performance Computing
HBM	High Bandwidth Memory
NUMA	Non Uniform Memory Access
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor
TPC	Texture Processing Cluster

1 Introduction

Within the last couple of years, GPUs have become one of the main drivers of the increase of compute power represented by the TOP500 list. However, this increase of compute power comes at the price of a corresponding ever-increasing energy consumption, accompanied by negative effects with regard to the environment and operational costs [16].

At the moment, there are different vendors for HPC GPUs: AMD and NVIDIA (with Intel at the Horizon) with different architectures, software stacks and a whole range of different characteristics. As an answer to this heterogeneous environment a few frameworks have been developed, that allow the user to run code on all architectures currently in use. The main focus of this work is only the below 2 in addition to the "native" framework:

- OpenCl offers a standardized interface for all platforms that are able to process data in a highly parallel manner like modern Central Processing Units (CPUs) and GPUs [24].
- HIP, while being the native framework for programming AMD GPUs, is also intended for usage with CUDA [17].

This paper will discuss the impact of the choice of framework on the power draw and energy usage of the running kernels compared to the "native" software stack (i.e. CUDA and HIP for NVIDIA and AMD respectively). The goal is to gain an understanding whether the choice has a large enough impact on energy consumption to be significant for the choice of framework when aiming for green computing.

Due to the limited time frame of this work the paper will not try to fully understand the exact reasons for the possible differences in energy performance of the frameworks. The more important goal is to spot these differences and possibly give a good foundation to further investigation later on.

Structure of the remainder of this paper:

- Chapter 2 discusses the most important technical details regarding commonly utilized hardware architecture, framework architecture and statistical methods.
- Chapter 3 presents the literature discussing surrounding topics to the issues handled in this thesis.
- Chapter 4 explains the experimental setup used for gathering the data in discussion in this paper. It covers the algorithms, data gathering methods and GPUs used.
- Chapter 5 illustrates the most important differences between the frameworks and algorithms used in terms of their average power draw, execution time and total energy consumed.
- Chapter 6 concludes the thesis by restating the most important observations and conclusions drawn from the data.

2 State of Technology

This chapter covers the general architecture of the GPUs used. Additionally, some important statistical and mathematical methods needed for the later analysis are discussed.

2.1 GPU Architecture

To start this chapter this section presents an overview of the layout and architecture of the used GPUs in this work.

2.1.1 NVIDIA

The NVIDIA architecture follows two main hierarchies: a memory and a computational hierarchy. The latter consists of GPU Processing Clusters (GPCs) at the top-level, and can be thought of a division of the GPU into smaller manageable units that enable the easier scalability of the GPU architecture. Each GPC is further divided into seven or eight Texture Processing Clusters (TPCs) that manage the workload distribution for their two Streaming Multiprocessors (SMs).

Each SM further subdivides into four parallel Compute Units (CUs) which are each able to execute one warp at the time. A warp consists of at most 32 threads and is executed in a Single Instruction Multiple Threads (SIMT) manner. This means that an instruction is executed in parallel on all threads in the CU on their respective data. In addition to Single Instruction Multiple Data (SIMD) it is possible that threads in the same warp diverge in their control flow e.g. due to an if-else structure. This divergence will result in a sequential execution of all possible control flow paths, whilst masking the threads not in the currently executed path. Because of this behaviour, highly diverging programs will suffer poor performance [28].

On the memory site, the highest data level is the High Bandwidth Memory (HBM) accessible by all parts of the GPU. The next level consists of an also shared L2 cache. However, the cache is partitioned such that each GPC is having a dedicated L2 cache. Whilst the partitions not associated with the current GPC are still accessible this cross access is slightly slower in responding to memory accesses, this is considered a Non Uniform Memory Access (NUMA) architecture [28].

The next cache level is on the SM level and is no longer shared among all warps. Additionally, it doubles as a shared memory between warps scheduled on the same SM. Such allocations can be achieved by the `__shared__` keyword [28].

2.1.2 AMD

AMD follows a similar structure as NVIDIA for subdividing the GPU. However, the terms used and numbers vary on each level. At a top-level the GPU is divided into several Graphics Compute Dies (GCDs), which are separate silicon dies connected via a bus for communication. Each GCD is further subdivided into 2-4 Asynchronous Compute Engines (ACEs) with 112 or 120 CU in total divided equally among them. Each CU is the equivalent of an NVIDIA CU with the noticeable exception that 64 threads are used instead of 32 [2].

2.1.3 Streams and Events

To give the user control over which kernels, commands and tasks to run in which order, both architectures employ so-called streams and events.

A stream consists of an ordered sequence of kernels, commands, tasks and events that are executed in the order they were appended to the stream. However, it is only guaranteed that the parts of the stream are scheduled in the order they were appended. Unless otherwise specified, the stream does not have to wait on the completion of one task before scheduling the next [12]. However, it seems to be the consensus that kernels issued within one stream will be executed sequentially without any overlap [7, 31].

By default, all functions will scope to the default stream of the GPU but the user is able to create custom streams for more complex behaviour if needed [12].

An event can be used to achieve forced ordering as it will be considered complete only if all parts of the stream before the event are complete, thus allowing the stream to wait on a specific event. Optionally the events can be used to track the point in time the completion has occurred, allowing for an easy way to time kernels [12].

Events can not only be referenced by the scope they are in, but can be used to synchronize multiple streams or reflect dependencies between kernels or data copies of two separate streams [12].

One example of the usage of events is the ability to wait on the complete execution of a kernel before executing the next kernel. This can be seen in Listing 2.1.

```
1  CUevent last_event = nullptr, new_event = nullptr;
2  while (true) {
3    // Kernel acting as load
4    cuEventCreate(&new_event, CU_EVENT_DEFAULT);
5    cuEventRecord(new_event, 0);
6    cuStreamWaitEvent(0, new_event, 0);
7    if(last_event) {  // wait for the event of the previous loop
8      cuEventSynchronize(last_event);
9      cuEventDestroy(last_event);
10   }
11   last_event = new_event;
12 }
```

Listing 2.1: Forced linear execution for CUDA kernels

In Listing 2.1, after each load kernel an event is created (L. 4), which is associated with the current content of the default stream (L. 5). To ensure the previously scheduled events are finished before the next kernel, the stream is instructed to wait on the newly created event (L. 6). Then the event created in the previous iteration is used to synchronize the CPU (L. 7-9). This reduces the number of kernels that are pending in the stream. It is important that the `last_event` is used here to hide the latency of CPU-GPU interaction in the execution of the load. Lastly the `last_event` is moved to the newly created event (L. 11) for usage in the next iteration.

The same can be achieved for HIP by replacing the CUDA functions with their HIP equivalent [4].

For OpenCl the code can be written differently as it is possible to directly attach an event to the execution of a kernel and let the kernel wait on events in the function call itself. This results in the code shown in Listing 2.2 [24].

```
1  cl_event last_event = nullptr, new_event = nullptr;
2  while (true) {
3    cl_event * wait_for = nullptr; cl_int wait_for_int = 0;
4    if(last_event) {
5      wait_for = &last_event;
6      wait_for_int = 1;
7    }
8    clEnqueueNDRangeKernel(command_queue, kernel, /* range params */, wait_for_int, wait_for, &
     new_event);
9    if(last_event) {
10     clWaitForEvents(wait_for_int, wait_for);
11     clReleaseEvent(last_event);
12   }
13   last_event = new_event;
14   new_event = nullptr;
15 }
```

Listing 2.2: Forced linear execution for OpenCl kernels

In Listing 2.2, depending on whether a previous event already exists, the waiting list is set up (LL. 4-7) and then passed to the kernel (L.8) and waited upon before the next kernel is launched (LL. 9-12). Finally, the event created by the kernel is switched to the `last_event` to be ready for the next iteration.

2.1.4 Grid execution

As GPUs are intended for usage on highly parallel work loads that are often organized in a grid, it is sometimes necessary to access data from a thread that is next to the current thread in a 1-D, 2-D or 3-D grid. Within this grid the threads are further grouped into a coarser grouping called work-groups in OpenCl and blocks in CUDA and HIP [4, 12, 24].

All threads within one work group will execute in same compute unit, allowing them to share local shared memory. This limits the number of threads per work group to 1024 for NVIDIA and typically 1024 or larger for AMD [4, 12, 24].

In order to locate the thread within the grid it is possible to retrieve the global thread id in OpenCl via `get_global_id()` and the id within the work group by `get_local_id()` [24]. In HIP and CUDA the global id needs to be computed using the local id and block id using the constants `threadIdx` and `blockIdx` [4, 12].

2.2 Basic Linear Algebra Subprograms (BLAS)

Having covered the main points of the architecture and frameworks in use in this paper this section will cover the typical libraries in use for matrix multiplications. Sadly these could only be used on one system due to the lag of administrator privileges on the other. As later described this resulted in the need for a naive implementation of a matrix multiplication that is generally used throughout the paper.

Basic Linear Algebra Subprograms (BLAS) describes a collection of functions around matrix and vector operations [6]. In this section the General Matrix Multiplication (GeMM) routine will be discussed.

The GeMM routine corresponds to a general matrix multiplication of the following form [11, 15]:

$$C \leftarrow \alpha AB + \beta C \tag{2.1}$$

Here A,B and C are matrices and α and β scalar values. To be complete it should be mentioned that the matrix A and B can be transformed before the multiplication e.g. transposed, but as this won't be used in this paper an explanation of the details will be skipped. After the transformation, the matrix dimensions need to line up, such that a matrix multiplication is possible.

```
1 hipblasStatus_t hipblasSgemm(hipblasHandle_t handle, hipblasOperation_t transA,
  hipblasOperation_t transB, int m, int n, int k, const float*alpha, const float* A, int lda,
  const float* B, int ldb, const float*beta, float*C, int ldc)
2 cublasStatus_t cublasSgemm(cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t
  transb, int m, int n, int k, const float *alpha, const float *A, int lda, const float *B, int
  ldb, const float *beta, float *C, int ldc)
```

Listing 2.3: Definition of GeMM routines for CUDA and HIP

Both HIP and CUDA follow the same structure in their function call allowing for a combined explanation of the function parameters [11]:

- `handle` is an opaque handle holding information about the current state of the BLAS library. It is tied to the specific GPU in use and implicitly specifies the current stream used to schedule the operation.
- `transa, transb` represent the operation to perform on the matrix A and B. For this paper these will be set to `CUBLAS_OP_N` and `HIPBLAS_OP_N` respectively.
- `m, n, k` dimensions describing the matrices in order: rows of A, columns of B and rows/cols of A/B.

- `alpha, beta` the scalar values of α and β.
- `*A, *B, *C` device pointers to the matrices
- `lda, ldb, ldc` The size of the leading dimension of the two-dimensional array storing the respective matrix.

The S in the function name refers to the fact that single precision floating-point numbers are used. To use the double precision version of the function the S needs to be replaced with a D.

2.3 Dynamic Voltage and Frequency Scaling (DVFS)

As energy consumption can also be influenced by the choice of frequency and voltage of the compute unit, this section will discuss the fundamentals of power draw and Dynamic Voltage and Frequency Scaling (DVFS).

In general the power draw of a processing unit can be described by the following formula:

$$P(total) = P(static) + P(dynamic) \tag{2.2}$$

Here the static power describes the power draw that is independent of the workflow and results from the processing unit being in a non-idle state [20, 23]. However, the power draw can still vary due to external factors such as temperature. As the real physical relationship of power draw with respect to temperature is quite complicated, a linear relationship is assumed for this paper as mentioned in [18]

On the other hand, dynamic power is mainly defined by the voltage and frequency used to switch the transistors within the processing unit and follows the following general law [23]:

$$P(dynamic)\ \alpha\ Voltage^2 \times Frequency \tag{2.3}$$

Thus, the dynamic power draw is the main object of DVFS, which is a standard technique to reduce the amount of power that is drawn by adapting the voltage and frequency used by a processing unit [23].

2.4 Statistical testing methods

The goal of this work is to spot differences between different frameworks in terms of their use of energy, time and power draw. As all these values are probably impacted by values that are not fully controllable, e.g. slight fluctuations in temperature, the measured values should be considered as random variables. Therefor this section will describe the statistical test that will be used in Chapter 5 to differentiate between different measurements.

To achieve such a comparison, statistical tests are used that consider some sort of statistical distribution for the random variables under comparison and determine the probability that the measured values are from the same underlying random distribution.

As it is hard to know the distribution the measured data will follow, it is assumed that all measured data follows a normal distribution as the statistics of it are best understood, especially compared to more complex distributions.

A standard distribution is defined by its mean ($\bar{X}_i$) and standard deviation (s_i). As neither is known before the measurement, it must be assumed that the measured standard deviation is that of the underlying distribution.

In order to compare two measured distributions, the question needs to be answered whether the deviation in their means ($\bar{X}_1$, $\bar{X}_2$) is likely considering the standard deviation of the distribution describing the difference between the two means ($\Delta\bar{X}$). As it is not given that the standard deviation of the distributions under scrutiny are the same, the test used in this work is the Welch's t-test.

In the Welch's t-test the standard deviation of the distribution of differences between the two means of the random samples is estimated by using the following equation [35]:

$$s_{\Delta\bar{X}} = \sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}} \tag{2.4}$$

Given the number of samples in the respective samples tested (N_1, N_2), it is further assumed that the differences between the two means follows a t-distribution with the following amount of degrees of freedom v, assuming that variance estimation has $(N_i - 1)$ degrees of freedom [35]:

$$v = \frac{\left(\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}\right)^2}{\frac{s_1^4}{N_1^2(N_1-1)} + \frac{s_2^4}{N_2^2(N_2)}} \tag{2.5}$$

Putting this together, the t value can be estimated by Equation (2.6) and then used to calculate the probability that a bigger variation than the observed is measured. This is achieved by calculating the probability a t value that is absolute bigger than the calculated is reached given a t-distribution with v degrees of freedom [35].

$$t = \frac{\Delta\bar{X}}{s_{\Delta\bar{X}}} \tag{2.6}$$

This is also called the two-tailed test as it is the probability added that the deviation bigger in its absolute value. As there is no previous notion in this paper on what value will be bigger than the other, the two-tailed test is always used to calculate the p value.

2.5 Discrete integrals to calculate the total energy consumption

After discussion of the statistical methods needed for this work, this section will discuss the means to determine the main metric of this work: energy consumption. Sadly as discussed in Chapter 4 the only metric that could be gathered for this work is the power draw at a specific point in time ($P(t)$). Assuming a continuous function describing the power draw over time, this information can be used to calculate the energy consumption over that time period using Equation (2.7).

$$E(total) = \int_{t_s}^{t_e} P(t)\, dt \tag{2.7}$$

However, the Application Programmable Interfaces (APIs) return only a power draw for discrete points in time. Assuming these points as a time-ordered sequence with N points, each can be described using the time $T(i)$ and power draw $P(i)$ of a specific point $i \in [1, N]$. For convenience, it is assumed that the time window between two measures consists of a linear transition between the two power draws as shown in Figure 2.1 as a solid line.

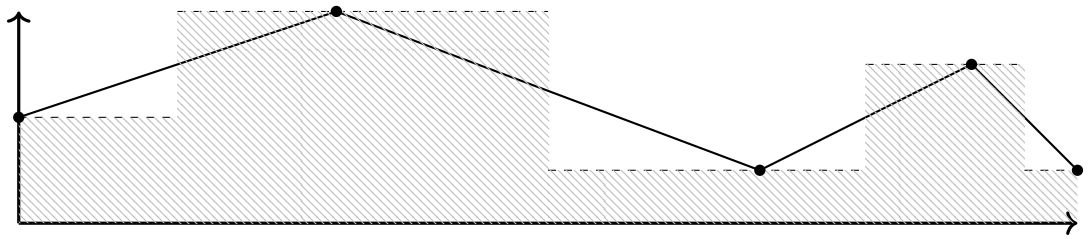

Figure 2.1: Visualisation of area calculation for discrete integral of the power draw to determine energy consumption

Calculating the integral for the solid line is equal to calculating the area of the hatched area where the jump points between two power draw samples are in the exact middle between them. This results in the calculation of the total energy consumption $E(total)$ as shown in Equation (2.8).

$$\begin{aligned} E(total) &= \sum_{i=1}^{N} P(i) \left(\frac{T(i) - T(i-1)}{2} + \frac{T(i+1) - T(i)}{2} \right) \\ &= \sum_{i=1}^{N} P(i) \left(\frac{T(i+1) - T(i-1)}{2} \right) \end{aligned} \tag{2.8}$$

Note: For the border cases, simply setting $T(-1) := T(0)$ and $T(n+1) := T(n)$ will result in the desired behaviour.

3 Related Work

There are several papers that discuss the energy consumption characteristics [10, 19, 21] of various algorithms. In addition to discussing the difference between algorithms, the impact of DVFS on power consumption is discussed [10, 21].

DVFS has been a widely discussed topic in literature as well. It is mostly focused on CPUs [9, 14, 20, 33], but also discussed in relation to GPU systems [23]. Here, the general conclusion seems to be that a slight reduction in voltage or execution frequency results in a slight increase in execution time, but often has a higher effect in reducing the energy consumption [9].

When looking at DVFS, there are also studies that change multiple frequency parameters such as the memory frequency and core frequency/voltage. This seems to have varying results on different GPU generations [23].

