CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

# Evaluation of Rust for GPGPU high-performance computing

## A performance comparison between native C++-CUDA kernels and Rust-CUDA kernels on NVIDIA hardware

Master's thesis in Computer science and engineering

Viktor Franzén, Carl Östling

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

# Evaluation of Rust for GPGPU high-performance computing

A performance comparison between native C++-CUDA kernels and Rust-CUDA kernels on NVIDIA hardware

Viktor Franzén, Carl Östling

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Evaluation of Rust for GPGPU high-performance computing
A performance comparison between native C++-CUDA kernels and Rust-CUDA kernels on NVIDIA hardware



Supervisor: Miquel Pericas, Department of Computer Science and Engineering
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: C++ running away from a hungry Rust.

Typeset in LaTeX
Gothenburg, Sweden 2022

Evaluation of Rust for GPGPU high-performance computing
A performance comparison between native C++-CUDA kernels and Rust-CUDA kernels on NVIDIA hardware
Viktor Franzén, Carl Östling
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

Research within computer science constantly aims to find ways to improve computing performance in various ways. With the apparent death of Moore's Law, researchers are focused on exploiting other ways of improving performance, for example, programming language optimizations. Fields such as web development, databases and, machine learning are adopting new programming languages but GPU programming is not seeing this adoption. This thesis aims to evaluate Rust as a competitor to the status quo of programming languages for GPU programming, namely C++. In order to achieve this, comparable CUDA kernels are built in both C++ and Rust which are then profiled and analyzed for insights that can indicate reasons why one might outperform the other. The results show that *execution time* and *energy consumption* of C++ kernels are lower in nearly every experiment when having line-by-line comparable code. However, slight adjustments to the implementation of the Rust kernels close this gap to the point that they are performing at the same level. This indicates that it is possible to implement high-performance GPGPU programs in Rust, while still bringing some of its features such as memory safety and modern programming utilities to the GPU programming community.

Keywords: HPC, Rust, C++, NVIDIA, CUDA, PTX, Rust-CUDA, NSight Compute, performance.

# Acknowledgements

We would like to humbly thank Miquel Pericas, our supervisor, for his guidance throughout the project.

Viktor Franzén, Gothenburg, June 2022
Carl Östling, Gothenburg, June 2022

# Contents

# List of Figures

# List of Tables

# Listings

# 1

# Introduction

As the digitization of society is ramping up, applications such as machine learning and AI using high volumes of data for heavy, and sometimes real-time computations require more and more computational performance while minimizing energy consumption. Now that we are looking at the potential end of Moore's Law, computer scientists are focused on other ways of improving computational performance, such as hardware acceleration and programming language optimizations. As the GPU, graphics processing unit, is one of the most common hardware architectures for massively parallel problems due to its high core count, programming languages and frameworks have emerged to aid with implementing highly performant programs. For a long time, C, C++, and FORTRAN have been the status quo in HPC, high-performance computing. These languages provide a rich ecosystem, a large community, fine-grained manual control, and first-rate performance. They are also all in a continuous state of development, for example, the latest stable release of C++ is dated to December of 2020. The languages were however created many years ago, meaning that even though they are in active development they all have old heritages that constrain the language development. As other fields of programming adopt modern languages such as Python, JavaScript, Golang, and Rust, we believe it is reasonable to be critical of these older languages and challenge the status quo by evaluating other language candidates. In this thesis, we evaluate a potential candidate for HPC programming, Rust. Rust is a highly performant language with one of its core design principles being to match C in performance. Rust has a lot of promising features, such as memory safety and zero-cost abstractions. This means that it is easier to write code that has fewer bugs and memory errors since these errors will be found and prevented during compiling.

Because of Rusts high execution speed, along with its modern features, it is well suited for a comparison to challenge the programming languages in HPC's status quo. To this date, few studies exist comparing Rust with these languages, and even fewer that make comparisons regarding performance differences when doing GPGPU high-performance computing, thus there is a need for research in this area.

## 1.1 Goals and problem formulation

The goal of the thesis is to provide a scientific and accurate performance evaluation of Rust as a language for GPU programming targeting NVIDIA hardware. The novelty of the thesis is researching GPU programming considering *execution time*, *energy consumption* and, limitations/capabilities in Rust.

More specifically, we want to research the following questions:

- What is the relative performance of highly parallelizable Rust GPU programs compared to programs made with C++-CUDA?

- What are the capabilities/limitations in GPU programming using Rust compared to C++-CUDA that brings significant differences regarding performance?

By answering these questions, the thesis provides a comprehensive performance evaluation of Rust for GPU programming. The comparison to C++-CUDA ground the results in a benchmark against the status quo of GPU programming.

## 1.2 Delimitations

In order to scope the project to be reasonable and precise, some delimitations will be imposed.

**No CPU evaluation:** Even though it would be interesting to evaluate both single and multi-threaded programs ran on the CPU, this thesis will focus on evaluating the performance of programs ran on the GPU.

**No graphics:** Programs involving graphics rendering are well suited for the GPU and could definitely be a potential candidate for a program to profile. However, as they are likely more challenging to implement compared to a more straight-forward algorithm such as matrix multiplication, it will be out of scope for this thesis. There is also no benefit for our research goals to profile a graphical rendering program compared to something more simple.

**NVIDIA GPUs:** The code will only be guaranteed to run on NVIDIA GPUs with CUDA support. With this limitation, we are likely to have more consistent results and a less complex environment for implementation and profiling compared to if we were to include other GPU vendors.

## 1.3 Previous work

There is some previous work comparing Rust to C++ regarding performance. In the paper "Rust Language for Supercomputing Applications" [1] the authors Bychkov and Nikolskiy benchmark C++ and Rust in a few domains, such as matrix multiplication, linear algebra libraries, shared memory parallelism, and MPI. The authors found that the Rust code had similar performance compared to C++ while keeping the memory safety guarantees that come with the Rust language. However, the paper lacked a comparison for GPU programming, something that the authors also noted and suggested as further work.

Holk et al. [2] demonstrate in "*GPU Programming in Rust: Implementing High-Level Abstractions in a Systems-Level Language*" that it is possible to write GPU kernels in Rust through LLVM that can be used with the Rust OpenCL bindings. These kernels gained the advantages of the Rust language such as the memory safety and zero-cost abstractions while still retaining a performance that was comparable to manually programmed OpenCL kernels. The downside to their method is however that the GPU memory hierarchy such as shared memory is unavailable.

# 2

# Background

## 2.1 GPGPU programming

GPGPU is an acronym for *general-purpose computing on graphics processing units*, which is the concept of using a GPU for non-graphics related computing. The idea is to exploit the large number of cores available on the GPU, to execute massively parallel programs.

Algorithms that GPUs excel at are the ones that can be run at a massively parallel scale. Implementing massively parallel programs to run on a GPU in an optimal manner is no trivial task. There is plenty of research to be found on particular problems, such as in *A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures* by Cristóbal A. Navarro et al., 2014 [3] where implementations of select physical computational problems were designed with optimal parallel programming models in mind.

### 2.1.1 NVIDIA CUDA

For most modern NVIDIA GPUs created after 2006, the computing platform CUDA is available. NVIDIA describes CUDA as

> ...a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU [4].

CUDA supports the languages C, C++, and FORTRAN, but also various directives-based approaches and APIs. CUDA allows the programmer to divide their programs into both fine-grained and course-grained parallelism through hierarchies of thread groups, barrier synchronization, and shared memory.

CUDA's programming model is created around having code divided between the CPU (*host*) and the GPU (*device*) [5]. The CPU is responsible for the control-flow of the application, while the parts running on the GPU can be run in parallel on multiple threads without needing intervention by the CPU. The parallelism technique used is called SPMD (single program, multiple data) which means that each thread runs the same program in isolation, but with potentially different input data.

Traditional CUDA is normally represented in C/C++ files with the `.cu` file extension. NVCC (Nvidia Cuda Compiler) can compile these files into an executable binary.

### 2.1.2 PTX

PTX is a virtual machine and instruction set architecture (ISA) that enables NVIDIA GPUs for data-parallel computing [6]. Programs written in CUDA are compiled down to PTX instructions, which are then translated into binary code by the NVIDIA driver to be executed by the GPU. By design, it is efficient on NVIDIA GPUs and supports their many features, both across single-GPU and many-GPU systems.