Besides purely empirical approaches, a few models were developed that predict the energy consumption based on performance counters. Here, the counters were used to train various machine learning models to predict the power consumption when presented with a new set of counters [8]. Additionally, there exists a discussion of various physical measures such as temperature and amount of cores used to predict and ultimately optimize total energy consumption [18]. Lastly, [1] explores the possibility to predict power consumption based on intermediate code representation.

At the other end of the spectrum of exploration, [29] explores the ability to simulate power consumption using a purely mathematical model.

[36] discusses the difference between performance of CPUs, GPUs and Field-programmable Gate Arrays (FPGAs) when operating on BLAS 3 routines. While the work has shown that the FPGA has the highest performance for small scale problems, power consumption is not discussed in the paper. These aspects were discussed for a wide range of computer vision kernels in [30] showing that simple kernels have a high energy efficiency on GPUs, while an increase in complexity tends to favor FPGAs in terms of energy consumption.

With regard to nonuniform work loads with a variety of kernels, it is also possible to utilize efficient concurrent scheduling to reduce overall power draw [22].

In terms of comparing the performance of different frameworks there was also some work done. Depending on the goal of the research paper this has different result. On the one hand when trying to implement the same algorithm CUDA is in general better than OpenCl as platform specific optimisations can be used [32]. On the other hand when not using architecture specific optimisations both perform similar [13].

In general, the literature seems to only explore NVIDIA even in wide scale studies [8]. While there is some literature that focuses on AMD, it is not vast or else it covers consumer GPUs [34].

4 Experimental Setup

After showing the main theoretical points needed for this paper, this section will show the hardware and software used for running and measuring the tests used in this paper.

4.1 Hardware

The following hardware is used in all the following tests.

Host	gilgamesh	argon-tesla1	instinct
Kernel	4.18.0-348.7.1.el8_5 x86_64	5.4.0-100-generic	5.11.0-38-generic
Operating System	RHEL 8.5	Ubuntu 20.04.4 LTS	Ubuntu 20.04.3
CUDA version	11.1	11.1	-
HIP version	4.4	-	4.4
GCC version	8.5.0	9.4.0	9.4.0
GPU	1 x NVIDIA A100-PCIE-40GB[27]	2 x NVIDIA Tesla P100-PCIE-16GB[25]	2 x AMD Instinct MI100[3]
Double precision (TFLOPS)	9.7	4.7	11.5
Single precision (TFLOPS)	19.5	9.3	23.1
Memory Speed (GB/s)	1555	732	1200
Max Power Consumption (W)	250	250	300
Release year	2021	2016	2020

The hardware used for a specific run is marked in the figures by its host name. The OpenCl version is always the version offered by CUDA and HIP depending on the main platform the host is using. gilgamesh and instinct are standalone systems that are not behind any scheduling system. argon-fs on the other hand is using SLURM to ensure exclusive access.

4.2 Data Collection

In order to collect the physical characteristics in parallel for each benchmark, an independent process is used. In general this daemon collects the power data/temperature/... and stores it with a timestamp accurate in µs to a file. This is later matched with the benchmark by storing the start and

end timestamp for each benchmark in another file and considering all data points in between the two marks as belonging to the benchmark. It was not possible to classify the impact of this daemon on the overall performance of the GPU as this would require an external measurement of the power draw. However, as all measurements are faced with the same daemon, this should not influence the comparability of the results within this work.

If not otherwise stated in the paper there were warmup runs performed to ensure the GPU is in a thermally stable configuration before each measurement.

Depending on the manufacturer, different technologies are used.

4.2.1 NVIDIA

NVIDIA provides the tool `nvidia-smi`, which allows the background querying of the relevant physical properties such as temperature and power draw by running it with the following flags: `stats -f $FILE`. This results in a continuous output of data to a file divided into lines with each line containing the 0-based id of the graphic card it belongs to, which is relevant if multiple NVIDIA graphic cards are present in order to determine the corresponding one. Furthermore, the timestamp in µs, the type of data captured and the actual data point is stored per line. For the relevant values the following table shows the corresponding type and accuracy.

Description [26]	Key	Accuracy
Temperature	`Temp`	1 K
Current power draw	`pwrDraw`	1 W

4.2.2 AMD

On AMD systems, a custom daemon was written to accomplish the same functionality as `nvidia-smi`. Here the library `rocm-smi` was used with the following functions [5]:

- `rsmi_dev_power_ave_get` determines the power draw in mW. However, for the graphics card used in this paper, the granularity of the values returned by this function turned out to be only 1 W having 3 trailing zeros all the time.
- `rsmi_dev_temp_metric_get` for getting the temperature of the graphics card. As there are multiple locations to retrieve the temperature from the “Junction/hotspot temperature” was used, assuming it gives a fairly average values for the GPU.
- `rsmi_dev_energy_count_get` should be usable to retrieve an accurate accumulated energy consumption since the last reset of the GPU. However, for the used GPUs the values returned by this function were constant over time. This suggests a dummy implementation of the function which is not usable for the purpose of data gathering.

As all functions don’t return a time point for which their value is valid the current time was queried before each function call to use as the timestamp of the retrieved values.

4.3 Tests performed

This section will show the methods and code used for the individual measurements performed for this paper.

In general all tests performed are conducted using a kernel that is repeatedly scheduled to stretch the total execution time to a reasonable value into the arbitrary bound of 1 to 200 s. Such a repeating scheduling is considered an individual run. To eliminate the possibility of momentary disturbances during the run, skewing the comparison, each run is performed at least five individual times. These runs are interleaved with runs form other frameworks and the parameter under comparison. This especially means, if a set of parameters is used twice in separate comparisons the individual runs were performed for each comparison to rule out long term variations between the measurements.

The execution time and energy consumption is always associated to a complete run. As the power draw is measured in a time series parallel to each run, the average power draw over a set of runs is always calculated based on all data samples gathered during all runs grouped for the average.

Unless stated otherwise the repeated scheduling of the kernel is wrapped by the code shown in Listing 2.1 and Listing 2.2 to force linear execution.

As HIP and CUDA are extremely similar in their code structure and function calls a simple script to replace the CUDA functions with their HIP counterpart was used. This allows for the implementation of everything in CUDA and an automatic adaption to HIP. This was not possible for OpenCl. Therefore, a separate implementation for OpenCl is used and the differences will be pointed out explicitly for this section while HIP is not mentioned. Even though OpenCl has a different code base it should still be comparable with the other frameworks as the differences are only minimal.

4.3.1 Dynamic Voltage and Frequency Scaling (DVFS)

To start the investigation, a general analysis of the effect of compute frequency and matrix size on power draw is conducted. For this the manufacturer-provided BLAS routine as described in Section 2.2 is used.

To achieve the repeated scheduling of the same kernel the memory of the GPU is divided into the matrix amount that will fit on it and calling the kernel repeatedly while striding over the matrices A/B allocated, leaving C fixed. This results in the source code shown in Listing 4.1. Here the correct matrix is simply passed to the kernel by adding to the base pointer of the matrix buffer and offsetting the pointer by the matrix memory footprint times `i` (LL.4-5). While the `repetition_count` refers to the amount of passed done over the memory, the `iterations` reflect the available amount of matrices.

```
1 for (size_t j = 0; j < repetition_count; j++) {
2   for (size_t i = 0; i < iterations; i++) {
3     cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, matrix_size, matrix_size, matrix_size, alpha
  ,
4             (const double *)a_data_ptr + i * matrix_size * matrix_size, matrix_size,
5             (const double *)b_data_ptr + i * matrix_size * matrix_size, matrix_size,
6             beta, (double *)c_data_ptr, matrix_size);
7   }
```

```
8 }
```

Listing 4.1: Stride implementation for repeating kernel GeMM calls

4.3.2 Matrix Multiplication Kernel

As the usage of the BLAS libraries turned out to be impossible on any other system and framework then argon-fs using CUDA, a simple matrix multiplication kernel is used for the rest of the tests performed. In order to mimic the behaviour of the BLAS implementation of GeMM, the kernel shown in Listing 4.2 is used.

```
1  #define BLOCK_SIZE 16
2  template<typename T> __global__ void gemm_kernel(const T* A, const T* B,
3      T* C, T alpha, T beta, int m, int n, int k) {
4    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
5    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
6    if (row < m && col < n) {
7      T sum = 0;
8      for (int i = 0; i < k; i++) {
9        sum += A[row * k + i] * B[i * n + col];
10     }
11     C[row * n + col] = beta * C[row * n + col] + alpha * sum;
12   }
13 }
14 const dim3 local(BLOCK_SIZE, BLOCK_SIZE);
15 template<typename T> void gemm(bool, bool, int m, int n, int k,
16   T *alpha, T *A, int, const T *B, int, T beta, T *C, int) {
17   dim3 global(n / BLOCK_SIZE, m / BLOCK_SIZE); // Assuming n and m % BLOCK_SIZE == 0
18   gemm_kernel<<<global, local>>>(A, B, C, *alpha, beta, m, n, k);
19 }
```

Listing 4.2: Naive matrix multiplication kernel implemented in cuda

Listing 4.2 gives each thread the responsibility to build the vector product of its respective row of matrix A (x dimension) and column of matrix B (y dimension). This is achieved in the inner heart of the kernel (LL. 7-10). The result of the dot product stored in `sum` is then further used to calculate the result for the cell with the respective GeMM formula (L. 11).

Here the allocation of the respective row and column a thread should use is achieved by scheduling them in a 2D grid with a coarse `global` grid consisting of smaller 16x16 `local` grids. This allows for the later optimisations regarding shared memory. The size of 16 is used as it works over all frameworks and GPU architectures and there is no intend in discussing the impact of group size on the energy consumption.

The global grid size is calculated assuming n and m are divisible by the `BLOCK_SIZE` (L. 17) which is guaranteed by the fact that the kernel is only executed with matrix sizes being a power of 2 starting at 64.

In order to cut down on complexity the leading dimension parameters and matrix transformation parameters are ignored, leaving them without a variable name in the function definition (LL. 15-16). This can be done as it is assumed that the size of the leading dimension of the matrix is equal to the respective m, n, k parameters.

The main differences for the OpenCl code are the fact that there was no usage of the template type such that a separate implementation for single (`float`) and double precision (`double`) data types was written. Additionally, the calculation of the `row` and `col` index is replaced with the respective OpenCl function. Also, the global dimensions used to call the kernel are not divided bey the `BLOCK_SIZE` as this is not necessary for OpenCl.

4.3.3 No explicit synchronisation between kernels

As discussed in Section 2.1.3, it is not clear whether it is necessary to use events to force a linear execution of kernels. As the matrix multiplication was up to now executed with forced sequentiality, this section will look into the differences in power draw, time taken and total energy consumption when not using events to forcefully synchronise between kernel calls. In particular this means that the kernels are simple queued without any calls in-between.

```
1 CUDA_SAFE_CALL(cudaDeviceSynchronize(), device_index);
2 gettimeofday(start_time, NULL);
3 // Schedule all kernels
4 CUDA_SAFE_CALL(cudaDeviceSynchronize(), device_index);
5 gettimeofday(end_time, NULL);
```

Listing 4.3: Scheduling of all kernels without any synchronisation for CUDA

The code for this is shown for CUDA in Listing 4.3. The HIP and OpenCl use `hipDeviceSynchronize` and `clFinish` instead of `cudaDeviceSynchronize` respectively. The time to consider as start and end time is taken after the queue clear to ensure that no kernel is scheduled before the start of the measurement and all kernels are finished before the end.

4.3.4 Shared memory matrix multiplication

The final test will be performed utilizing the shared memory offered to warps in the same SM. The implementation used for CUDA is shown in Listing 4.4.

```
1  template<typename T> __global__ void gemm_shared_kernel(const T* A, const T* B, T* C, T alpha,
    T beta, int m, int n, int k) {
2      int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
3      int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
4      __shared__ T l_A[BLOCK_SIZE][BLOCK_SIZE];
5      __shared__ T l_B[BLOCK_SIZE][BLOCK_SIZE];
6      if (row < m && col < n) {
7          T sum = 0;
8          for (int gr_i = 0; gr_i < k; gr_i+= BLOCK_SIZE) {
9              l_A[threadIdx.y][threadIdx.x] = A[row * k + gr_i + threadIdx.x];
10             l_B[threadIdx.y][threadIdx.x] = B[(gr_i + threadIdx.y) * n + col];
```

```
11              __syncthreads();
12              for(int i = 0; i < BLOCK_SIZE; i++) {
13                  sum += l_A[threadIdx.y][i] * l_B[i][threadIdx.x];
14              }
15              __syncthreads();
16          }
17          C[row * n + col] = beta * C[row * n + col] + alpha * sum;
18      }
19  }
```

Listing 4.4: Shared memory GeMM implementation for CUDA

The main difference of this kernel is a separation of fetching the data and calculating. The local block is sharing 2 memory regions (L.4-5) that are caching a sub-block of their respective matrix at a time. This is done by letting each thread of the block fetching exactly one value and storing it in the cache for the other threads to use (L.9-10). Then each kernel is adding the dot product of its respective rows and columns of the cached matrix. To ensure that all reads are through and visible an explicit synchronisation is used (L.11) and then repeated after the calculation to ensure no subsequent read is already overwriting the local cache (L.15).

To adapt this code to OpenCl the obvious replacement for the local thread id and an explicit use of the datatype need to make. Additionally, the shared memory needs to be declared as `__local` instead.

5 Results

5.1 Discussion of Dynamic Voltage and Frequency Scaling (DVFS)

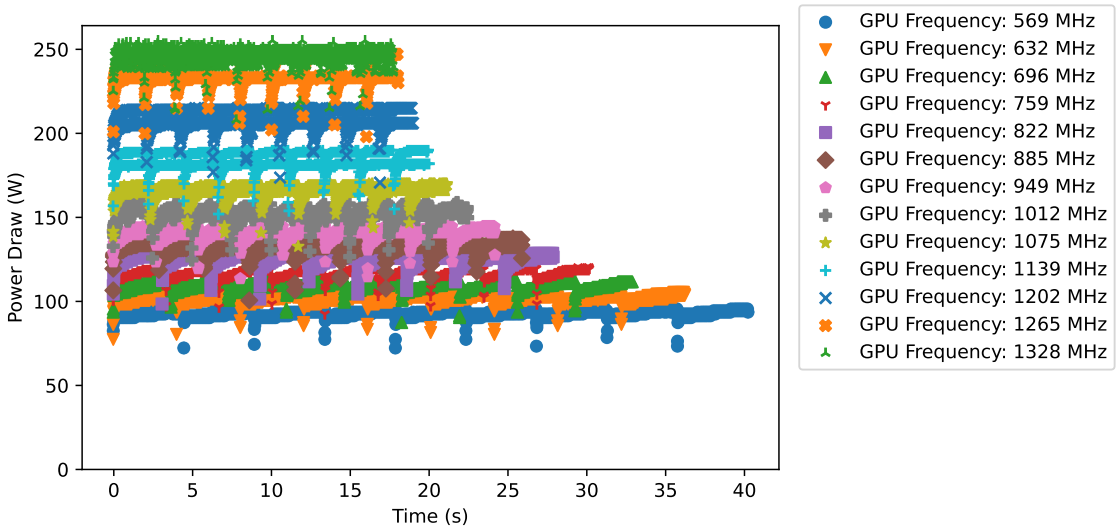

Figure 5.1: NVIDIA, argon-fs: 9 consecutive Matrix multiplication kernel for various GPU frequencies. Scatter plot power draw over time.

As described in Section 4.3.1 the investigation starts with a short analysis of the frequency of a GPU on the power draw and energy consumption. In order to gain an initial understanding of the kernel's behaviour, the power consumption over the time of a run execution is discussed.

Figure 5.1 shows a fairly stable power load on the GPUs, even though it is noticeable that there are nine drops in power consumption. Depending on the frequencies these drops range from 20% to 25% from the maximum power draw in a specific run. The maximum power draw is 255 W when using a frequency of 1328 MHz. The lowest power draw occurs for the lowest frequency of 569 MHz and is 73 W. Beyond the values of power draw it can also be clearly seen that with decreasing frequency the execution time increases monotonic with decreasing frequency from 17.5 s at the highest to 40.2 s at the lowest frequency.

The nine drops in energy consumption could be reasonably correlated to the switch between two matrix multiplications as there are also nine multiplications run in total. This could mean that the switch between two multiplications is not utilizing the complete GPU. However, as the consecutive execution between the kernels is already implemented at the GPU queue level, there doesn't seem to be a reasonable way to reduce this drop in power draw between kernels.

Going forward, this initial experiment can be used to verify the assumptions about DVFS. As there is a constant voltage, a linear relation between power draw and frequency is expected. Additionally, it stands to reason that smaller matrix sizes should result in a lower or equal power draw due to less memory and fewer compute resources used.

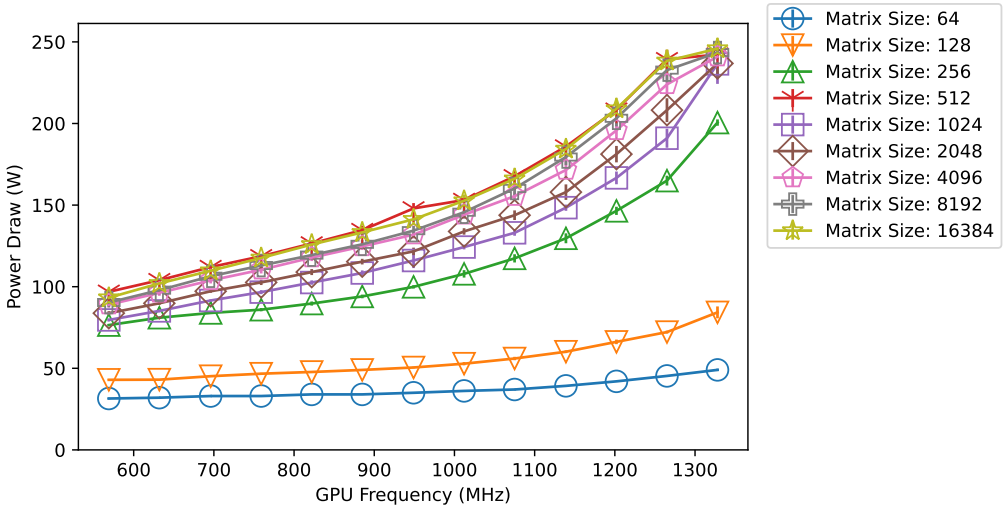

Figure 5.2: NVIDIA, argon-fs: Average power draw over Matrix multiplication kernel plotted over GPU frequency using different matrix sizes

Figure 5.2 shows the average power draw of the matrix multiplication kernel for various frequencies and matrix sizes. The power draw is averaged over all data samples disregarding the fact that they are not sampled in a time uniform manner. This was done for ease of calculation. The biggest matrix size is the same data as shown in Figure 5.1 meaning the largest power draw is still 255 W. This is more than the theoretical maximum of 250 W for this GPU, suggesting that the limit is not enforced well or the measurement circuit has a systematic error. Regardless of that this also explains why the graph seems to fall off at the top as there is not much leeway in getting a higher power draw for the continuation of the trend. On the other side of the spectrum the lowest power draw of 31 W can be observed at the lowest frequency and smallest matrix size.

The separation of the smallest matrix sizes from the other power draw curves suggests that these are not utilizing the complete GPU and thus not reaching the limits of the other matrix size. The fact that the other matrix sizes are stacking up at a seeming limit suggests that they reach the theoretical maximum for this specific core frequency.

This shows that the qualitative assessment of increase in frequency and matrix size results in an increase in power draw is correct, but the quantitative relationship doesn't seem to be linear. As this contradicts several papers [20, 21] there should be a reasonable explanation for this non-linear relationship. As [18] indicates, the temperature might have an impact on overall power draw.

And as Figure 5.3 shows there is a difference in temperature between different frequencies. The temperature is increasing with the matrix size and GPU frequency. With the lowest being at 40 °C and the highest at 64 °C. Notably the in run range of temperature is always at 1 K with one exception at the biggest matrix size and frequency spanning 4 K. This could be explained with

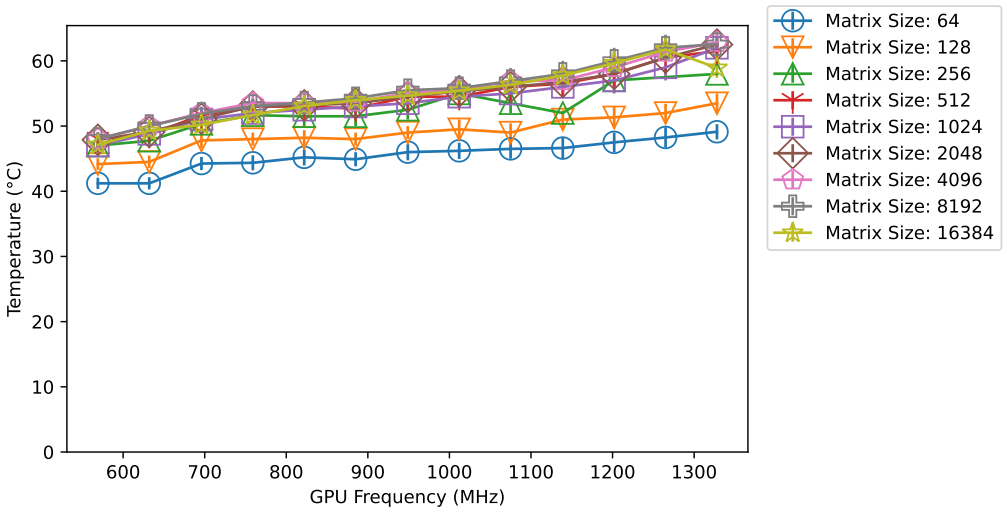

Figure 5.3: NVIDIA, argon-fs: Average temperature during Matrix multiplication kernel plotted over GPU frequency using different matrix sizes

the fact that the increase in power draw is causing an increase in the Temperature due to raising the stable thermal equilibrium. This increase in Temperate intern increases the power draw until a stable configuration is reached on both count.

As shown in Figure 5.4 DVFS can also be used for saving energy consumption as the highest compute frequency is not necessarily the best in terms of energy. In case of the biggest matrix size it is even the worst with 217 J. By using the best frequency for this particular matrix size 949 MHz saves 22 %.

5.2 Naive Matrix Multiplication

After a small excursion into DVFS this section will start with comparing the power draw of a simple matrix multiplication kernel on NVIDIA and AMD GPU using different software stacks.

As there were several issues trying to compile the BLAS implementation for HIP and OpenCl, a simple matrix multiplication kernel (see Listing 4.2) is used for the purpose of the discussion in the rest of the paper.

5.2.1 NVIDIA

This section will start with the discussion of the first NVIDIA results on gilgamesh going through the power draw, execution time and total energy consumption.

The power draw for different runs of the matrix multiplication kernel is shown in Figure 5.5.

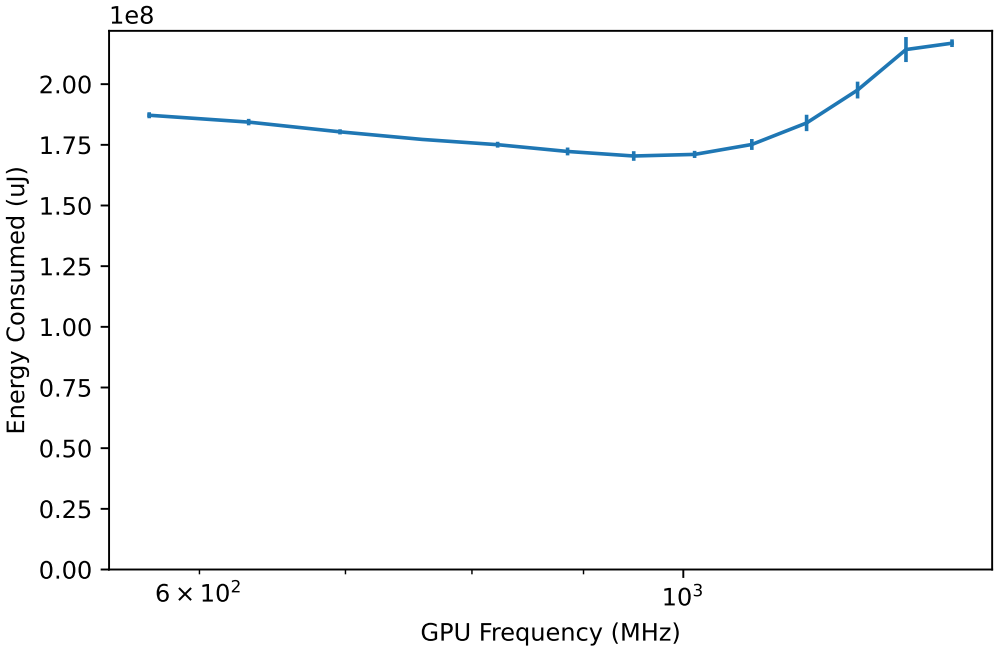

Figure 5.4: NVIDIA, argon-fs: Energy consumption for a single Matrix multiplication of a 16384x16384 matrix plotted over GPU frequency

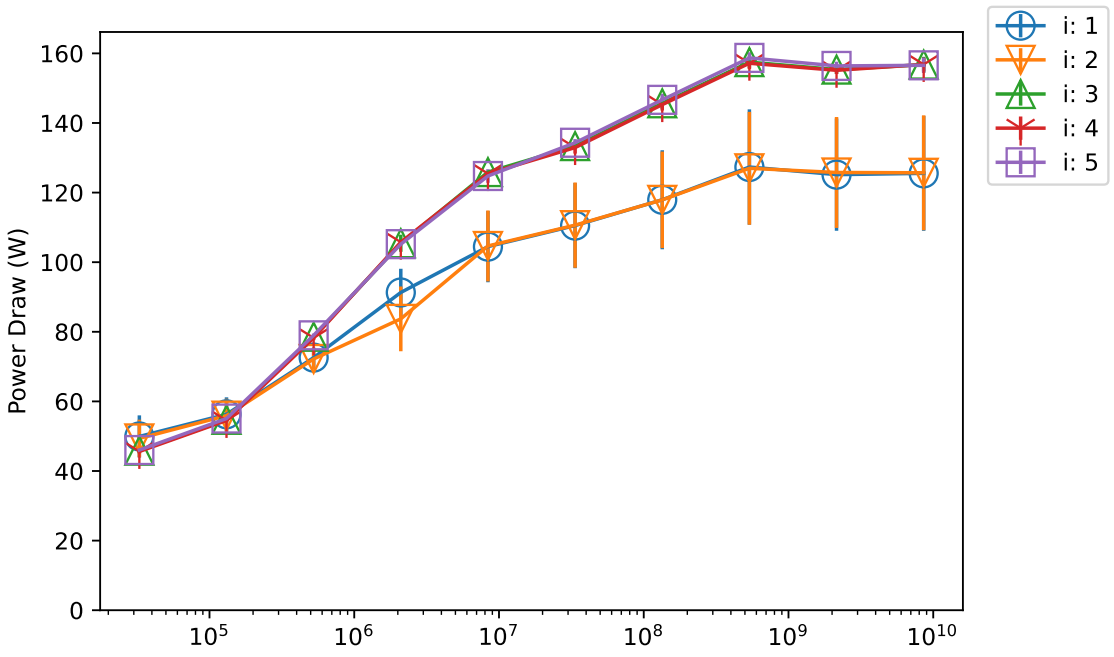

Figure 5.5: NVIDIA CUDA, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. Five individual runs plotting average power draw (W) over matrix size.

Figure 5.5 shows the execution of the Listing 4.2 kernel running for matrix sizes ranging from 64x64 to 32768x32768 in multiples of 2. The kernels are run sequentially after each other using Listing 2.1 with the biggest matrix being run 5 times whilst subsequent smaller matrices are run 4 times more often than their respective bigger size to ensure sufficient execution time. Each sequence of matrix multiplication kernel was run 5 times with each run being plotted as a separate line and marked in the legend with their index i (1-5).

Noticeably the five runs divide themselves into 2 different categories, the first having a high standard deviation within the run and a lower average power draw (Run 1&2). Here the maximum average power draw (127 W) is reached at a matrix size of 8192. With the two bigger matrix sizes reaching an average of only 126 W a difference that is negligible compared to the standard deviation within the runs. The standard deviation peaks at about 16 W for the higher matrix sizes. The difference between run 1 and 2 average and standard deviation are within a 1% range. The only exception to this seems to be at matrix size 512 were the average varies from 91 to 83 W.

On the other hand runs 3, 4 and 5 have a significantly higher power draw and lower standard deviation within them. Noticeably the highest average power draw is again observed for a matrix size of 8192 being 159 W. Closely followed by the bigger matrices having 157 W average power draw. The standard deviation reaches 2 W in as its highest values between all runs, thus being significantly lower compared to the other category of run.

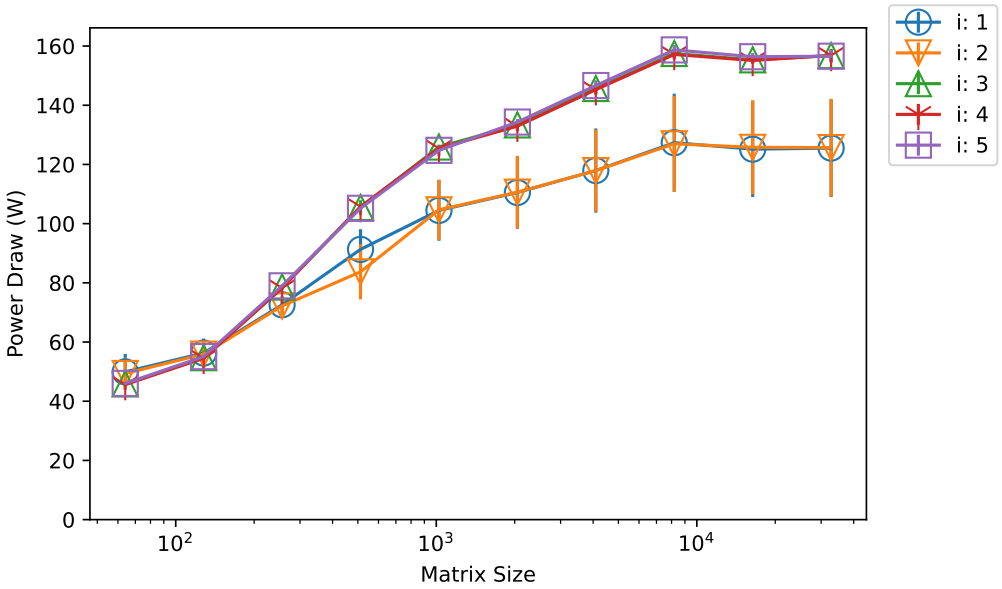

Figure 5.6: NVIDIA OpenCl, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. Five individual runs plotting average power draw (W) over matrix size.

This division seems to hold for the OpenCl (Figure 5.6) and HIP(not shown because identical with CUDA, except for run 2 having a higher power draw) of this run as well. The only exception seems to be the OpenCl run where a switch between the two power modes is happening. Keeping the order the data was gathered in mind; CUDA, OpenCl, HIP repeat, with each framework running from high matrix size to low matrix size within the run; this suggests that the first runs until OpenCl run

2 were performed in the “lower” power result and the subsequent runs in the “high” result. As this dual mode result doesn’t seem to repeat, in future runs it will be noted as an oddity for this paper. This supports the earlier notion of interleaving the runs for comparability as a separate collection of data for all frameworks might have resulted in having the external influence only on one framework skewing the discussion completely.

Looking at a different group of runs for NVIDIA has resulted in much cleaner graphs as shown in Figure 5.7.

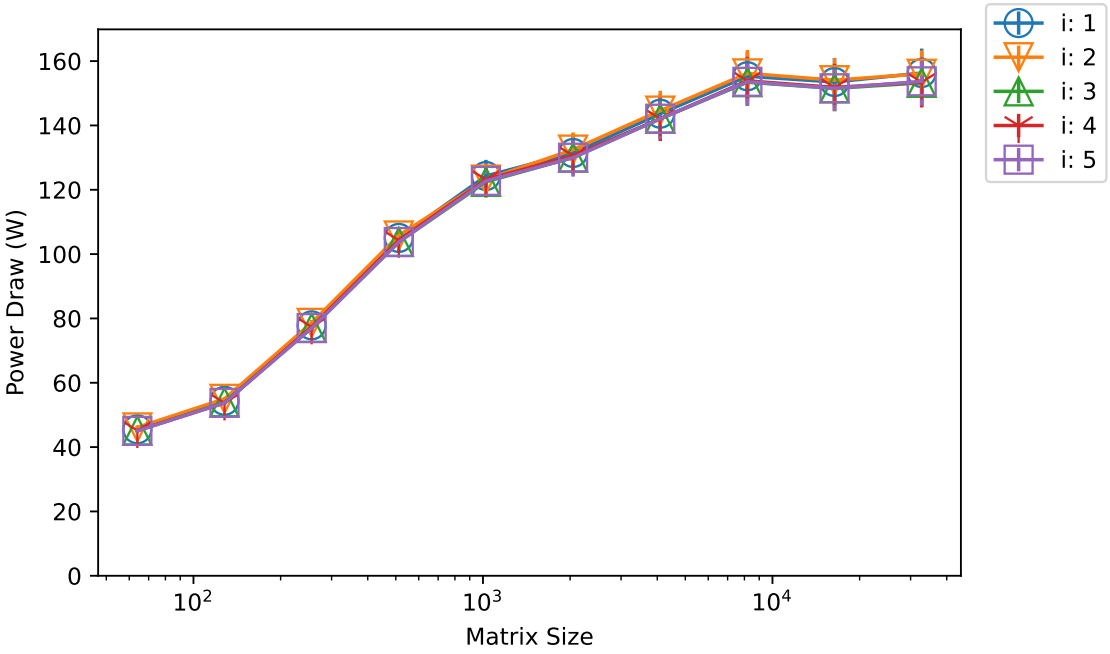

Figure 5.7: NVIDIA CUDA, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. Five individual runs plotting average power draw (W) over matrix size.

Figure 5.7 shows the same benchmark as the previous runs. It was run on gilgamesh using CUDA. The five individual runs have resulted in a similar average power draw over all matrix sizes. The most inter-run variation of the average power draw is 3 W at a matrix size of 32768 between the values 153 and 156 W. This 2 % variation seems reasonable compared to the intra run standard deviation having a maximum of 7.