### 2.1.3 Kernels and their launch parameters

CUDA allows the definition of kernel functions. A kernel, when called, is executed `N` number of times by `N` threads on the GPU whereas a regular function is only called once, on the CPU. As an example, in C++, a CUDA kernel is defined as in Listing 2.1 with the keyword `__global__` [4].

```
__global__ void copy_array(float *a, float *c)
{
    int i = threadIdx.x;
    c[i] = a[i];
}
```

**Listing 2.1:** Example of a CUDA kernel in C++

To keep track of threads and their memory space, an abstract hierarchy is formed and each thread is assigned to a *block*, which holds a maximum of 1024 threads. Each *block* is then assigned to a *grid* illustrated in Figure 2.2. This gives the threads a potential three-dimensional structure. By utilizing the thread, *block*, and *grid* positions, a unique ID for each thread can be calculated. In the case of a flat structure with only one *block*, the ID is the $x$ position of the thread. For a 2D structure where *blocks* are organized in a row, the ID is calculated as $x + y * D_x$ where $D_x$ is the *block's* $x$ position in the *grid*. The ID of a thread in a 3D structure is calculated as $x + y * D_x + z * D_x * D_y$ [4].

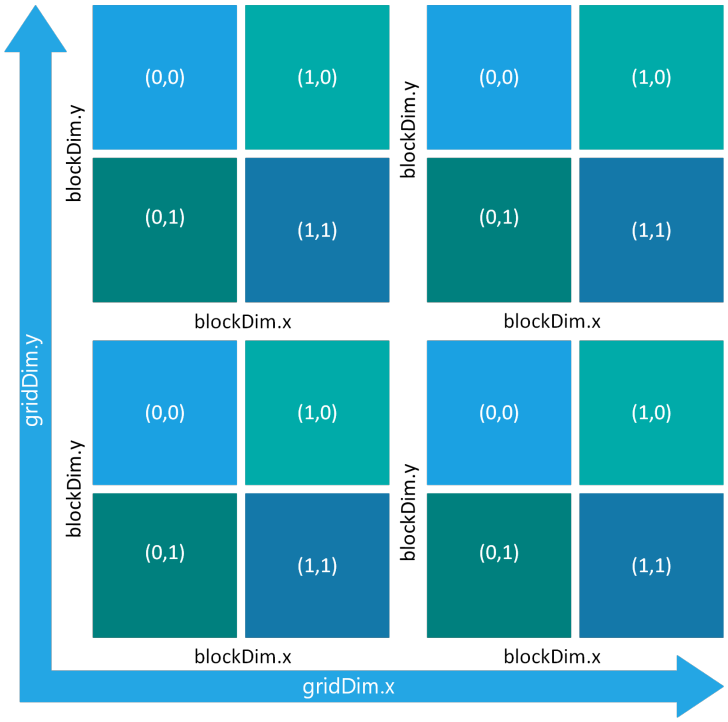


**Figure 2.1:** Illustration of a 4x4 CUDA grid

Before a kernel is launched, the kernel code and the data required for the computations must be copied over from the *host* (CPU) to the *device* (GPU). After the kernel has finished its execution, the results calculated in the *device* memory are copied to the *host*. This is done manually by the programmer at the selected program stage by using built-in CUDA functions such as `cudaMemcpy(...)`.

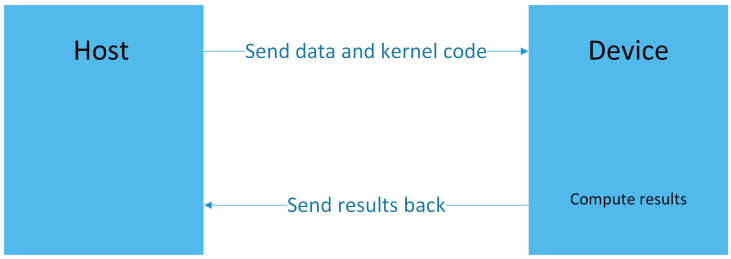


**Figure 2.2:** Host to Device relationship during kernel execution

## 2.2 C++ & Rust

C++ is a language that was created in 1979 by the programmer Bjarne Stroustrup [7]. While Stroustrup was working with one of the first languages to support object-oriented programming, he felt that the paradigm was useful but the language was too slow. This prompted him to begin working on C++, or "C with Classes" as it was called back then. C++ is a compiled, strongly-typed language that has

support for many different paradigm choices such as procedural and object-oriented programming [8]. It is a language that gives the programmer a lot of control of the execution, meaning that it requires a skilled programmer to do things correctly, but when done correctly it is one of the fastest languages currently existing.

Rust is a relatively new language and is seeing rapid updates to its capabilities. First created in 2010 by Graydon Hoare while working at Mozilla, Hoare wanted a language with more concurrency and increased memory safety [9]. The language grew quickly and in 2015 the first stable release of Rust was released. Its syntax is similar to that of C++. The language itself has a high focus on providing high-performing code while also being reliable regarding memory safety [10]. It does this by having no runtime or garbage collector while also using an ownership model that the compiler uses to find memory errors at compile-time.

Even though the Rust compiler enforces memory safe code, it sometimes is necessary to work with systems that are inherently memory unsafe. To enable this it is possible to turn off the compile-time enforcement of safe code with what is called `unsafe` Rust [11]. `Unsafe` Rust allows the programmer to work with raw pointers, enabling the programmer more freedom at the cost of potential memory related bugs.

Rust has three different release channels called *nightly*, *beta* and *stable* [12]. The *nightly* release channel creates one release each night which has the following form *nightly-2022-03-15* and can be seen as a rolling release. Releases to the *stable* channel are made each 6:th week, while the *beta* releases contain the features that are going to go into the next *stable* release.

## 2.3 GPGPU programming in Rust

As Rust is a relatively new language, GPU programming has yet to mature with official support from vendors such as NVIDIA. The surrounding community is working hard on providing open-source packages (crates) to the ecosystem and we see several interesting projects mature for GPU programming in Rust with some of the most prominent and active ones being `rust-gpu` by Embark Studios [13] and `arrayfire-rust` by ArrayFire [14]. `rust-gpu` allows writing GPU kernels for OpenGL and SPIR-V, the open-source graphics API by Khronos Group, and aims to provide a fully community-driven open-source GPU programming package [13, 15]. As it uses OpenGL, it mainly focuses on shaders for computer graphics but does allow for compute shaders. `arrayfire-rust` is a wrapper for CUDA, OpenCL and CPU HPC programming in Rust. ArrayFire is a well-established software library that exists for multiple programming languages which provide hundreds of functions for parallelism, GPU programming, machine learning, and more. With all these features, it is by no means light-weight and it requires installation of software larger than 1GB in size [16].

### 2.3.1 The Rust CUDA Project

One of the more recent libraries that enables writing CUDA kernels in Rust is the `Rust-CUDA` [17]. It aims to make Rust a top candidate for fast computations using the CUDA Toolkit. It was created in order to improve the previously existing alternatives that were all using the LLVM PTX backend to generate the NVIDIA kernels. The authors of `Rust-CUDA` saw multiple problems with using this backend, one of which was that many of the common Rust operations would not currently be supported by the backend because of missing features or bugs [17, 18]. This would in turn make the generated PTX invalid, even for smaller programs. This uncertainty means that it is too unstable to be used in production. Another reason the authors found was that because of linking issues with dynamically linked library (dylib) files, one could not run the backend on windows machines without using a specialized way of building LLVM [18].

Instead, the authors have created a library inside `Rust-CUDA` called *rustc_codegen_nvvm* that compiles the kernels written in Rust into a format called NVVM IR. NVVM IR is a compiler intermediate representation that represents GPU kernels, in this case CUDA kernels, and is based on LLVM IR [19]. More specifically, NVVM IR is a subset of LLVM IR, meaning that a program represented in NVVM IR will always be a valid LLVM program, but the same statement is not true when it comes to LLVM IR and NVVM. Feeding NVVM IR into the NVIDIA NVVM compiler generates PTX code that can be run on the GPU.

The project works by targeting the CUDA Driver API. This is because it gives more fine-grained control over parts of the execution compared to using the Runtime API [18]. One downside of this is that some CUDA libraries, such as cuRAND, will not work since it is dependent on the usage of the CUDA Runtime API [20].

The requirement for the project is a CUDA version of 11.2 or higher [21]. It also uses LLVM 7.x, meaning minor versions between 7.0 and 7.4. The reason for this is that NVVM IR is based on LLVM 7.0.1 [19]. It also requires the developers to use the *nightly* version *nightly-2021-12-04* of Rust when building the GPU kernels through *rustc_codegen_nvvm* [21]. All of the other libraries used to create the *host* code has no such requirement. Another requirement is that the kernels written in `Rust-CUDA` are required to be `unsafe`.

Some features in Rust are particularly useful for writing code that targets CUDA, two of which are RAII and Results [22]. RAII, which stands for *Resource Acquisition is Initialization*, is a feature that exists in Rust used to ensure that no memory leaks occur [23]. Variables in Rust have ownership over their own resources on the heap, when one of the variables goes out of scope its destructor is automatically called, freeing the resources it is owning without having to manually free memory. This in contrast to for example C++ where one needs to call both `free(variable)` and `cudaFree(gpu_variable)` to free up the resources, which if forgotten leads to memory leaks. `Result` is a Rust data type that makes it easy to create error handling where errors can easily be propagated up the call stack [24]. The compiler forces the programmer to explicitly handle the `Results` that occur in the code, meaning

that it is hard for unexpected errors to kill the execution. This data type is used heavily in `Rust-CUDA` to ensure proper and reliable error handling to avoid segfaults and make the code more robust [22].