Similar to the run shown in Figure 5.7 the corresponding OpenCl and HIP runs are fairly stable regarding there inter-run average power draw variation. This allows for looking at an accumulated average power draw comparing the different frameworks. Accumulated average power draw meaning a simple average calculation over all runs with the same matrix size.

As visualized in Figure 5.8 the average power draw ranges from 43 W for OpenCl 64 matrix size to 154 W at 32768 matrix size for CUDA. It is noticeable that CUDA and HIP seem to have a virtually identical power draw. Their highest absolute difference at 32768 matrix size is 1 W. Whilst this

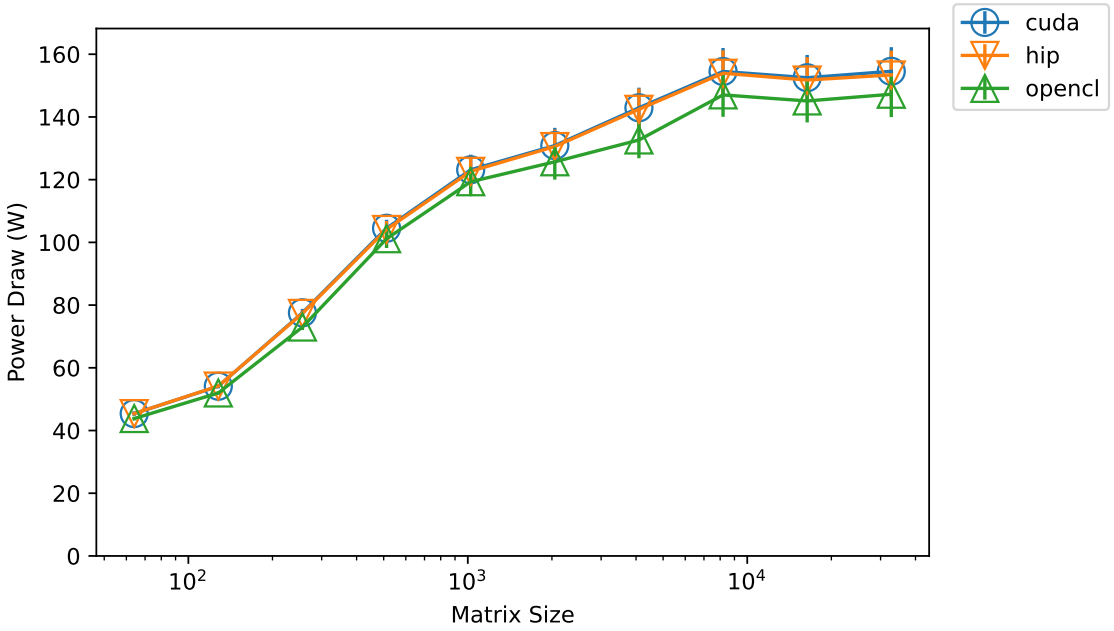

Figure 5.8: NVIDIA, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs per framework plotting average power draw for their respective frameworks used.

variation is large for the number of samples that go into it, resulting in a p-value way below 0.1%, it is still negligible compared to the differences of OpenCl to both frameworks. The difference of OpenCl varies from 2 W at 64 matrix size to 10 W at 4096 matrix size.

The next interesting metric for the matrix multiplication kernel is execution time per kernel. As only the execution time of a sequence of sequentially executed kernels was measured, this metric needs to be calculated by dividing the total time a run took by the total number of kernels run within it.

Figure 5.9 shows the execution time per kernel is increasing with the matrix size. Also, the execution time between frameworks seems to be virtually identical when viewed in this graph. Therefor Figure 5.10 shows the relative differences between the frameworks. However, before going the details of the differences the trajectory of the overall time taken is explored. Noticeably there is a switch at a matrix size 256 between 2 slopes of the curve. This switch might be related to the fact that the amount of warps scheduled reaches the limit of the GPU. This seems to hold as the maximum amount of threads per SM is 2048 [28] resulting in a total thread count of 262144 is equivalent to the amount of cells in the matrix and thus the amount of threads needed.

The variation between the average time of the 5 respective runs in-between frameworks reaches a maximum of 0.3% for OpenCl and CUDA. Whilst the difference between CUDA/HIP and OpenCl range from 0.4% to 10%, with the OpenCl implementation always taking longer than the other two implementations.

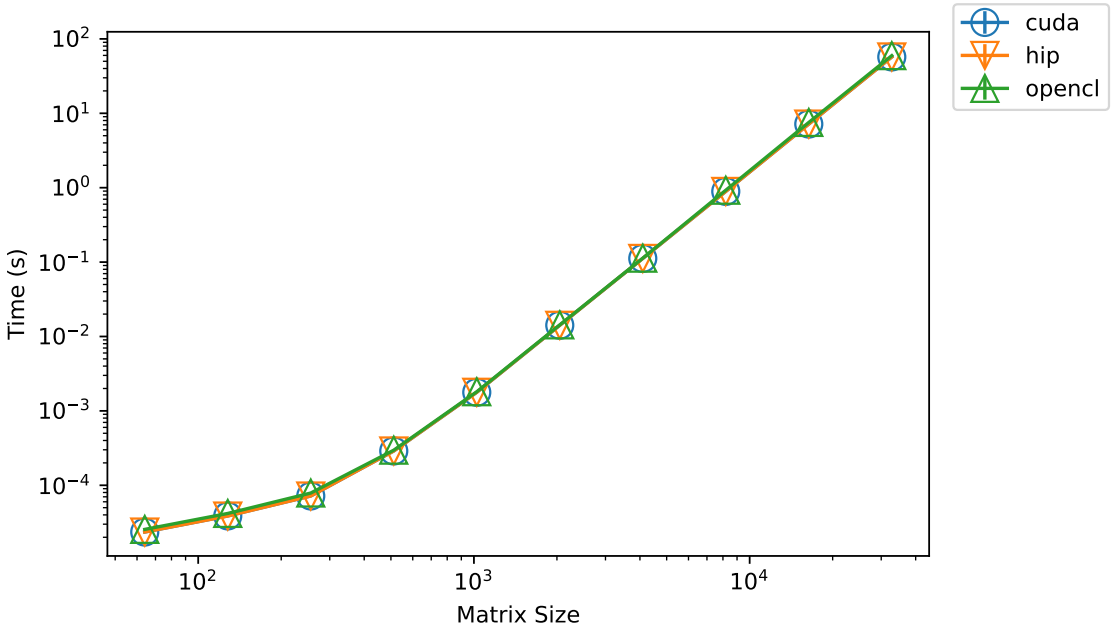

Figure 5.9: NVIDIA, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs per framework plotting $\frac{average\ time\ taken\ per\ run}{number\ of\ kernels\ within\ the\ run}$.

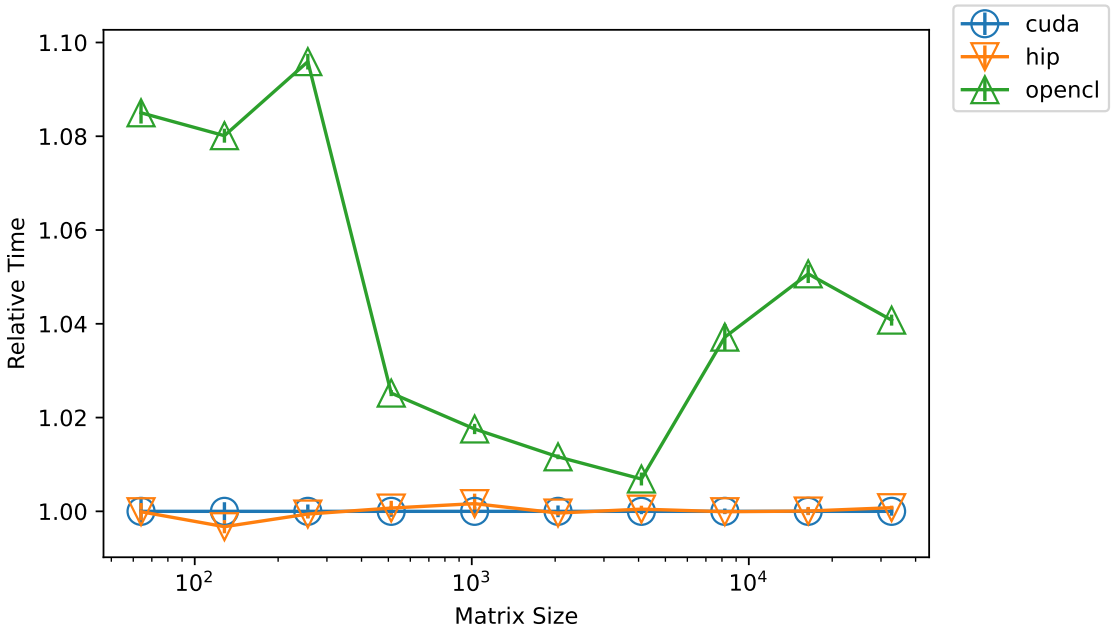

Figure 5.10: NVIDIA, gilgamesh: $5*(32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs per framework plotting $\frac{average\ time\ taken\ per\ run}{number\ of\ kernels\ within\ the\ run}$ normalized by the CUDA run.

However, the standard deviation of each time value over the five individual runs per framework only amounts to a maximum of 0.3 % of the respective time value (relative error), suggesting that the OpenCl timing is different to that of CUDA and HIP.

This is supported by a Welch t-test on the distributions which cannot find a difference between the CUDA and HIP execution time, except for a matrix size of 128 having a p-value of 2 %. However, as the comparison of OpenCl against the other two frameworks always reaches p values below 9×10^{-3}%, CUDA and HIP should likely be considered equal.

Putting the timing and power draw information together results in the total energy draw of the GPU for running the matrix multiplication kernel. As OpenCl has separated itself in average power and time taken from the other frameworks, it could be expected that it will also be separate in terms of energy consumption. However, as both separations lower power draw and longer time taken, have an opposing effect on the energy consumption it is possible that there is no separation in the final energy consumption.

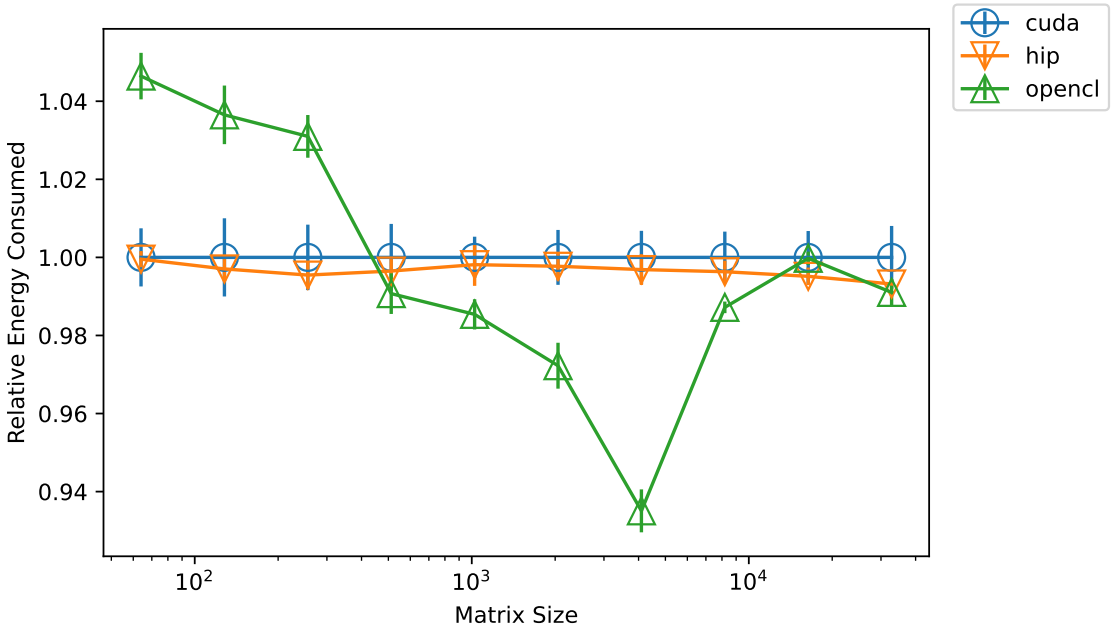

Figure 5.11: NVIDIA, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs per framework plotting $\frac{energy\ consumed\ per\ run}{number\ of\ kernels\ within\ the\ run}$ normalized to the CUDA run.

To calculate the total energy draw, the discrete integral over the power draw is used as described in Section 2.5.

The trajectory of the total energy consumption is quite similar to that of the execution time. Therefor only the relative changes are shown in Figure 5.11.

As before, the difference between the three frameworks seems rather small. The relative error of power consumption within each framework reaches a maximum of 1 %.

As CUDA and HIP were nearly identical in power draw and energy consumption, it stands to reason that the difference between these two is similar in the energy consumption as well. This seems to be the case as the highest relative offset between the two energy consumptions is 0.5 % and thus in the range of the standard deviation. However, it should be noted that the average energy consumption of CUDA is bigger in all matrix sizes then that of HIP. Looking at the p-value of the Welch's t-test between the two energy consumptions doesn't support that claim, as no p-value is below 10 %.

OpenCl paints a less clear picture. To start, the energy consumption is bigger than that of CUDA and HIP for all matrix sizes less than 512 and starts getting smaller than the other frameworks for matrix sizes starting at 512, with the exception of 16384 where it is in-between CUDA and HIP. With 2048, 4096 having 3 % and 6 % difference. However, the difference decreases to about 0.5 % for the higher matrix sizes. Looking at the p-values the energy consumption seem to be separated for all matrix sizes below 16384 with values below 2 %.

This means that for the described matrix multiplication, the choice of framework has most definitely an effect on the amount of energy used, with the highest differences reaching 6 %. However, the effect is non-trivial, as small matrix sizes seem to favor CUDA and HIP whilst big matrix sizes tend to favor OpenCl with huge matrix sizes performing almost equal.

5.2.2 AMD

After taking a look at the behaviour of one NVIDIA GPU. The next step is to take a look at the behaviour of AMD. For this the same metrics as NVIDIA will be taken, starting with looking at the average power draw compared between different runs, the variation between the two frameworks OpenCl and HIP, followed by the comparison of the time taken per kernel and finally the energy draw. As the instinct system has two GPUs all benchmarks were run on both GPUs in two separate threads without any synchronisation between the two. As the two GPUs performed the same, their values are both taken into account for this chapter, doubling the individual runs.

As Figure 5.12 shows there is no real differentiation between the power draw of each individual. However, it is noticeable that the qualitative behaviour of the power draw varies from that of NVIDIA in Figure 5.8. The power is starting significantly higher for the smallest matrix size at 105 W, which is about a third of the maximum power draw of the AMD MI100 GPU. This power draw quickly reaches a maximum power draw at a matrix size of 512 that is held for all higher matrix sizes. However, it does not reach the theoretical maximum of 300 W since it stagnates at 289 W. The inter-run variation is at most 0.5 W within each matrix size, whilst the standard deviation varies between 1 W and 25 W.

OpenCl has similar inter-run variance as HIP, thus the detailed analysis is skipped at this point and it is possible to move forward to the comparison of the two frameworks. As Figure 5.13 shows the qualitative behaviour is similar between the two frameworks with a high power draw for small matrix sizes that quickly saturates around a matrix size of 512. The saturation for OpenCl is also at 289 W, falling short of the theoretical maximum.

In the details however there are noticeable differences. The power draw of OpenCl does not fully saturate until a matrix size of 1024, with 512 having only a power draw of 280 W. This lower power draw is also observable for the smaller matrix sizes with 256 falling 12 W short of the 221 W power draw of HIP.

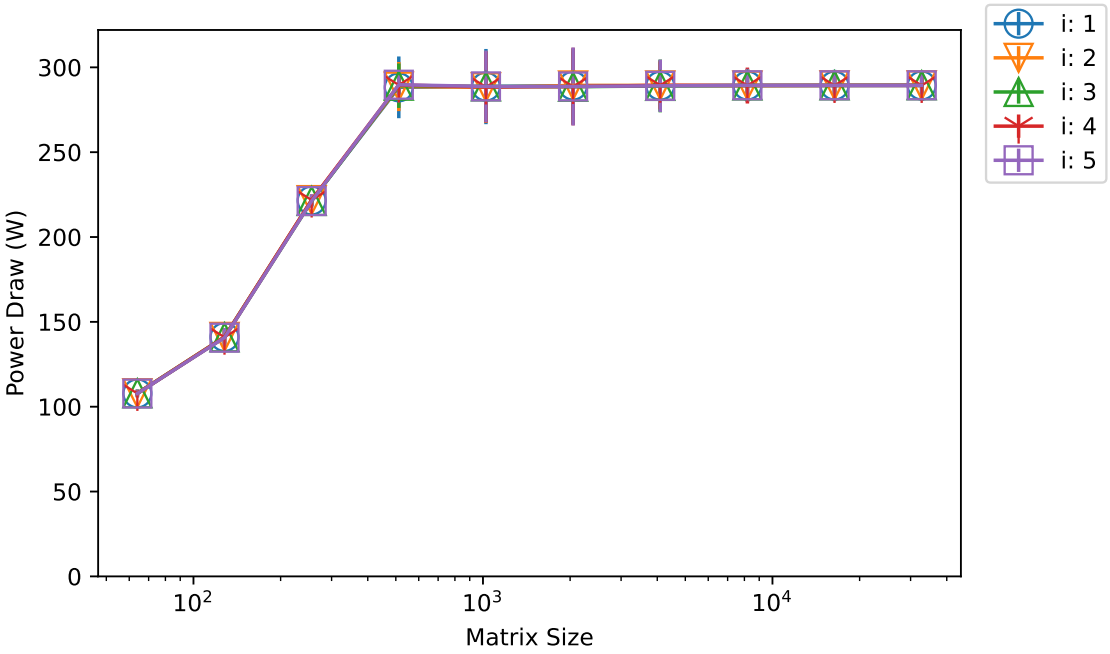

Figure 5.12: AMD, instinct: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels in five individual runs on 2 GPUs respectively performed in HIP showing the average power draw and standard deviation within each run.

Having analysed the average power draw Figure 5.14 shows the average time taken per the 5 times 2 (GPUs) runs. As the actual time taken per individual kernel looks similar to Figure 5.9 the time is additionally divided by the average time the respective HIP runs took to visualize the differences better. Here it can be seen that while the OpenCl framework takes longer to calculate the matrix multiplication for matrix sizes smaller than 2048, the circumstances change for bigger matrix sizes. Noticeable this switch does not seem to correlate with the maxing out of the power draw as this happens at a matrix size of 1024. Additionally, the separation between the two frameworks is much more pronounced for the smaller matrix sizes, varying between 7 % and 18 %, whilst the opposite case only varies between 1.4 % and 9 %.

Even though there is only a small separation between the average times for a matrix size of 8192, it could still be considered significant compared to the relative error of 0.8 %. For all other matrix sizes, the difference is most definitely statistically significant as the relative error only reaches a maximum of 1.4 %. The Welch's t-test supports that claim as no p-value exceeds 1.4×10^{-3}%.

As the power draw does not vary for a high matrix size between the two frameworks, it would be expected that the energy consumption follows the same behaviour. For the smaller matrix sizes the higher execution time and smaller power draw should shorten the gap between the 2 frameworks.

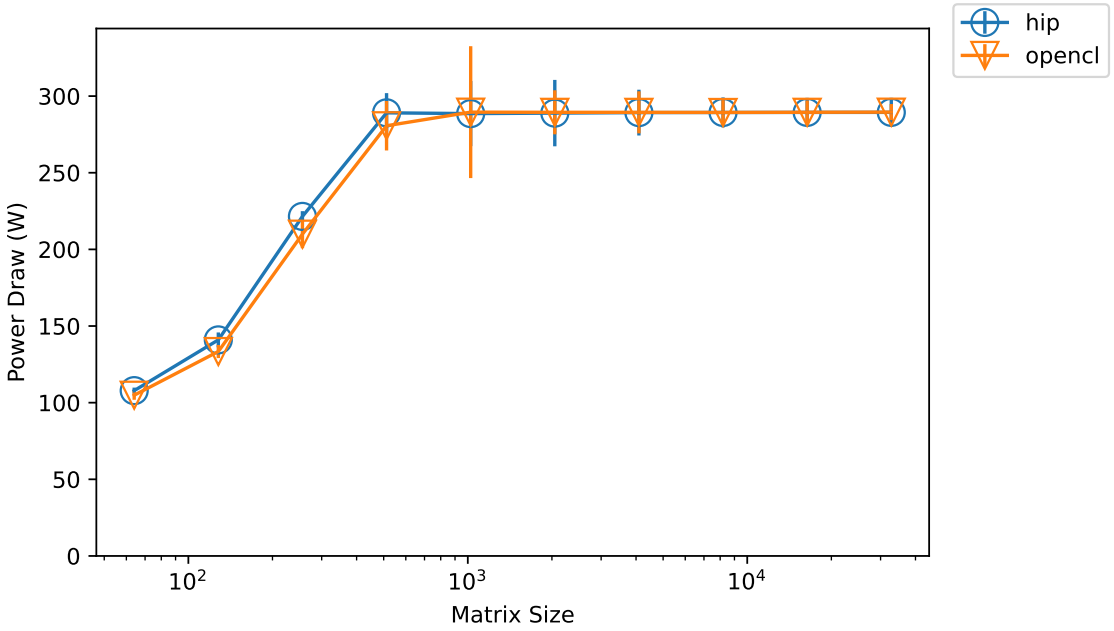

Figure 5.13: AMD, instinct: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels in five individual runs on 2 GPUs respectively performed in HIP and OpenCl showing the average power draw and standard deviation accumulated over all runs within the framework.

As the qualitative look of execution time per kernel plotted against matrix size is similar to that of NVIDIA as shown in Figure 5.9, it is more reasonable to directly look at the differences between the runs as shown in Figure 5.15. For the execution time the switch between the dominance of the 2 frameworks is happening between 2048 and 4096 with HIP having a smaller power draw for small matrix sizes and OpenCl being better for higher matrix sizes.

As expected, the separation between the 2 frameworks on the high matrix sizes is identical to that of the execution time. On the other hand, whilst still being clear, the separation for smaller matrix sizes is less extreme. The variation only varies between 2.2 % and 15 % sill holding significant compared to the maximum relative error of 1.6 %. Additionally, no p-value is exceeding 3.7×10^{-3}% meaning the differences are genuine.

This seems to suggest that for small matrix sizes less than or equal 2048 it is most definitely advantages to use HIP as it is faster and more energy efficient than OpenCl. For bigger matrix sizes OpenCl wins in both categories.

5.3 No explicit synchronisation between kernels

As discussed in Section 4.3.3 this section will show the differences in energy consumption when not using explicit synchronisation.

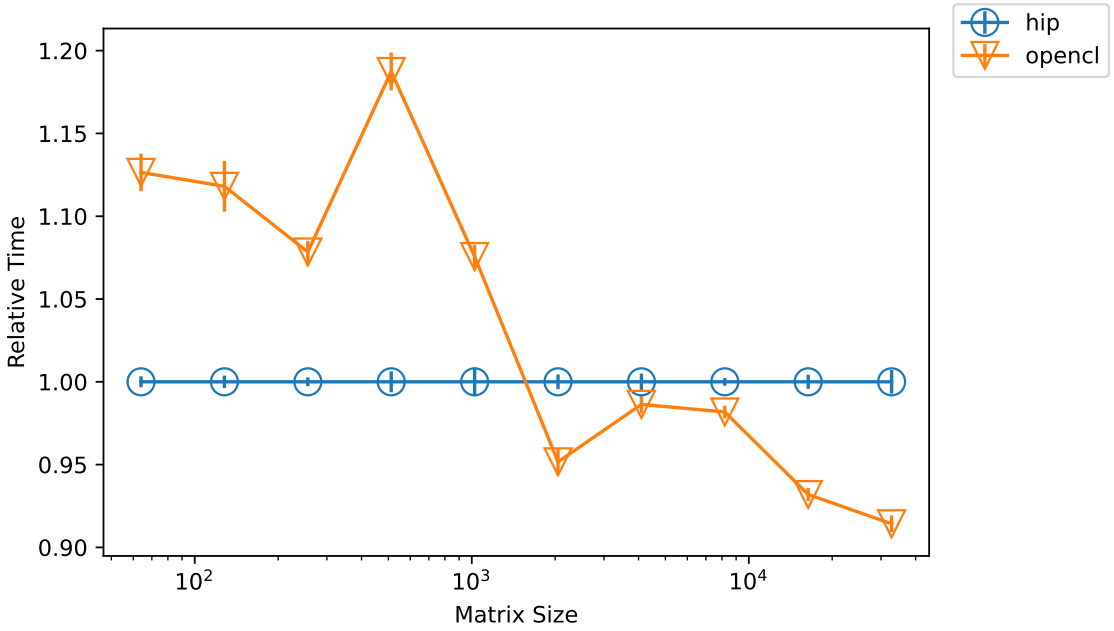

Figure 5.14: AMD, instinct: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels in five individual runs on 2 GPUs each performed in HIP and OpenCl showing the average time taken normalized on the HIP average time taken per runs.

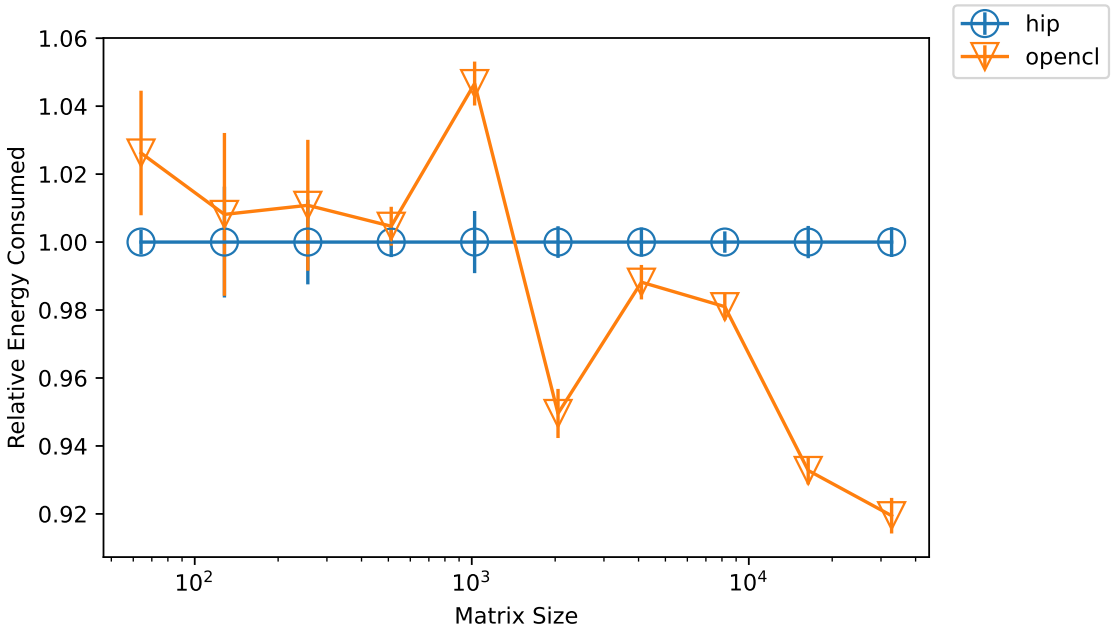

Figure 5.15: AMD, instinct: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels in five individual runs on 2 GPUs each performed in HIP and OpenCl showing the average time taken normalized on the HIP average time taken per runs.

5.3.1 NVIDIA

Looking at Figure 5.16 shows that energy consumption when using no event synchronisation is in general lower than that when using event synchronisation.

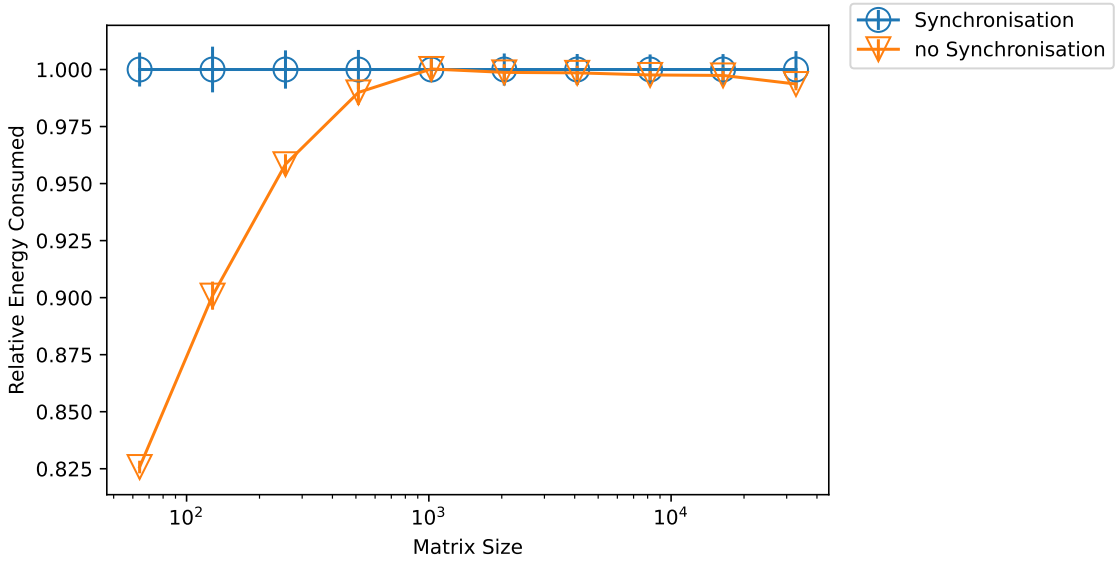

Figure 5.16: NVIDIA, gilgamesh, CUDA: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs in an explicit event synchronisation configuration and without events, plotting $\frac{energy\ consumed\ per\ run}{number\ of\ kernels\ within\ the\ run}$ normalized on the synchronisation run.

It is noticeable that the difference between the two implementations seems to diminish with increasing matrix sizes, vanishing between 512 and 1024 matrix sizes. The biggest difference accounts for an 18 % energy savings.

This could be caused by several behaviours. First the kernel time could be less than time that is required to synchronize with the last event and schedule the next, resulting in idle power draw adding to the total energy consumption. Second, the events could incur a constant overhead that vanishes compared to the huge load the big matrix sizes incur. Without further investigation this is hard to determine.

In the other frameworks the synchronisation change has a similar impact on relative change in energy consumption. This means that the relations between the frameworks in terms of energy use stay quite similar when not using events as shown in Figure 5.17. The only change seems to be that the relative differences between the frameworks are slightly bigger e.g. from 4 % to 6 % in case of the 64 matrix size.

5.3.2 AMD

After investigating the impact of no synchronisation on NVIDIA GPUs, this section discusses the same for AMD.

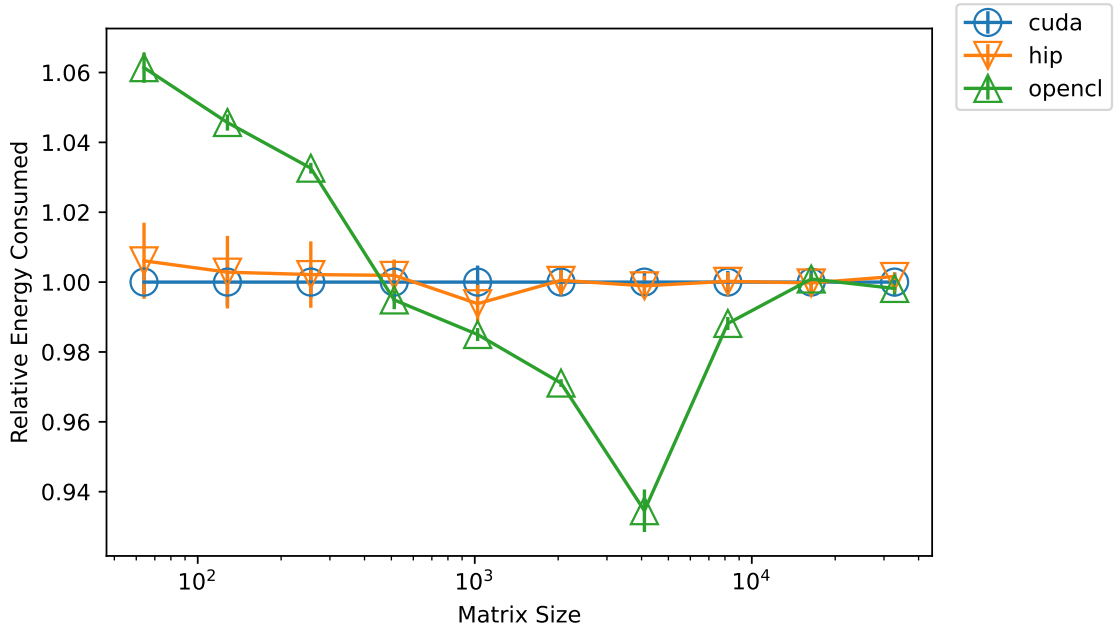

Figure 5.17: NVIDIA, gilgamesh, CUDA: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs in an explicit event synchronisation configuration and without events, plotting $\frac{energy\ consumed\ per\ run}{number\ of\ kernels\ within\ the\ run}$.