As mentioned, `Rust-CUDA` is used as the main GPU programming tool for evaluating Rust in this thesis. It was selected based on its native Rust implementation without acting as a wrapper, such as `arrayfire-rust`. It also compiles the kernels directly to PTX code which allows for comparison with PTX compiled from C++ code. Further, it is CUDA only which fits the scope of this thesis. The version used in this thesis is *0.3*.

## 2.4 Profiling

Profiling is analogous to benchmarking and captures data during the execution of a program. The data includes metrics such as *execution time*, memory usage, core usage, and *energy consumption*. Running a profiler does impact the performance of the profiled program as it is essentially running a separate program simultaneously which contributes to the load on the hardware. This needs to be taken into account when selecting a profiler and when evaluating the results.

When it comes to profiling GPU programs there are a few different tools available. CUDA programs developed for NVIDIA GPUs can be profiled with NVIDIA Nsight Compute [25]. This profiler works on all NVIDIA GPUs that support CUDA. With it, one can profile different performance metrics of the GPU and also get a visual representation of API calls, kernel launches, and memory movement. The Nsight profiling tool is the latest generation of profiling tools from NVIDIA and developers are recommended to migrate from the older NVIDIA Visual Profiler and NVProf tools [26, 27].

In Figure 2.3 the details page of Nsight can be seen. Here information revolving for example *execution time*, cache misses, instruction counts, and the roof-line model can be found. It can also auto-detect issues in the code and displays a warning accompanied by a text explaining the problem.

The source page seen in Figure 2.4 displays detailed information regarding the execution of the kernel. For example, it shows the instructions executed, the number of live registers, and all of the warp stalls along with where they occurred. Using this page gives insights into the code execution in an efficient way.

Unfortunately, all of the metrics from the older tools are not possible to capture through the Nsight tool. Examples of this are metrics regarding power usage, GPU temperature, and GPU fan speed, which were possible to profile with the NVProf tool by using the system profiling option.