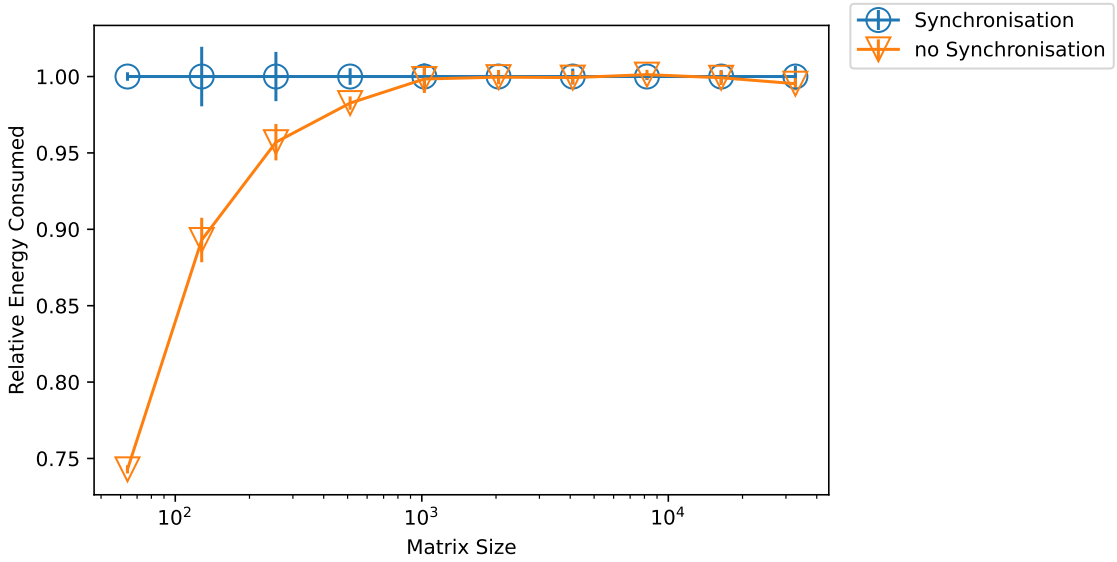

Figure 5.18: AMD, instinct, HIP: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs in an explicit event synchronisation configuration and without events, plotting $\frac{energy\ consumed\ per\ run}{number\ of\ kernels\ within\ the\ run}$ normalized on the run using synchronisation.

As Figure 5.18 shows the qualitative behaviour of the energy consumption as on NVIDIA GPUs. The only major difference is a 26 % improve in energy consumption for the smallest matrix size as opposed to the 20 % on NVIDIA and the subsequent bigger energy gains. Other than that there is not big difference as the difference seem to become negligible at a matrix size of 64 as well.

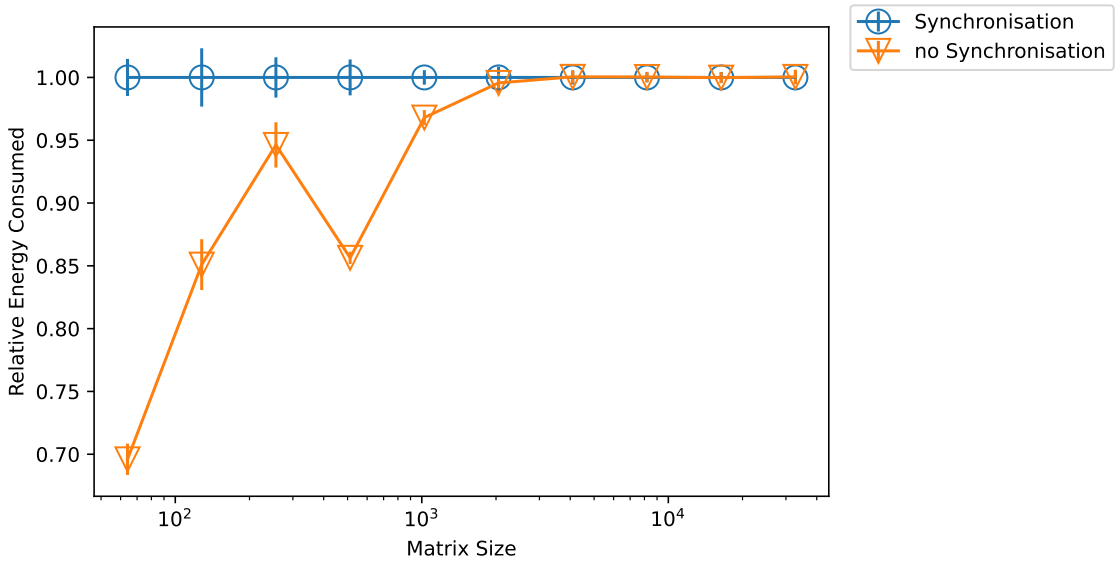

Figure 5.19: AMD, instinct, OpenCl: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs in an explicit event synchronisation configuration and without events, plotting $\frac{energy\ consumed\ per\ run}{number\ of\ kernels\ within\ the\ run}$ normalized on the run using synchronisation.

Instead of showing similar behaviour for OpenCl, Figure 5.19 shows significantly different behaviour for matrix sizes 512 and 1024 which break the trend of the surrounding values and having a significantly higher energy consumption gain. As the runs were taken in interleaved with the HIP runs and there is no high inter-run variation this seems to be caused by some internal behaviour of OpenCl not obvious. Other than that it can be noted that the energy saving further exceed them of the HIP implementation reaching 30 %.

These higher energy savings in OpenCl also remove the inter framework differences for small matrix sizes as shown in Figure 5.20. In numbers this means that matrix sizes 128-512 are only separated by about % whilst the relative error on the individual values reaches 2 %. This means that only matrix size that can confidently be assumed to benefit from HIP in terms of energy consumption is 1024. The bigger matrix sizes profit from OpenCl as already discussed in Section 5.2.2.

Due to the obvious beneficial effect on execution time and energy consumption, all subsequent benchmarks are run without using synchronisation.

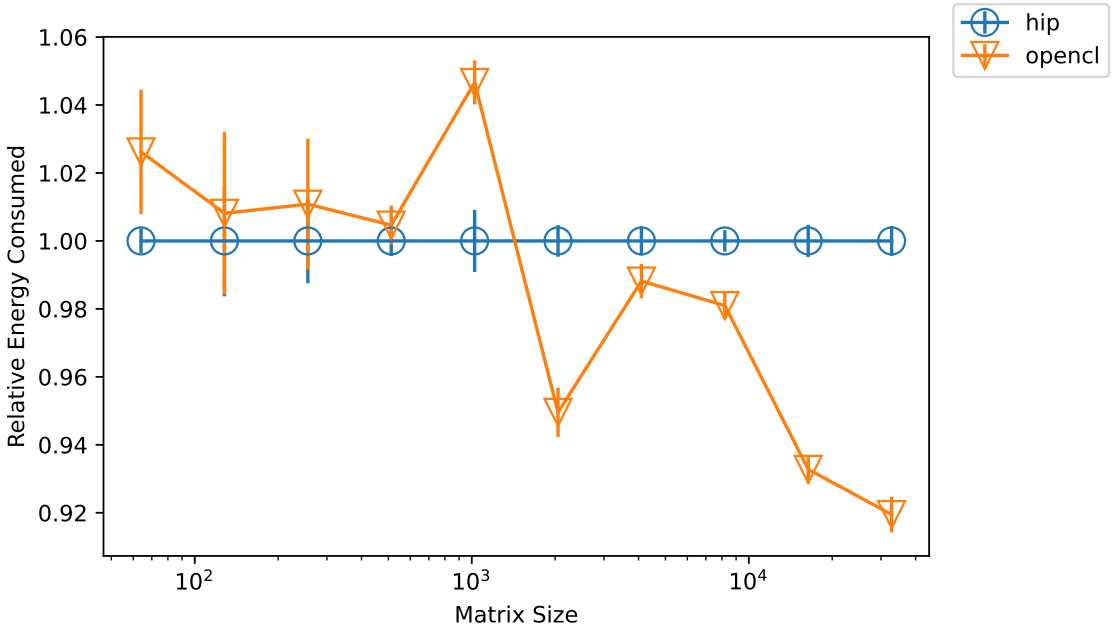

Figure 5.20: AMD, instinct: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs using no explicit event synchronisation with OpenCl and HIP, plotting $\frac{energy\ consumed\ per\ run}{number\ of\ kernels\ within\ the\ run}$ normalized on average energy of the HIP runs.

5.4 Comparing single and double precision

After the discussion of the impact of no synchronisation on energy savings in the different frameworks, this section will discuss the difference between single and double precision floating-point values on the power draw, execution time and energy consumption.

5.4.1 NVIDIA

To start the comparison, this part looks at the result for running single and double precision for NVIDIA on gilgamesh.

Comparing the single and double precision power draw as shown in Figure 5.21 no real predictable pattern seems to arise. The power draw is increasing constantly with matrix size used. However single and double precision switch in terms of the higher power draw. In contrast to the power draw of AMD the matrix size is not enough to max out the power limit of the GPU. Whilst the double precision power draw seems to plateau at a matrix size of 8192 the single precision implementation does not reach such a limit or at least it is not obvious.

Another interesting observation is that the inter framework variation has vanished for the single precision implementation suggesting that the actual instructions used on the GPU are similar for the smaller precision level.

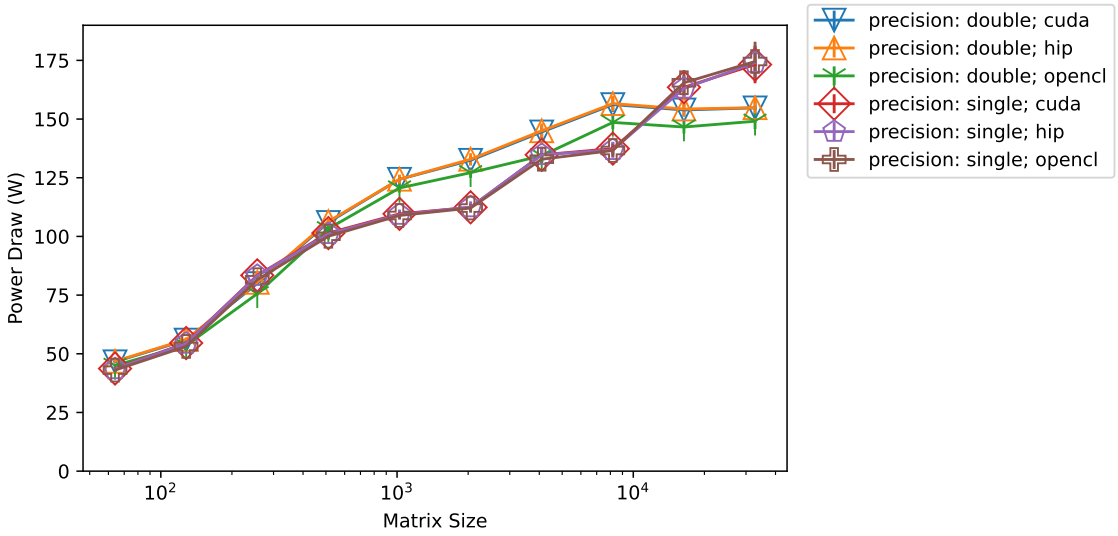

Figure 5.21: NVIDIA, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs, comparing single and double precision and framework, plotting average power draw over all runs.

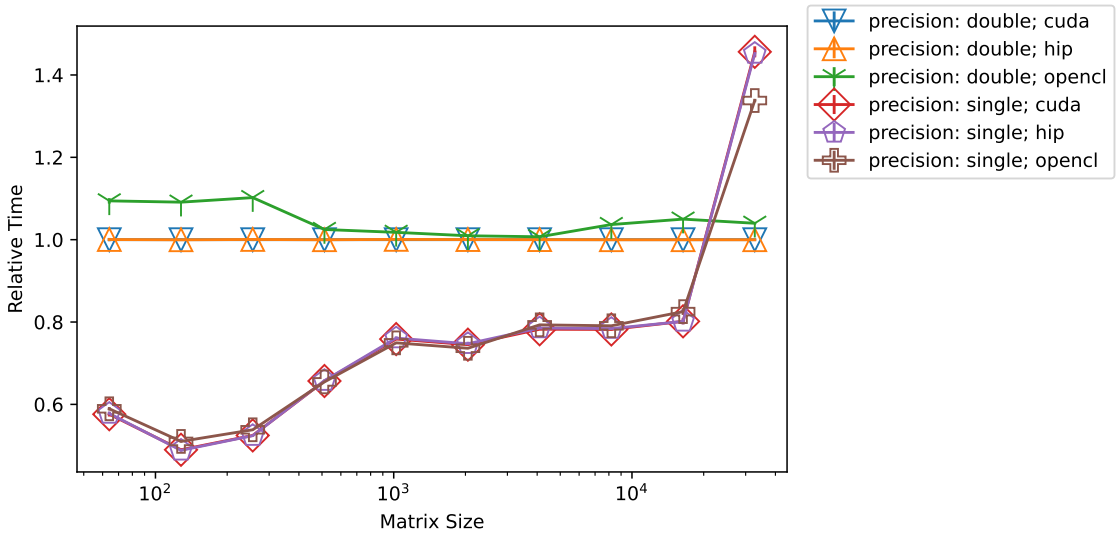

Figure 5.22: NVIDIA, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs, comparing single and double precision and framework, plotting average time taken per kernel normalized to the time of the cuda double precision values.

The absolute time values for the single precision multiplication pose a similar picture as the double precision values keeping the characteristic break as matrix size 256. Therefore, only the relative time taken per kernel is shown in Figure 5.22. Interestingly, the time taken per kernel starts off with about half of that of the double precision implementation. This difference diminishes as the matrix size increases, with the biggest matrix size completely breaking the picture and taking more than 30 % longer than the double precision implementation. This contradicts all logic as both the higher count of single precision Arithmetic Logic Units (ALUs) and the smaller memory footprint should result in a lower execution time, especially for large matrix sizes.

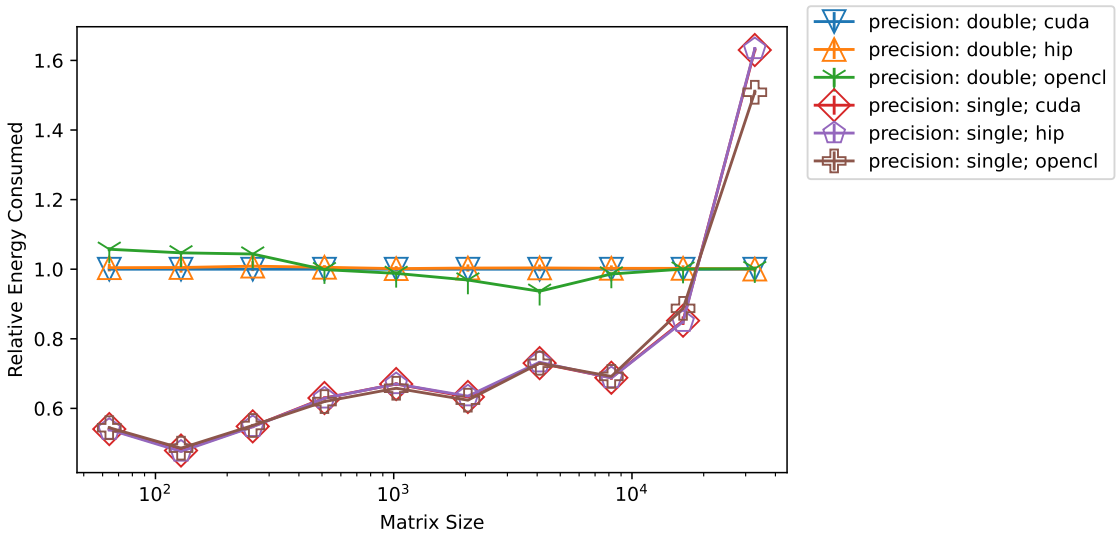

Figure 5.23: NVIDIA, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs, comparing single and double precision and framework, plotting average used per kernel normalized to the time of the cuda double precision values.

As the power draw is quite similar between precision and the time taken vastly differs, it is not surprising that the relative energy consumption looks quite similar to the relative time taken per kernel as shown in Figure 5.23. The biggest difference seems to be the peak of the single precision energy consumption exceeding the double precision implementation by over 60 %.

In conclusion, precision has a significantly higher impact on energy consumption than the choice of framework. However, even though not really visible in the graphics OpenCl still separates itself from the other two frameworks in an unpredictable pattern vastly varying over the matrix sizes but being mostly different with p-values below 5% compared to the other frameworks.

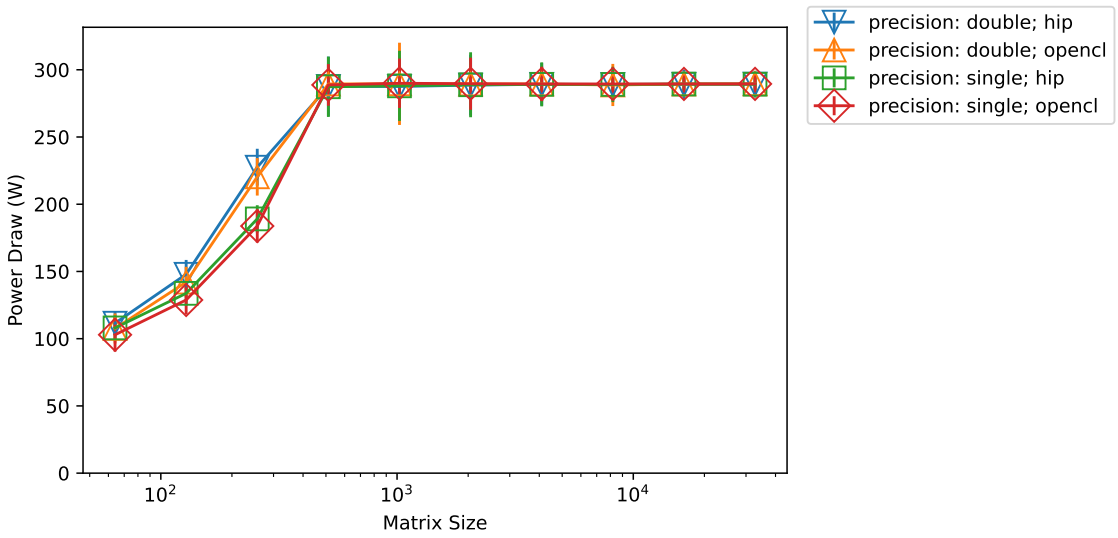

Figure 5.24: AMD, instinct: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs with OpenCl and HIP, comparing single and double precision, plotting average power draw over all runs.

5.4.2 AMD

As Figure 5.24 shows the change in precision results in a reduction of power draw for both frameworks that seems to exceed the inter framework differences for small matrix sizes (except 64). Single precision values have a lower power draw until reaching the maximum power draw of again 289 W. The biggest differences between the precision values being reached at a matrix size of 256, reaching about 40 W thus being significantly bigger than the framework differences of 5 W.

In Figure 5.25, it can be seen that the separation effect seen in the power draw for small matrix sizes is reversed in the time analysis meaning bigger matrix sizes have a clear separation whilst small matrix sizes do not have a big difference in execution time. On another note it also seems that the kink in the graph has moved from 1024 in single precision to 512 in double precision, with the smaller and larger values respectively forming a straight line. This could be explained by the better single precision peak performance of the GPU [2], causing it to be saturated a matrix size later than double precision.

As Figure 5.26 shows, that for large matrix sizes > 512, the time variation caused by change of precision far outweighs the time variation caused the choice of framework. The change of precision seems to cause an approximately 50 % faster execution. However, it is noticeable that for large matrix sizes and single precision the choice of framework is not impacting the execution time. Looking at the p-values non is smaller than 20 % making it impossible to consider the minimal differences between the frameworks more than statistical chance. This is surprising as the framework most definitely has an impact on the execution time when using double precision.

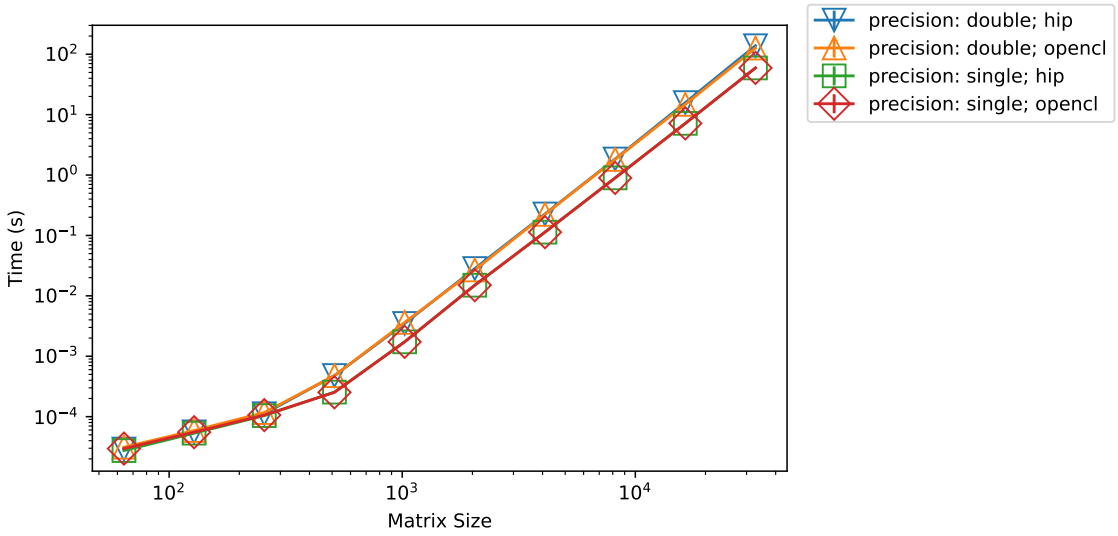

Figure 5.25: AMD, instinct: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs with OpenCl and HIP, comparing single and double precision, plotting average time taken per kernel against the matrix size used.

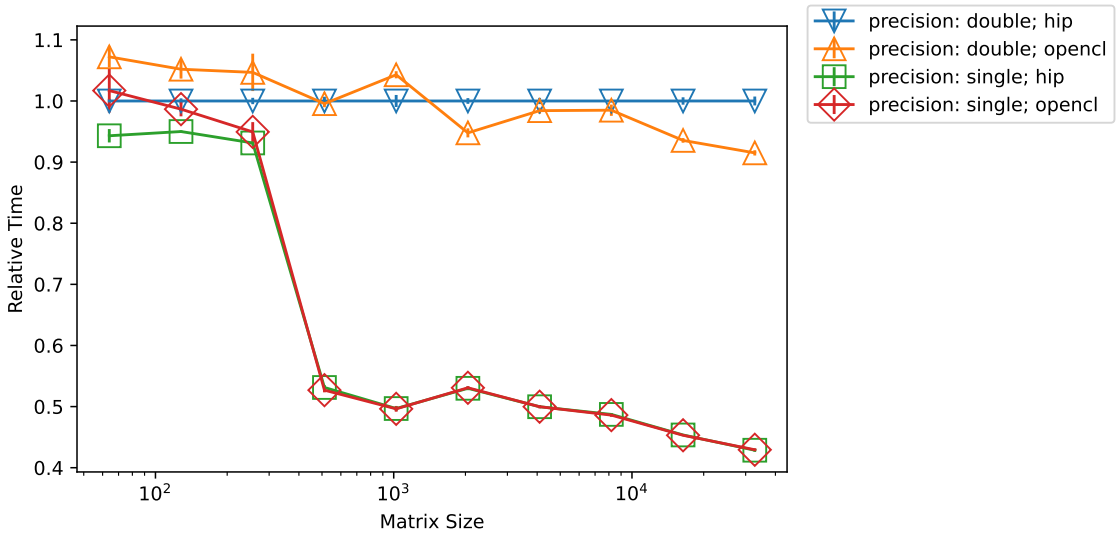

Figure 5.26: AMD, instinct: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs with OpenCl and HIP, comparing single and double precision, plotting average time taken per kernel against the matrix size used normalized by the time the double precision HIP implementation needed.

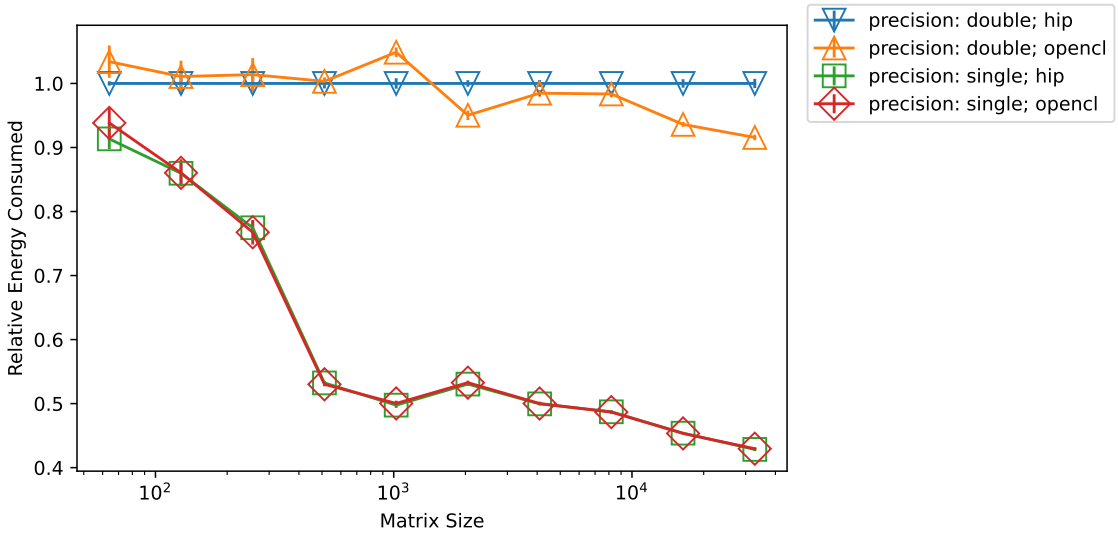

Figure 5.27: AMD, instinct: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs with OpenCl and HIP, comparing single and double precision, plotting average energy consumed per kernel against the matrix size used normalized by the time the double precision HIP implementation needed.

As both the execution time and power draw is reduced by use of single precision, it is not surprising that the effect persists; it is also visible in the energy consumption per kernel as shown in Figure 5.27. The only difference to the execution time is that the lesser power draw for small matrix sizes also reduces the energy consumption in this region moving the small matrix sizes clearer away from their double precision equivalent.

In conclusion reducing the precision reduces the energy consumption, execution time and power draw regardless of the framework used. Surprisingly, the inter-framework variation seems to vanish for large matrix sizes and single precision.

5.5 Shared memory matrix multiplication

This section will discuss the differences between the shared memory and naive matrix multiplication. The runs were performed without synchronisation and with double precision floating-point values using the kernel presented in Listing 4.4.

5.5.1 NVIDIA

Looking at the power draw (see Figure 5.28) between the naive and shared memory implementation shows that the power draw seems fairly similar between the 2 implementations and frameworks. However, it is still noticeable that generally the power draw is smaller in the shared memory implementation than the non-shared memory implementation. This difference varies from 1 % to 5 % for the smaller matrix sizes and reaching a maximum of 17 % for the 1024 matrix size, vanishing

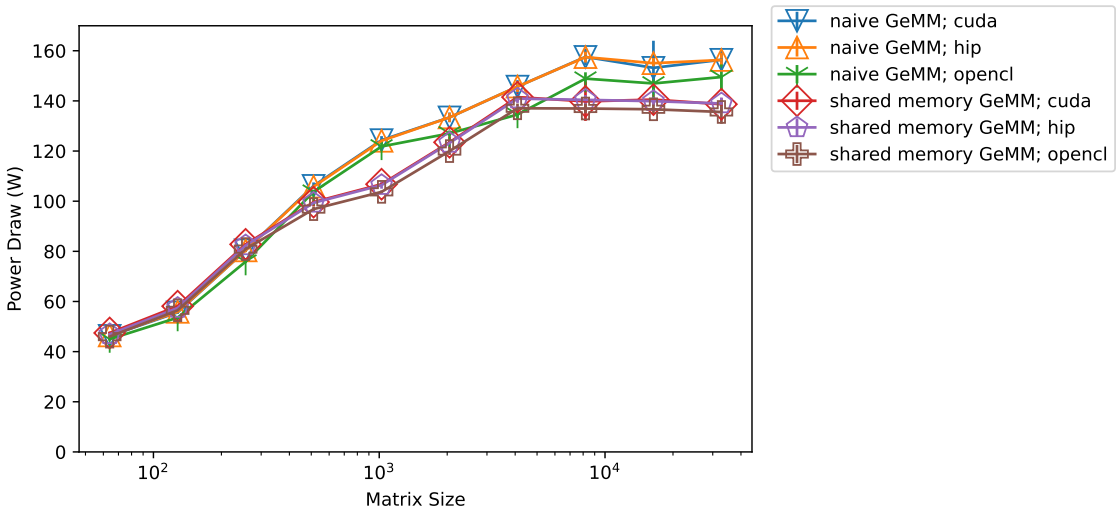

Figure 5.28: NVIDIA, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs, comparing naive and shared memory implementation and framework, plotting average power draw over matrix size.

at 4096 to below % and finally increasing again to 12 % for the highest matrix sizes. The latter seems to be due to the fact that the shared memory implementation has reached a plateau when using about 140 W for CUDA and HIP, and 136 W for OpenCl. The biggest relative error is 2 %.

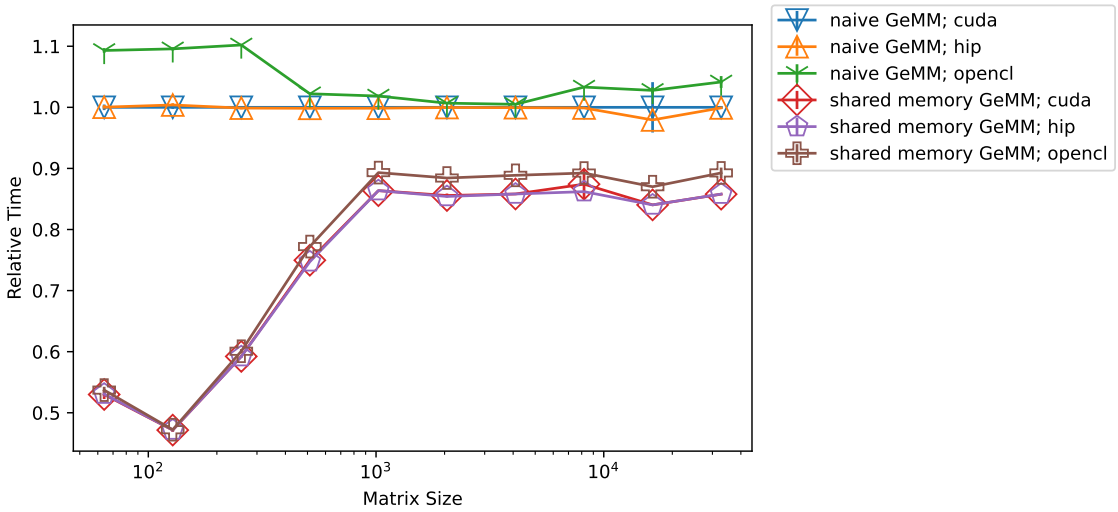

Figure 5.29: NVIDIA, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs, comparing naive and shared memory implementation and framework, plotting time taken per kernel over matrix size normalized to the naive CUDA implementation.

As before, the logarithmic time plot shows a characteristic change in slope at matrix size 512, allowing for a direct investigation of the relative execution time change as compared to the CUDA naive matrix implementation shown in Figure 5.29. These show a fairly constant difference for large matrix sizes starting with 2048 of about 14 %. However, the OpenCl implementation is still holding a roughly 3 % performance loss compared to the HIP and CUDA implementations with a p-value always below 0.2 %. This is in contrast to repeating inseparability of HIP and CUDA with a p-value above 12 %.

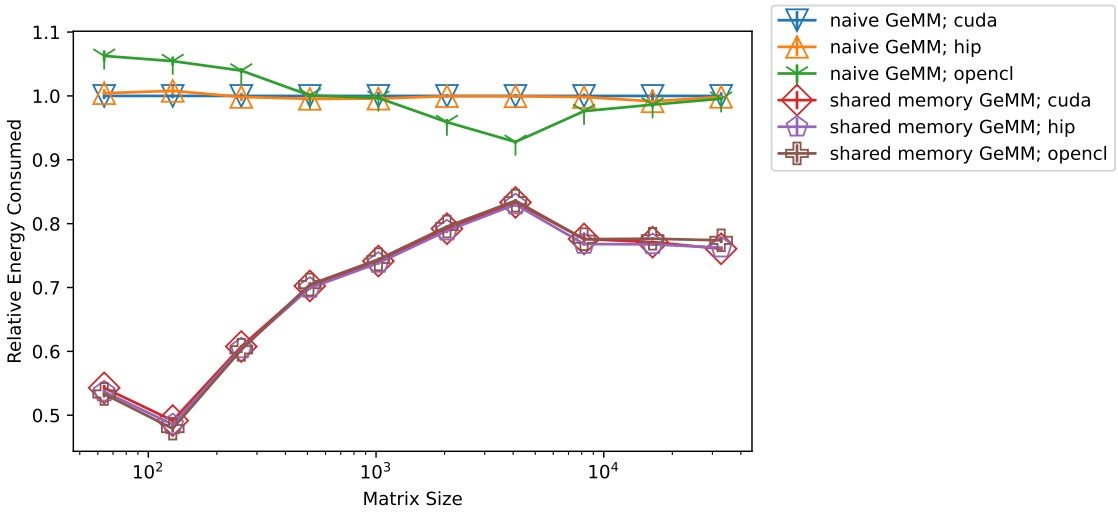

Figure 5.30: NVIDIA, gilgamesh: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs, comparing naive and shared memory implementation and framework, plotting average energy used per kernel over matrix size normalized to the naive CUDA implementation.

As both the power draw and execution time are reduced by the use of shared memory, it is not surprising that the overall energy consumption is also reduced as shown in Figure 5.30, with the highest energy savings of 50 % being reached at a matrix size of 128. Also, the energy savings seem to be inverted between the frameworks with OpenCl winning for small matrix sizes by 8 % and CUDA being beneficial for large matrix sizes by 2 %.

In conclusion, the savings gained by using a better algorithm far outweigh the inter framework differences. This suggests that an algorithm to be implemented should first be optimized before discussing the best framework to use for the algorithm.

5.5.2 AMD

After discussion of the behaviour of the NVIDIA shared memory run, this section discusses it for AMD.