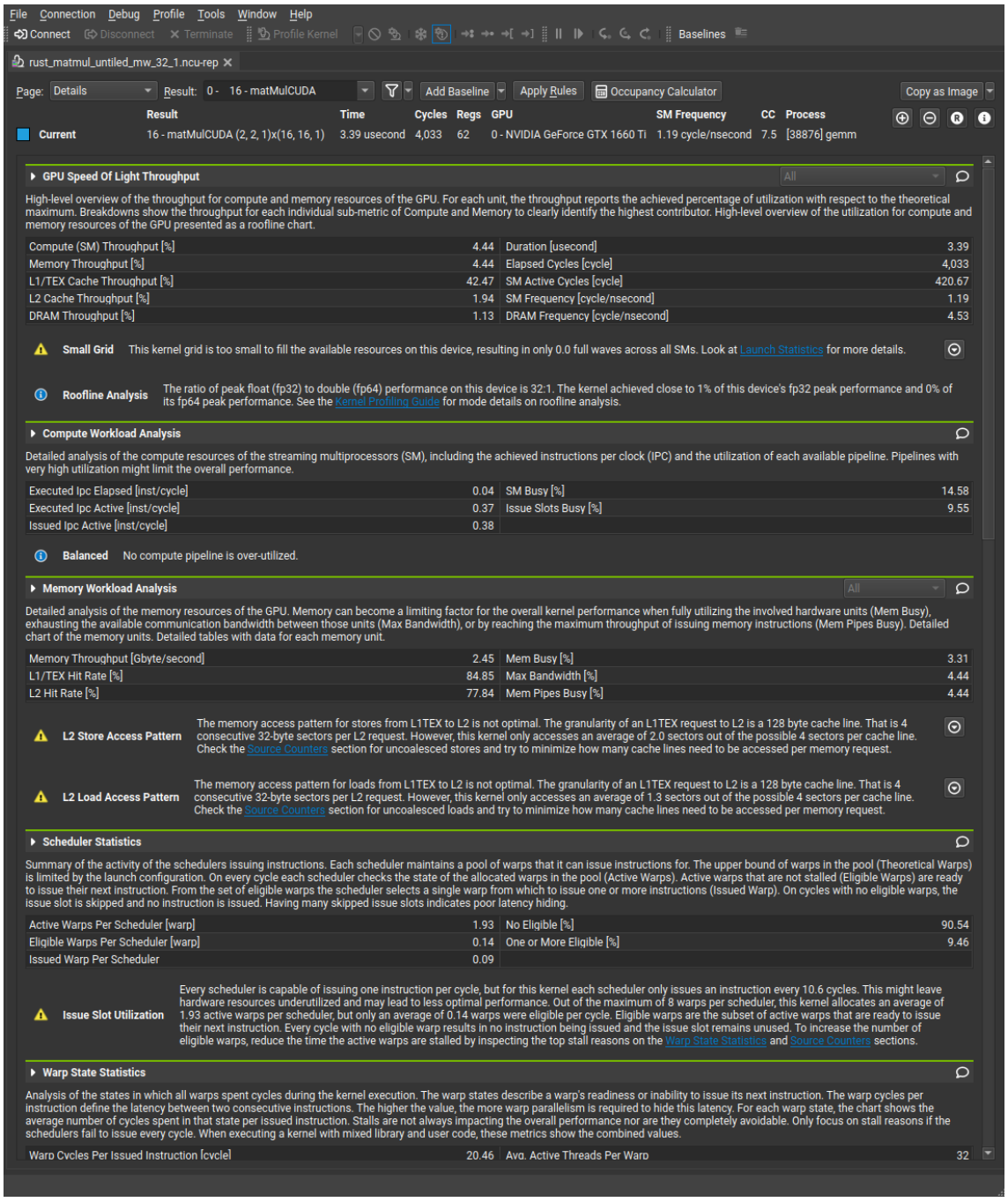


**Figure 2.3:** The details page inside of NVIDIA Nsight

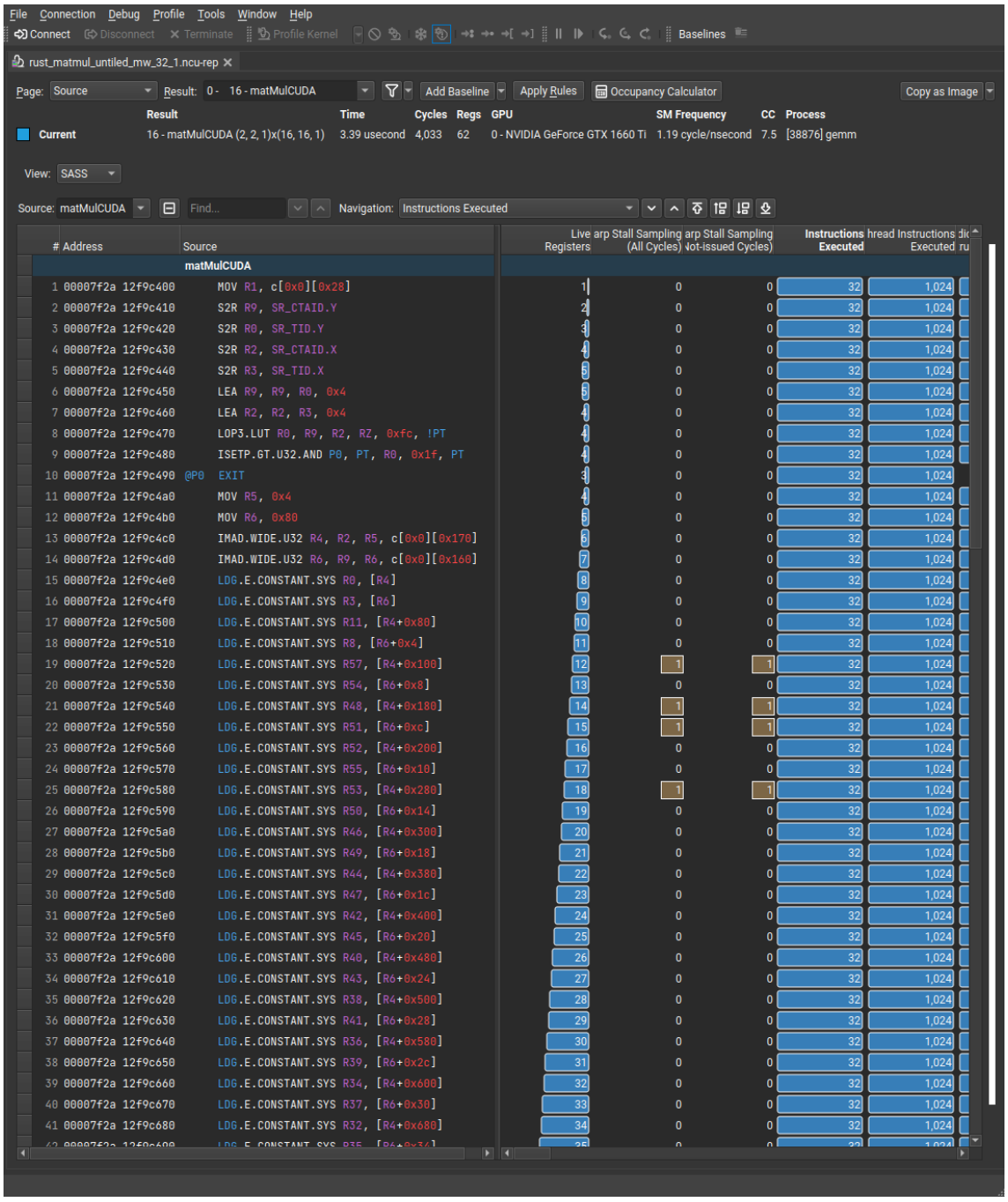


**Figure 2.4:** The source page inside of NVIDIA Nsight

# 3

# Method

In order to evaluate the performance of Rust relative to C++ for GPU programming, selected programs were implemented in both languages and then profiled to capture metrics that allow for direct comparison. Through further analysis of collected data, key insights were revealed describing some reasons why these performance differences occur. These insights were revealed by looking at the comprehensive data captured by and viewed directly in Nsight Compute. Then, modifications were done to the code in order to pinpoint and prove certain insights.

## 3.1 Programs & Implementation

For this thesis, matrix multiplication and array copying were selected as programs to implement for experimentation. These were implemented in both Rust and C++ with the aim of both implementations to be as comparable as possible, which is important since we want the profiled metrics to depend on language/library differences only. In order to achieve this, the programs use similar data structures and data types correlated line-by-line where applicable. Overhead in terms of code not related to the core functionality is minimized. Further, the programs selected are differentiated in that they should, for example, be either *compute-bound* or *memory-bound*. By having programs that can be categorized differently, the results can be more insightful.

### 3.1.1 Implementation strategy

The approach for implementing comparable programs in the two languages was straightforward. Each program was first implemented in C++ in simple and clear code, minimizing overhead when possible. From this implementation, each program was recreated in Rust line-by-line. When this was not possible, for example when a different data type was required, efforts were made to make the required adjustments as similar as possible to the original C++ code. To facilitate this, the `Rust-CUDA` documentation was used to convert C++ CUDA API calls to their corresponding Rust function.

One larger difference when implementing the programs in Rust is that we need to split the *host* and *device* code into two separate crates, where the *host* crate imports

and compiles the *device* crate during its compilation. In the C++ implementations, both *host* and *device* code resides in the same `.cu` source file, effectively requiring less code/structure. This difference is however purely structural and should have no effect on the runtime performance.

### 3.1.2 Classifications

Computational problems can be classified differently, for example, they can be either *compute-bound* or *memory-bound*. The *execution time* of a *compute-bound* problem is primarily limited by the throughput of the hardware, whilst a *memory-bound* problem is limited by the memory bandwidth of the hardware.

### 3.1.3 Matrix multiplication

Matrix multiplication is a classic computational problem well suited for the high level of parallelism offered by GPUs. The reason for this is that each value in the resulting matrix can be computed without depending on previous computations. Each output element is calculated by multiplying elements in each respective column and row of the input matrices as illustrated in Figure 3.1. Matrix multiplication is classified as a *compute-bound* problem as data accesses are kept low in relation to the arithmetic operations required for computations.

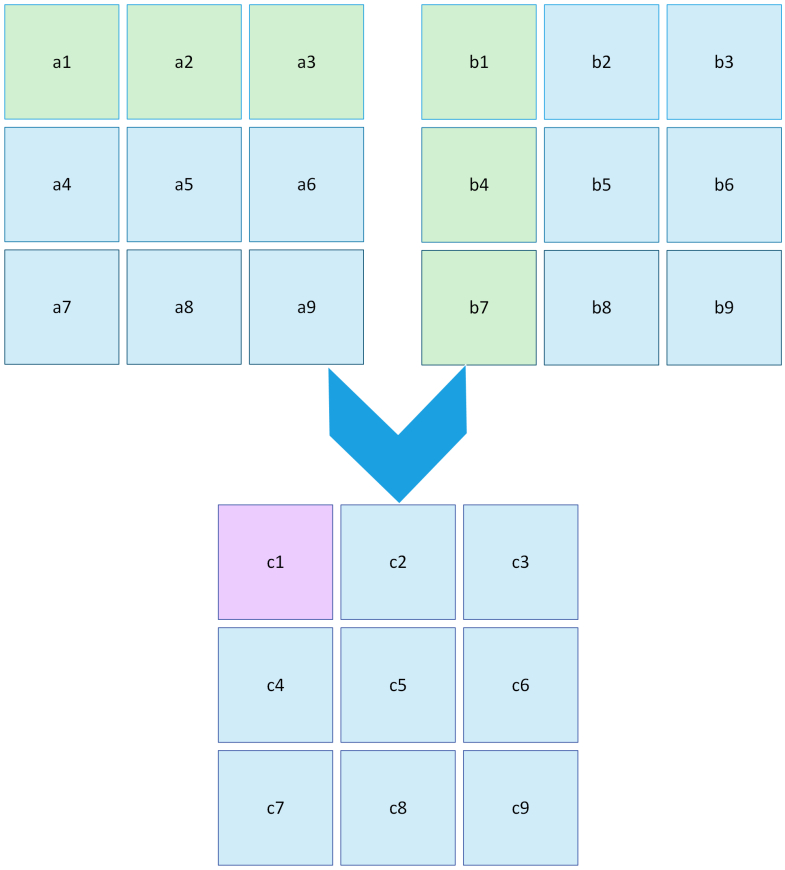


**Figure 3.1:** Illustration of a 3x3 matrix multiplication

Two different implementations of matrix multiplication are used for the experiments in this thesis. One version uses tiling, which makes use of shared memory. The other version is much simpler and uses no shared memory. By using these two different versions, we can analyze how Rust and `Rust-CUDA` perform when utilizing shared memory.

#### 3.1.3.1 Tiling for matrix multiplication

Tiling is a technique used to reduce the number of global memory accesses, which is desired as these accesses are slow compared to the *block*-level memory called shared memory. This ultimately boosts the efficiency of the kernel since it is a higher-performing type of memory regarding access speed. In tiling for matrix multiplication, the idea is to have each thread in a *block* be responsible for loading a certain amount of input elements from the input matrices into the shared memory. The number of elements loaded from each matrix is dependent on the matrix size and launch parameters of the kernel in terms of *grid* and *block* size. Once the elements are loaded into shared memory, they become available to all threads within a *block*. Without tiling, each thread would have to access every element required for its computation from global memory, which would be significantly slower.

### 3.1.4 Array copy

Array copy is a simple program with its task being to replicate data from one array to another. Each input element is simply duplicated to the output array, as illustrated in Figure 3.2. In its most basic form, this program does not need to perform any arithmetic computations and is thereby considered a *memory-bound* problem. As each copy operation can be done in isolation without depending on other copy operations being completed, this problem is well suited for parallelism and execution on a GPU.

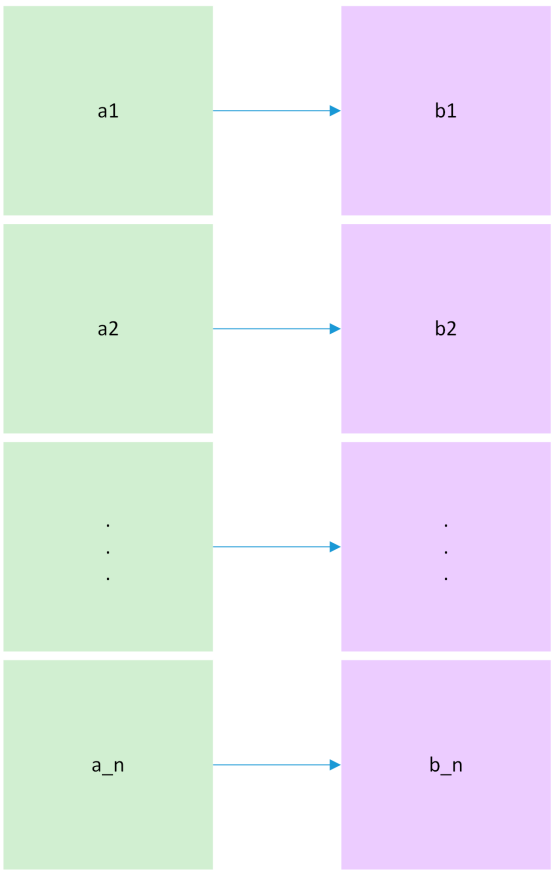


**Figure 3.2:** Illustration of array copy

## 3.2 Performance metrics

Performance is an abstract concept that could mean a lot of things. In the context of HPC, it usually refers to both *execution time* and *energy consumption*. To make it even more concrete, the most important metrics that we want to base our evaluation on are:

**Execution time** The most basic and absolute metric for evaluating performance is *execution time* which is the time required to finish the execution of a program and is measured in seconds. This can also, over several executions, allow us to gauge if

the *execution time* is repeatable and predictable.

**Energy consumption** Another metric is the *energy consumption* required to finish the execution of a specific program which is measured in watt-hours or joules. Thus, this will require the wattage over time for the execution of a program. Further, the average, peak, and minimum wattage during the execution will be recorded. The energy consumed to execute a program is more relevant than ever. In certain applications such as embedded, where we have a limited amount of power, a lower *energy consumption* can be critical.

**Executed Instructions** The number of different low-level instructions executed by the GPU during the execution of a program can indicate or highlight key differences between C++ and Rust. While not directly linked to performance in the way *execution time* or *energy consumption* is, it can still be an important metric for finding differences between implementations.

## 3.3 Profiling

In order to capture performance data that can be used to compare the different implementations, a selection of profiling tools is used. Where applicable, profilers agnostic to the programming language are used so that the results are more comparable, as different tools will provide a varying degree of accuracy in the collected data.

The main profiling tool used in this work is NVIDIA Nsight Compute which is used to capture the vast majority of all results. Because of its limitation regarding capturing data about power consumption as mentioned in Section 2.4, the NVProf tool is also used to capture this metric. During profiling these tools are run separately, meaning that the results are captured from different executions of the same compiled binary so as to not interfere with each other.

Compiling and profiling the different programs were made with bash scripts, ensuring that compilation is done in the same way every time. It also enables the possibility to run and profile the same binary multiple times in an automated fashion to average out any outliers during the profiling. The compilation of C++ programs was made through `nvcc` with default optimization flags (the default optimization level for kernel code is 3 [28]). The Rust compilation was made through the package manager Cargo with the command: `cargo build -release`. The `-release` flag sets the optimization level to 3, which is the highest one available [29]

The generated binaries are then profiled first by NVIDIA Nsight, through the `ncu -set full <binary>` command, for the main profiling. By using the `-set full` option, the profiler collects the full set of profiling metrics [30]. System profiling is then made with NVProf through the command `nvprof -system-profiling on <binary>`, which generates data that can be used for calculating the *energy consumption*. In both these commands `<binary>` means the path to the generated binary. Flags related to the output locations of the results have been omitted for

clarity.

Data was captured on varying problem sizes for each problem. For the tiled matrix multiplications, two main variables were used: the matrix width and the block width. The matrix width is the width of the two input matrices and the output matrix. Block width means the width of each block inside the CUDA compilation. To ensure that data was collected for both small and large problem sizes, the matrix widths chosen in this thesis were 32, 128, 512, and 2048. Each size is four times as large as the previous one giving a uniform spread of sizes. Attempting to profile a size of 8192 was unfortunately too slow, which is the reason the maximum is 2048. Similarly, the block widths chosen were 4, 8, 16, and 32. Since the maximum thread count allowed in a CUDA block is 1024, a larger block width than 32 would not make sense since it is the maximum, any larger value would not execute successfully. This applies to the untiled matrix multiplication as well, with the main difference that the block width is not variable. The reason for this is that since we are not using any shared memory in this kernel, varying the block width does not have as much of an impact on performance, as it does with the tiled matrix multiplication. Instead, a good default of 16 is used.

The only variable used for the array copy kernel was the length of the array to copy. As before, to profile both small and larger problem sizes, the array lengths were chosen to be 1024, 32768, 1048576, and 33554432 (each length is 32 times as large as the previous one).

## 3.4 Hardware

The profiling and collection of results were done through multiple computers at different levels of computing power to ensure that the results were not hardware-specific. The specifications of these machines can be found in Tables 3.1 and 3.2.

| **Operating system** | Ubuntu 20.04.3 LTS |
|---|---|
| **CPU** | Intel Core i5 9600K 3.7 GHz 9MB |
| **GPU** | ASUS GeForce GTX 1660 Ti TUF Gaming X3 OC 6GB |
| **RAM** | Corsair 16GB (2x8GB) DDR4 2666Mhz CL16 Vengeance |
| **NVIDIA Driver version** | 510.39.01 |
| **CUDA version** | 11.6 |

**Table 3.1:** Description of Machine 1

| **Operating system** | Ubuntu 20.04.3 LTS |
|---|---|
| **CPU** | Intel Core i7 8700K 3.7 GHz 12MB |
| **GPU** | MSI GeForce RTX 2080 Ti 11GB DUKE OC |
| **RAM** | Corsair 32GB (2x16GB) DDR4 2666Mhz CL16 Vengeance |
| **NVIDIA Driver version** | 510.60.02 |
| **CUDA version** | 11.6 |

**Table 3.2:** Description of Machine 2

# 4

# Results

This chapter presents selections of collected data and the experiments with code that revealed insights as to why performance differs. The CUDA kernels implemented in both C++ and Rust can be found in Appendix A. For the full source code please look at https://github.com/vifraa/Evaluation-of-Rust-for-GPGPU-high-performance-computing.

All results collected are averaged over twenty executions for each experiment unless stated otherwise. The results shown in the chapter are profiled on Machine 1 (described in Table 3.1). Profiled data for the same experiments on Machine 2 can be found in Appendix B.2.

## 4.1 Performance metrics of implemented programs

This section presents collected metrics for the experiments regarding the line-by-line translate code. More detailed graphs regarding power usage can be found in Appendix B.

### 4.1.1 Execution time

*Execution time* is a staple benchmark in HPC and is a common metric for evaluating the performance of a program.

Figure 4.1 and Table 4.1 display the differences in *execution time* for the tiled matrix multiplication experiments.

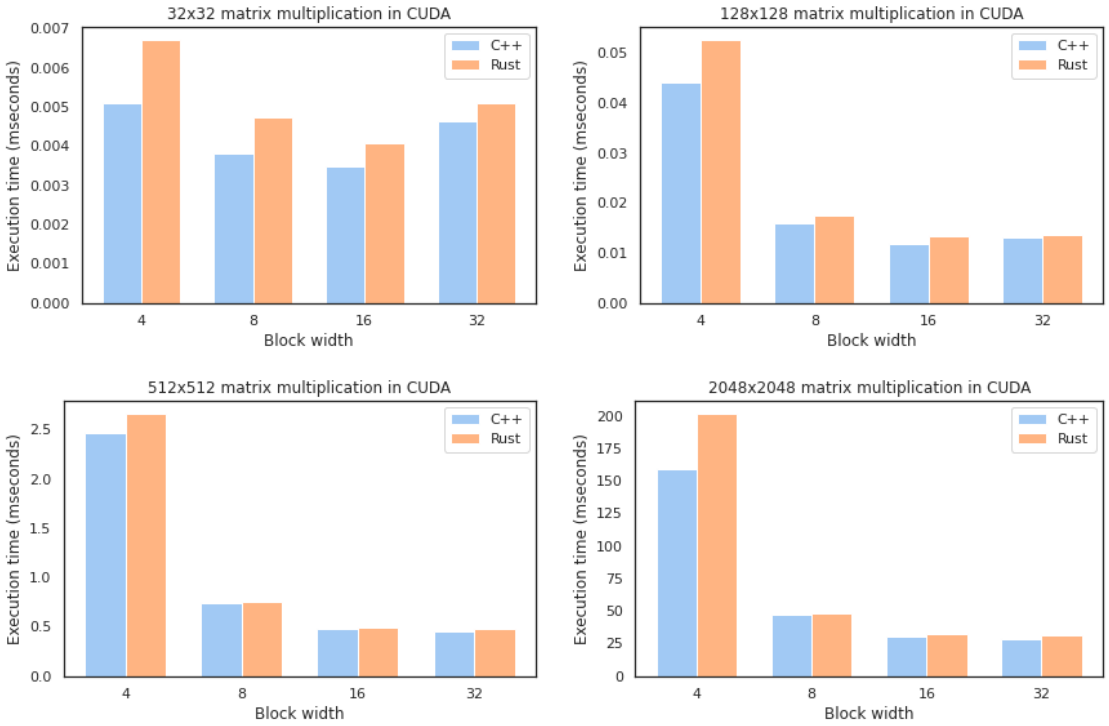


**Figure 4.1:** Difference in execution time for Rust and C++ for tiled matrix multiplication with different matrix and block widths.

| Matrix Size | Average Speedup |
|---|---|
| 32x32 | 1.21 |
| 128x128 | 1.11 |
| 512x512 | 1.04 |
| 2048x2048 | 1.13 |
| All combined | 1.12 |

**Table 4.1:** Average speedup of C++ over Rust for all experiments within a given matrix size for tiled matrix multiplication.

It is clear from Figure 4.1 that C++ outperforms Rust in all experiments for this kernel in terms of *execution time*. In Table 4.1, the speedups for C++ over Rust are reported per matrix size over all experiments. We see that the average speedup of C++ over Rust for all experiments of the matrix multiplication kernel is 1.15.

The captured results for matrix multiplication without tiling show larger performance differences as can be seen in Figure 4.2 and Table 4.2. The main difference between these versions is the tiling which uses shared memory. This might indicate that there is a larger performance difference while accessing global memory than with shared memory.

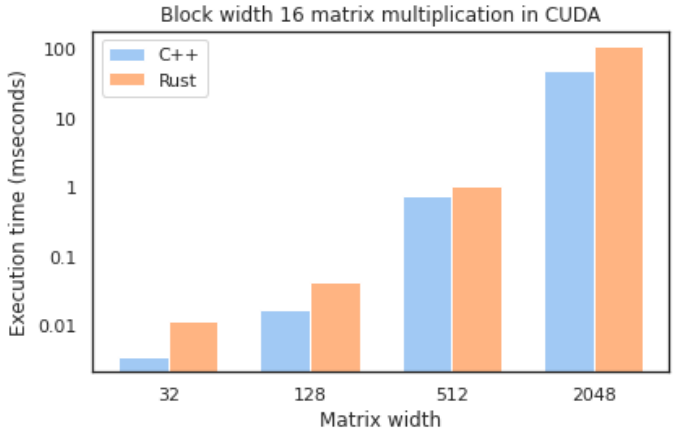


**Figure 4.2:** Difference in execution time for Rust and C++ for untiled matrix multiplication with different matrix widths.

| Matrix Size | Average Speedup |
|---|---|
| 32x32 | 3.33 |
| 128x128 | 2.52 |
| 512x512 | 1.37 |
| 2048x2048 | 2.24 |
| All combined | 2.37 |

**Table 4.2:** Average speedup of C++ over Rust for all experiments within a given matrix size for untiled matrix multiplication.

Metrics collected during profiling of the array copy kernel can be seen in Figure 4.3 and Table 4.3. Here we see next to no performance difference. A large difference that differentiates the copy kernel compared to the matrix multiplication is as mentioned in Section 3.1.4 that copy is a *memory-bound* program compared to the *compute-bound* matrix multiplication.

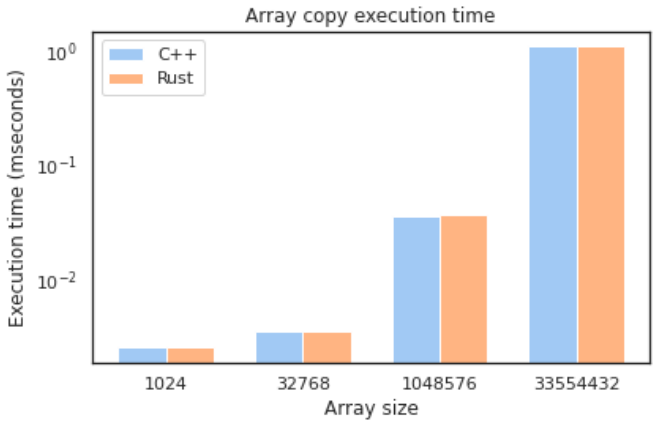


**Figure 4.3:** Difference in execution time for array copy.

| Array size | Average Speedup |
|---|---|
| 1024 | 1.008 |
| 32768 | 0.997 |
| 1048576 | 1.004 |
| 33554432 | 1.0 |
| All combined | 1.002 |

**Table 4.3:** Average speedup of C++ over Rust for all experiments of the array copy kernel.

### 4.1.2 Energy consumption

*Energy consumption* is another staple benchmark in HPC. It is often closely related to *execution time*, as time is used in the calculation of *energy consumption*.

As can be seen in Figures 4.4, 4.5, and 4.6, the *energy consumption* results look very similar to the *execution time* results in the previous chapter. Since there is such a close relationship between the *execution time*, this is to be expected.

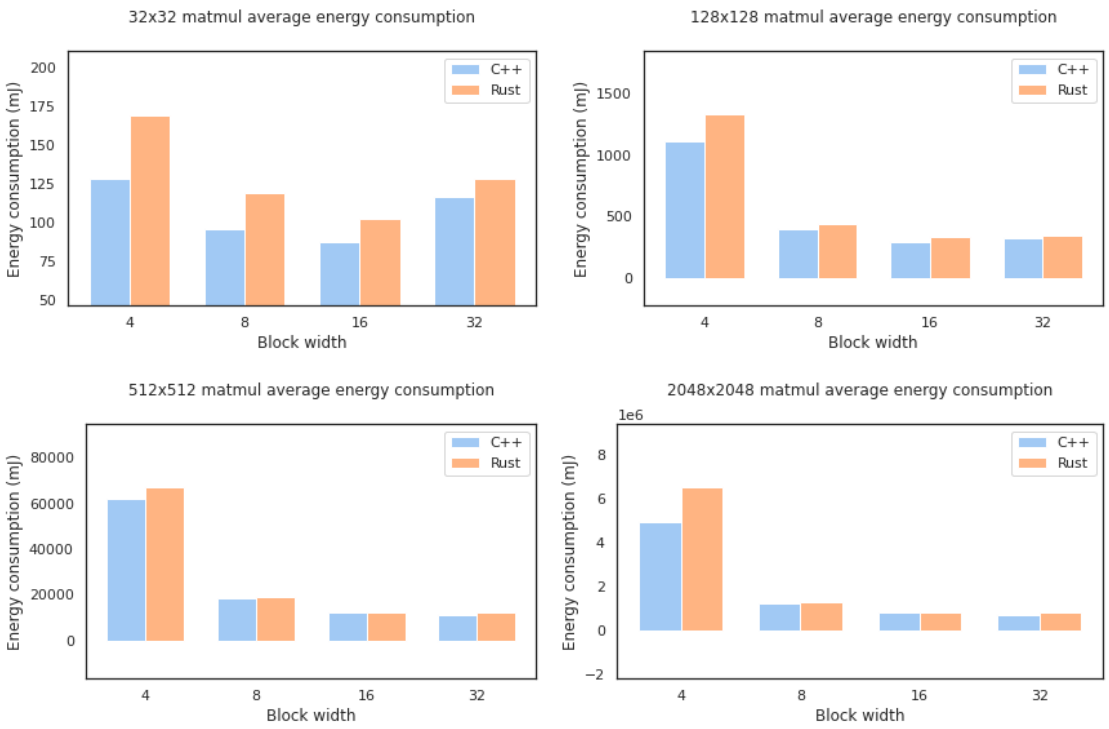


**Figure 4.4:** Difference in average energy consumption for Rust and C++ for tiled matrix multiplication with different block widths.

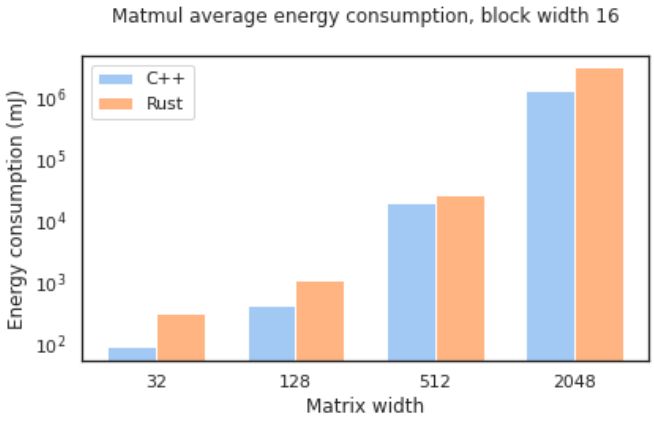


**Figure 4.5:** Difference in average energy consumption for Rust and C++ for untiled matrix multiplication with different different matrix sizes. Y-axis in logarithmic scale.

The consumption difference for the copy kernel is minimal as can be seen in Figure 4.6.

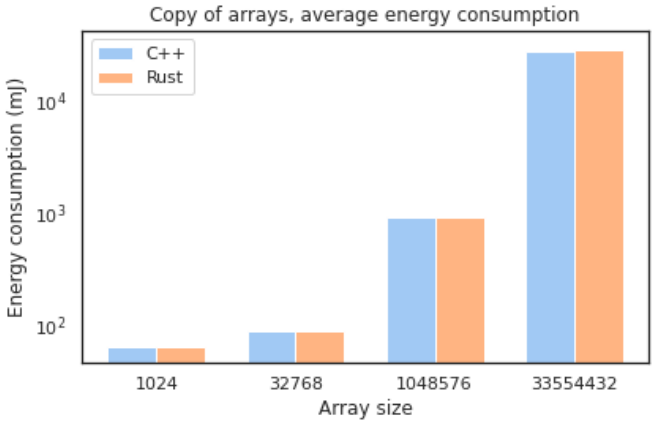


**Figure 4.6:** Difference in average energy consumption for array copy. Y-axis in logarithmic scale.

### 4.1.3 Executed instructions

Nvidia Nsight Compute provides data of the instructions run by the GPU during the execution of a kernel. This data gives insight into how the C++/Rust implementations differ in the executions at the lowest level.

In Tables 4.4, 4.5, and 4.6 we can see a subset of the executed instructions for the tiled matrix multiplication, untiled matrix multiplication, and the copy kernels. Instructions with a relatively low instruction count are omitted from the table for the matrix multiplication tables.

| Instruction Name | Rust Executed Instructions | C++ Executed Instructions |
|---|---|---|
| LDS | 335 544 320 | 335 544 320 |
| FFMA | 268 435 456 | 268 435 456 |
| IMAD | 42 860 544 | 68 157 440 |
| ISETP | 109 182 976 | 84 279 296 |
| MOV | 34 078 720 | 67 633 152 |
| LEA | 92 667 904 | 786 432 |
| Total | 1,177,944,064 | 1,027,735,552 |

**Table 4.4:** Selected instructions and their number of executions for implementations of tiled matrix multiplication in C++ and Rust for matrix width of 2048 and block width of 16.

| Instruction Name | Rust Executed Instructions | C++ Executed Instructions |
|---|---|---|
| ISETP | 1 107 427 382 | 17 956 864 |
| IMAD | 639 238 144 | 269 352 960 |
| IADD3 | 637 927 424 | 67 371 008 |
| BRA | 570 425 344 | 17 694 720 |
| LDG | 536 870 912 | 536 870 912 |
| FFMA | 268 435 456 | 268 435 456 |
| Total | 3 762 421 760 | 1 179 910 144 |

**Table 4.5:** Executed instructions for implementations of untiled matrix multiplication in C++ and Rust for matrices with size 2048.

| Instruction Name | Rust Executed Instructions | C++ Executed Instructions |
|---|---|---|
| IMAD | 98 304 | 98 304 |
| S2R | 65 536 | 65 536 |
| ULDC | 65 536 | 0 |
| SHF | 32 768 | 0 |
| ISETP | 98 304 | 32 768 |
| EXIT | 65 536 | 65 536 |
| STG | 32 768 | 32 768 |
| MOV | 0 | 65 536 |
| LDG | 32 768 | 32 768 |
| Total | 524 288 | 393 216 |

**Table 4.6:** Executed instructions for implementations of array copy in C++ and Rust for arrays with size 1048576.

Looking at the total amount of instructions executed for the three programs we see that the Rust kernels are executing significantly more than their C++ counterpart. As seen in Table 4.4, the tiled matrix multiplication have almost 150 million more total instructions. This kernel is quite a large one, and it might be hard to pinpoint

exactly where these instructions come from, however looking at the array copy kernel which is one of the simplest kernels possible, we see that even there the Rust version executes about 33% more instructions compared to the C++ version as seen in Table 4.6. Interesting enough, the worst performing Rust kernel is the matrix multiplication without tiling. In Table 4.5, we see that it executes around 2.5 billion more instructions, which is 3.2 times as many as the C++ kernel.

## 4.2 Differences affecting performance

With a baseline of the performance for line-by-line comparable kernels as seen in the previous section, it is important to understand why the performance differs the way that it do. This section explains and discuss the findings that impacted the performance.

### 4.2.1 Bounds checking

Since the matrix multiplication without tiling has a much simpler kernel, it showed differences in the generated PTX code clearer. One thing that stood out was that for each memory access to the global memory, in this case to the two input arrays, a bounds-check was added that was not present in the C++ version. If the access would be at a memory address outside of the targeted array, a trap instruction would be executed which essentially aborts all execution and produces an interrupt signal to the *host*. An example of this can be seen in Listing 4.2 on lines 7 to 9. This seems to be a feature of Rust to ensure that writes are not made to unintended memory locations, but unfortunately it has large performance downsides. By manually removing these bounds-checks from the PTX and re-profiling the kernel for a single 32x32 experiment, the Rust kernel went from being much slower compared to the C++ version to instead perform at the same level. It was also observed that this removed the large majority of all branch instructions that were run during the kernel execution which previously was a large difference. The reason that these appears seems to be that the regular indexing operation (`a[index]`) in Rust automatically add bounds checks. However, by changing these index operations to instead use the function `a.get_unchecked(index)`, these can effectively be removed since this function does not do the bounds checks. The `get_unchecked()` function requires the code to be placed in a code block marked as `unsafe` which all `Rust-CUDA` kernels already are.

By profiling the kernels with this change a greater performance was observed from the Rust kernels as can be seen in Figures 4.7, 4.8, and 4.9. With these changes, Rust now performs better or at the same level as the C++ counterpart in all of the cases. Comparing the relative *execution times*, seen in Tables 4.8, 4.9, and 4.10, a similar performance regarding the *execution time* can be seen for the different languages, except for the 32x32 matrix multiplications where C++ now performs worse than Rust. This can be seen as a huge difference, especially for the tiled 32x32 matrix multiplication, however it is worth noting that these kernels have an execution time

that is in the 3-5 microsecond range. Because of this, a small constant overhead in one implementation can cause a quite large percentage difference that is extra visible in the smaller matrix sizes, however won't be noticeable as the problem size increases, as can be seen in our experiments.

The *energy consumption* results follow the same pattern as with the *execution time*, and their graphs are therefore omitted from this section. They can however be found in Appendix B.1.1.

In Table 4.7, the total executed instructions can be seen for the matrix multiplication without tiling with a matrix width of 2048. Previously it was the worst-performing kernel, with removed bounds checks it is now executing 84% fewer instructions than the C++ version. This means that by removing the bounds check, the kernel is executing about 2.7 billion instructions less than it did with the bounds checks.

| Rust Total Instructions | C++ Total Instructions |
|---|---|
| 993,394,688 | 1,179,910,144 |

**Table 4.7:** Total executed instructions for untiled matrix multiplication with matrix of size 2048 without bounds checking.

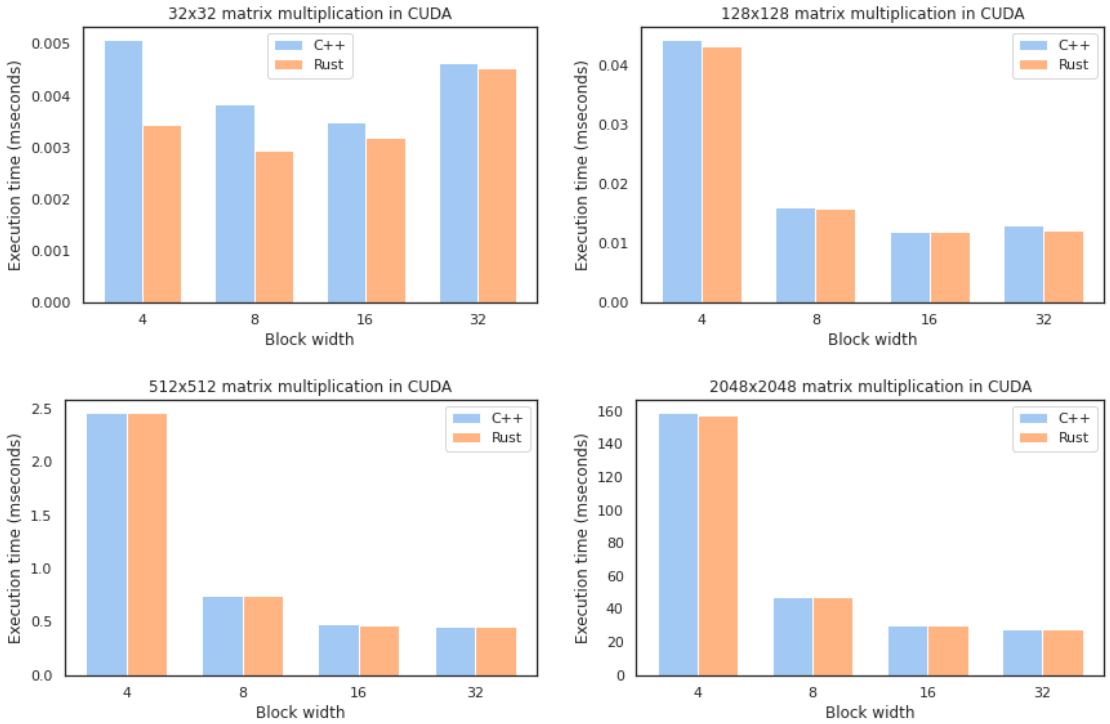


**Figure 4.7:** Difference in execution time for Rust and C++ for tiled matrix multiplication with different matrix and block widths without bounds checks when accessing global memory.

| Matrix Size | Average Speedup |
|---|---|
| 32x32 | 0.84 |
| 128x128 | 0.97 |
| 512x512 | 1.0 |
| 2048x2048 | 1.0 |
| All combined | 0.95 |

**Table 4.8:** Average speedup of C++ over Rust for all experiments within a given matrix size for tiled matrix multiplication with removed bounds checks.

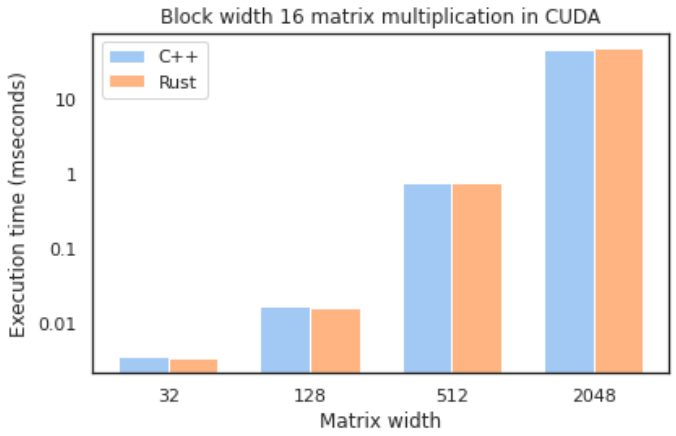


**Figure 4.8:** Difference in execution time for Rust and C++ for untiled matrix multiplication with different matrix widths without bounds checks when accessing global memory. Y-axis in logarithmic scale.

| Matrix Size | Average Speedup |
|---|---|
| 32x32 | 0.96 |
| 128x128 | 1.0 |
| 512x512 | 1.0 |
| 2048x2048 | 1.02 |
| All combined | 0.99 |

**Table 4.9:** Average speedup of C++ over Rust for all experiments within a given matrix size for untiled matrix multiplication with removed bounds checks.

As can be seen in Figure 4.9 and Table 4.10, barely any change regarding the *execution time* can be seen compared to the previous version seen in Figure 4.3 and Table 4.3. One possible reason for this is that since the array copy is a *memory-bound* computational problem, the bottleneck in this case is the memory bandwidth of the GPU. Because of this, the extra computations required during the bounds checks do not have the same effect as on the other *compute-bound* kernels.

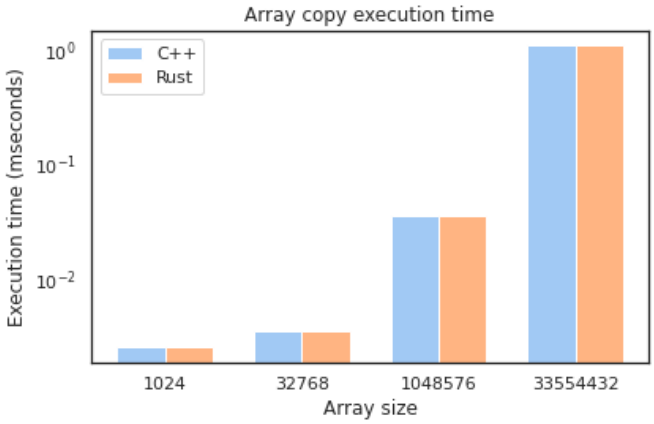


**Figure 4.9:** Difference in execution time for array copy with no bounds checks when accessing global memory. Y-axis in logarithmic scale.

| Array size | Average Speedup |
|---|---|
| 1024 | 1.003 |
| 32768 | 1.003 |
| 1048576 | 1.001 |
| 33554432 | 1.0 |
| All combined | 1.002 |

**Table 4.10:** Average speedup of C++ over Rust for all experiments of the array copy kernel with removed bounds checks.

### 4.2.2 Loop unrolling

Another interesting difference was found by comparing the PTX for the matrix multiplication without tiling was the generated loop unrolling. In Listings 4.1 and 4.2, parts of the PTX can be seen which depict the differences. The biggest difference between these is that the C++ version unrolls four loop iterations, the Rust version on the other hand unrolls all of the loop iterations directly. In the example in Listing 4.2 only a subset of the total unroll was added since it would otherwise be too large to display. This means that the source file for the Rust version is much larger, but also that multiple branching statements per kernel invocation are saved since it does not need to calculate the predicate and branch that the C++ version needs (seen on lines 21 and 22 in Listing 4.1).

```
$L__BB0_4:
  ld.global.f32  %f12, [%rd31];
  ld.global.f32  %f13, [%rd30+-8];
  fma.rn.f32     %f14, %f13, %f12, %f29;
  add.s64        %rd22, %rd31, %rd5;
  ld.global.f32  %f15, [%rd22];
  ld.global.f32  %f16, [%rd30+-4];
  fma.rn.f32     %f17, %f16, %f15, %f14;
  add.s64        %rd23, %rd22, %rd5;
  ld.global.f32  %f18, [%rd23];
  ld.global.f32  %f19, [%rd30];
  fma.rn.f32     %f20, %f19, %f18, %f17;
  add.s64        %rd24, %rd23, %rd5;
  add.s64        %rd31, %rd24, %rd5;
  ld.global.f32  %f21, [%rd24];
  ld.global.f32  %f22, [%rd30+4];
  fma.rn.f32     %f29, %f22, %f21, %f20;
  add.s32        %r29, %r29, 4;
  add.s64        %rd30, %rd30, 16;
  add.s32        %r28, %r28, -4;
  setp.ne.s32    %p6, %r28, 0;
  @%p6 bra       $L__BB0_4;
```

**Listing 4.1:** Loop unrolling C++

```
...
...
...
ld.global.nc.f32  %f78, [%rd4+2944];
ld.global.nc.f32  %f79, [%rd3+92];
fma.rn.f32        %f24, %f79, %f78, %f23;
add.s64           %rd62, %rd2, 24;
setp.ge.u64       %p50, %rd62, %rd8;
@%p50 bra         $L__BB0_64;
add.s64           %rd63, %rd1, 768;
setp.ge.u64       %p51, %rd63, %rd10;
@%p51 bra         $L__BB0_66;

ld.global.nc.f32  %f80, [%rd4+3072];
ld.global.nc.f32  %f81, [%rd3+96];
fma.rn.f32        %f25, %f81, %f80, %f24;
add.s64           %rd64, %rd2, 25;
setp.ge.u64       %p52, %rd64, %rd8;
@%p52 bra         $L__BB0_64;
add.s64           %rd65, %rd1, 800;
setp.ge.u64       %p53, %rd65, %rd10;
@%p53 bra         $L__BB0_66;

ld.global.nc.f32  %f82, [%rd4+3200];
ld.global.nc.f32  %f83, [%rd3+100];
fma.rn.f32        %f26, %f83, %f82, %f25;
add.s64           %rd66, %rd2, 26;
setp.ge.u64       %p54, %rd66, %rd8;
@%p54 bra         $L__BB0_64;
add.s64           %rd67, %rd1, 832;
setp.ge.u64       %p55, %rd67, %rd10;
@%p55 bra         $L__BB0_66;

ld.global.nc.f32  %f84, [%rd4+3328];
ld.global.nc.f32  %f85, [%rd3+104];
fma.rn.f32        %f27, %f85, %f84, %f26;
add.s64           %rd68, %rd2, 27;
setp.ge.u64       %p56, %rd68, %rd8;
@%p56 bra         $L__BB0_64;
add.s64           %rd69, %rd1, 864;
setp.ge.u64       %p57, %rd69, %rd10;
@%p57 bra         $L__BB0_66;
...
...
...
```

**Listing 4.2:** Loop unrolling Rust

### 4.2.3 Type conversion

As Rust is a strongly typed language, certain data types must be used for certain operations. If two dependent operations operate with different data types, a conversion from one data type to another is required. As an example, looking at the code for the copy kernel in Appendix A, Listing A.5, the `.add()` function on the raw pointer `c` requires a `usize` as argument, but the used `Rust-CUDA thread` functions return values with the data type `u32`. This requires the use of the `as usize` operation, which ultimately results in additional instructions to be executed in the kernel. In the PTX code this operation results in the use of the instruction `cvt.u64.u32` which converts from `u32` to `u64` in this case since the program compiles