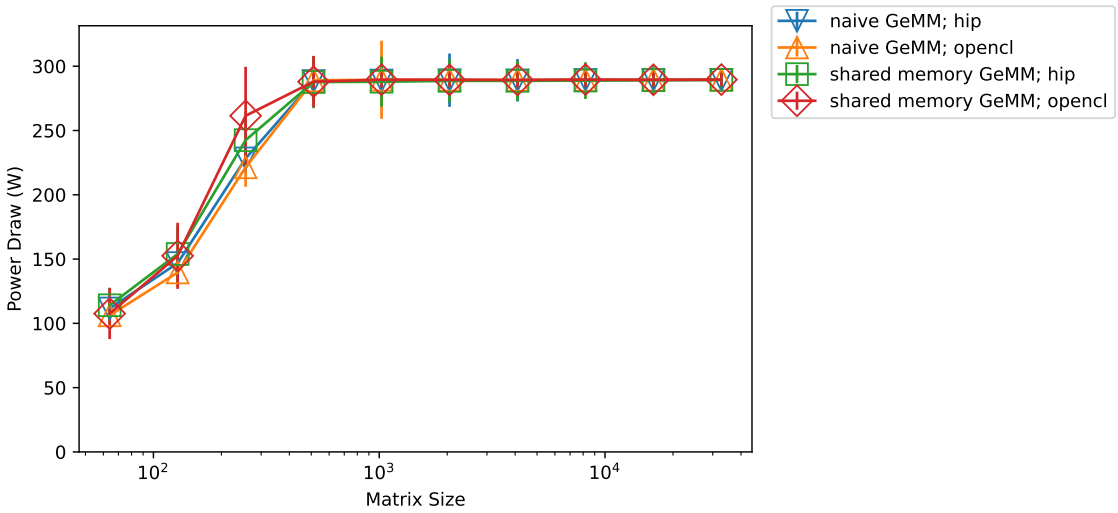

Figure 5.31: AMD, instinct: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs, comparing naive and shared memory implementation and framework, plotting average power draw over matrix size.

In contrast to the NVIDIA run, the implementation of shared memory seems to have increased the average power draw at least for small matrix sizes as shown in Figure 5.31. As before, the power draw reaches a maximum at matrix size 512. The maximum difference is reached again at a matrix size of 256 with approximately 15 %.

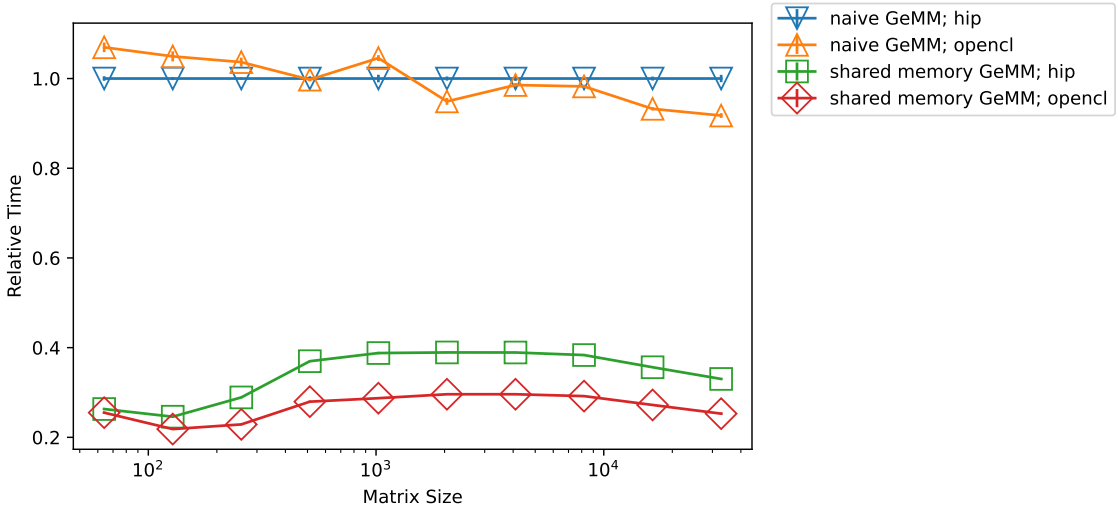

Figure 5.32: AMD, instinct: $5 * (32768/matrix\_size)^2$ consecutive matrix multiplication kernels. In five individual runs, comparing naive and shared memory implementation and framework, plotting time taken per kernel over matrix size normalized to the naive CUDA implementation.

Whilst power draw remains similar, as shown in Figure 5.32, execution time shows much more variation, with an at least 60 % improvement and even a 70 % improvement for small matrix sizes. Noticeably, HIP does not reach the same time improvements as OpenCl, being more than 30 % slower than OpenCl. This seems odd as HIP should be the native framework.

As power draw does not show a big variation compared to execution time, the energy consumption per kernel shows the same profile. Therefore, it can be concluded that for the best energy consumption the shared memory should most definitely be used. Additionally, the change also seems to have moved the beneficial framework to be most definitely OpenCl.

6 Conclusion and Outlook

6.1 Summary and Conclusion

This paper has covered the energy, power and time aspects of the implementation of a general matrix multiplication in a variety of frameworks, with regard to some parameters and minor optimisations used around this algorithm. In particular, variations examined include the difference between single and double precision values, the usage of events for explicit synchronisation to the CPU, no synchronisation and finally the usage of shared memory.

It was discovered that the major driver for energy savings seems to be the execution time of the kernel. This is rooted in the observation that the power draw of similar kernels does not vary as much as their execution time.

As for the single effects, it is probably best to use a shared memory single precision matrix multiplication without synchronisation before choosing a framework. These optimisations are already known to reduce execution time and are also the main driver for energy savings up to 60 %.

The choice of framework has only a minor role, of up to 20 %, in impacting the energy consumption. Whilst often only reaching 5-10 % of energy savings the specific framework to achieve this is often unpredictable. For example, CUDA is the better framework for small matrix sizes in the naive matrix multiplication implementation, whilst losing to OpenCl for bigger matrix sizes on both architectures. However, this relationship is completely reverted for the shared memory implementation, leaving OpenCl as the better choice for small matrix sizes on NVIDIA and the overall better choice on AMD by a clear margin.

Additionally, the impact of changing the frequency of an NVIDIA GPU on the power draw and energy consumption was discussed. This has shown that the presumable linear relationship between power draw and core frequency is not easily reproducible. This seems to be rooted in the power draw causing heat dissipation that changes the core temperature, which intern has an effect on the energy consumption. It was also confirmed that DVFS is a good measure to safe energy. In a single instance this resulted in a 20 % energy saving.

6.2 Future Research

It is left unclear why the differences between the frameworks arise and why they seem to be so unpredictable with regard to their energy-beneficial behaviour. Another future investigation could examine the specific byte codes that are generated, and how they are scheduled in the different frameworks.

As the discussion of high level code for energy consumption has yielded nearly unpredictable outcomes, it would probably be best to perform low level research of single instructions and their specific impact on the energy consumption on a GPU. This would be quite similar to the research utilizing the performance counter in the GPU to predict the energy consumption of a kernel.

This paper only considers a limited variety of parameters and comparisons of these parameters. There was not enough data gathered to give a comprehensive overview of the impact of the parameters on the energy consumption. Therefore, it would be a reasonable next step to gather this data and analyse whether there are general relationships deducible from it.

Bibliography

[1] *A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels | ACM Transactions on Architecture and Code Optimization*. URL: https://dl.acm.org/doi/abs/10.1145/3431731 (visited on 10/08/2021) (cit. on p. 25).

[2] Advanced Micro Devices. *AMD CDNA2 Architectureamd-cdna2-white-paper.pdf*. 2021. URL: https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf (visited on 03/08/2022) (cit. on pp. 18, 52).

[3] AMD. *AMD INSTINCT™ MI100 ACCELERATOR*. 2020. URL: https://www.amd.com/system/files/documents/instinct-mi100-brochure.pdf (visited on 03/26/2022) (cit. on p. 27).

[4] AMD. *HIP Programming Guide v4.2*. 2021 (cit. on pp. 19, 20).

[5] AMD. *ROCm SMI Manual 4.2*. May 11, 2021. URL: https://github.com/RadeonOpenCompute/ROCm/blob/master/ROCm_SMI_Manual_4.2.pdf (visited on 06/22/2021) (cit. on p. 28).

[6] *BLAS (Basic Linear Algebra Subprograms)*. June 29, 2021. URL: http://www.netlib.org/blas/ (visited on 02/15/2022) (cit. on p. 20).

[7] Bob Crovella. *07_Concurrency.pdf*. CUDA CONCURRENCY. July 21, 2020. URL: https://www.olcf.ornl.gov/wp-content/uploads/2020/07/07_Concurrency.pdf (visited on 03/16/2022) (cit. on p. 18).

[8] R. A. Bridges, N. Imam, T. M. Mintz. "Understanding GPU Power: A Survey of Profiling, Modeling, and Simulation Methods". In: *ACM Computing Surveys* 49.3 (Sept. 16, 2016), 41:1–41:27. ISSN: 0360-0300. DOI: 10.1145/2962131. URL: https://doi.org/10.1145/2962131 (visited on 10/05/2021) (cit. on p. 25).

[9] K. Cameron, R. Ge, X. Feng. "High-performance, power-aware distributed computing for scientific applications". In: *Computer* 38.11 (Nov. 2005). Conference Name: Computer, pp. 40–47. ISSN: 1558-0814. DOI: 10.1109/MC.2005.380 (cit. on p. 25).

[10] J. Coplin, M. Burtscher. "Energy, Power, and Performance Characterization of GPGPU Benchmark Programs". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). May 2016, pp. 1190–1199. DOI: 10.1109/IPDPSW.2016.164 (cit. on p. 25).

[11] *cuBLAS*. Archive Location: CUDA API References. URL: https://docs.nvidia.com/cuda/cublas/index.html (visited on 02/16/2022) (cit. on p. 20).

[12] *CUDA C++ Programming Guide*. Archive Location: Programming Guides. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 02/22/2022) (cit. on pp. 18–20).

[13] J. Fang, A. L. Varbanescu, H. Sips. "A Comprehensive Performance Comparison of CUDA and OpenCL". In: *2011 International Conference on Parallel Processing*. 2011 International Conference on Parallel Processing. ISSN: 2332-5690. Sept. 2011, pp. 216–225. DOI: 10.1109/ICPP.2011.45 (cit. on p. 25).

[14] W.-c. Feng, X. Feng, R. Ge. "Green Supercomputing Comes of Age". In: *IT Professional* 10.1 (Jan. 2008). Conference Name: IT Professional, pp. 17–23. ISSN: 1941-045X. DOI: 10.1109/MITP.2008.8 (cit. on p. 25).

[15] *Guidelines — hipBLAS documentation*. URL: https://hipblas.readthedocs.io/en/latest/functions.html (visited on 02/16/2022) (cit. on p. 20).

[16] S. Hemmert. "Green HPC: From Nice to Necessity". In: *Computing in Science Engineering* 12.6 (Nov. 2010). Conference Name: Computing in Science Engineering, pp. 8–10. ISSN: 1558-366X. DOI: 10.1109/MCSE.2010.134 (cit. on p. 15).

[17] *HIP-FAQ — ROCm Documentation 1.0.0 documentation*. URL: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-FAQ.html#can-a-hip-binary-run-on-both-amd-and-nvidia-platforms (visited on 08/31/2021) (cit. on p. 15).

[18] S. Hong, H. Kim. "An integrated GPU power and performance model". In: *Proceedings of the 37th annual international symposium on Computer architecture*. ISCA '10. New York, NY, USA: Association for Computing Machinery, June 19, 2010, pp. 280–289. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1815998. URL: https://doi.org/10.1145/1815961.1815998 (visited on 01/26/2022) (cit. on pp. 21, 25, 34).

[19] M. J. Ikram, O. A. Abulnaja, M. E. Saleh, M. A. Al-Hashimi. "Measuring power and energy consumption of programs running on kepler GPUs". In: *2017 Intl Conf on Advanced Control Circuits Systems (ACCS) Systems 2017 Intl Conf on New Paradigms in Electronics Information Technology (PEIT)*. 2017 Intl Conf on Advanced Control Circuits Systems (ACCS) Systems 2017 Intl Conf on New Paradigms in Electronics Information Technology (PEIT). Nov. 2017, pp. 18–25. DOI: 10.1109/ACCS-PEIT.2017.8302995 (cit. on p. 25).

[20] S. Irani, S. Shukla, R. Gupta. "Algorithms for power savings". In: *ACM Transactions on Algorithms* 3.4 (Nov. 1, 2007), 41–es. ISSN: 1549-6325. DOI: 10.1145/1290672.1290678. URL: https://doi.org/10.1145/1290672.1290678 (visited on 01/24/2022) (cit. on pp. 21, 25, 34).

[21] Y. Jiao, H. Lin, P. Balaji, W. Feng. "Power and Performance Characterization of Computational Kernels on the GPU". In: *2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on Cyber, Physical and Social Computing*. 2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on Cyber, Physical and Social Computing. Dec. 2010, pp. 221–228. DOI: 10.1109/GreenCom-CPSCom.2010.143 (cit. on pp. 25, 34).

[22] J. Li, B. Guo, Y. Shen, D. Li, Y. Huang. "Low-Energy Kernel Scheduling Approach for Energy Saving". In: *2016 13th International Conference on Embedded Software and Systems (ICESS)*. 2016 13th International Conference on Embedded Software and Systems (ICESS). Aug. 2016, pp. 30–35. DOI: 10.1109/ICESS.2016.29 (cit. on p. 25).

[23] X. Mei, Q. Wang, X. Chu. "A survey and measurement study of GPU DVFS on energy conservation". In: *Digital Communications and Networks* 3.2 (May 1, 2017), pp. 89–100. ISSN: 2352-8648. DOI: 10.1016/j.dcan.2016.10.001. URL: https://www.sciencedirect.com/science/article/pii/S2352864816300736 (visited on 10/07/2021) (cit. on pp. 21, 25).

[24] A. Munshi. “The OpenCL specification”. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. 2009 IEEE Hot Chips 21 Symposium (HCS). Aug. 2009, pp. 1–314. DOI: 10.1109/HOTCHIPS.2009.7478342 (cit. on pp. 15, 19, 20).

[25] NVIDIA. *Data Sheet: Tesla P100*. 2016. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf (visited on 01/10/2022) (cit. on p. 27).

[26] NVIDIA. *nvidia-smi documentation*. July 26, 2016. URL: https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf (visited on 03/26/2022) (cit. on p. 28).

[27] *NVIDIA A100 Tensor Core GPU*. manual. tex.organization: NVIDIA Corporation. 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf (visited on 08/30/2021) (cit. on p. 27).

[28] *NVIDIA Ampere Architecture*. NVIDIA. URL: https://resources.nvidia.com/c/ampere-architecture-white-paper?x=sFVHf4&lx=OhKlSJ&topic=Solution+Brief&xs=169656 (visited on 10/24/2021) (cit. on pp. 17, 39).

[29] S. Payvar, M. Pelcat, T. D. Hämäläinen. “A model of architecture for estimating GPU processing performance and power”. In: *Design Automation for Embedded Systems* 25.1 (Mar. 1, 2021), pp. 43–63. ISSN: 1572-8080. DOI: 10.1007/s10617-020-09244-4. URL: https://doi.org/10.1007/s10617-020-09244-4 (visited on 10/08/2021) (cit. on p. 25).

[30] M. Qasaimeh, J. Zambreno, P. H. Jones, K. Denolf, J. Lo, K. Vissers. “Analyzing the Energy-Efficiency of Vision Kernels on Embedded CPU, GPU and FPGA Platforms”. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). ISSN: 2576-2621. Apr. 2019, pp. 336–336. DOI: 10.1109/FCCM.2019.00077 (cit. on p. 25).

[31] Steve Rennich. *StreamsAndConcurrencyWebinar.pdf*. CUDA C/C++ Streams and Concurrency. URL: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf (visited on 03/16/2022) (cit. on p. 18).

[32] C.-L. Su, P.-Y. Chen, C.-C. Lan, L.-S. Huang, K.-H. Wu. “Overview and comparison of OpenCL and CUDA technology for GPGPU”. In: *2012 IEEE Asia Pacific Conference on Circuits and Systems*. 2012 IEEE Asia Pacific Conference on Circuits and Systems. Dec. 2012, pp. 448–451. DOI: 10.1109/APCCAS.2012.6419068 (cit. on p. 25).

[33] L. Wang, G. von Laszewski, J. Dayal, F. Wang. “Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS”. In: *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing. May 2010, pp. 368–377. DOI: 10.1109/CCGRID.2010.19 (cit. on p. 25).

[34] Q. Wang, N. Li, L. Shen, Z. Wang. “A statistic approach for power analysis of integrated GPU”. In: *Soft Computing* 23.3 (Feb. 1, 2019), pp. 827–836. ISSN: 1433-7479. DOI: 10.1007/s00500-017-2786-1. URL: https://doi.org/10.1007/s00500-017-2786-1 (visited on 10/07/2021) (cit. on p. 25).

[35] B. L. Welch. “The Generalization of ‘Student’s’ Problem when Several Different Population Variances are Involved”. In: *Biometrika* 34.1 (1947). Publisher: [Oxford University Press, Biometrika Trust], pp. 28–35. ISSN: 0006-3444. DOI: 10.2307/2332510. URL: https://www.jstor.org/stable/2332510 (visited on 03/17/2022) (cit. on p. 22).

[36] C. Xiong, N. Xu. “Performance Comparison of BLAS on CPU, GPU and FPGA”. In: *2020 IEEE 9th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*. 2020 IEEE 9th Joint International Information Technology and Artificial Intelligence Conference (ITAIC). Vol. 9. ISSN: 2693-2865. Dec. 2020, pp. 193–197. DOI: 10.1109/ITAIC49862.2020.9338793 (cit. on p. 25).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature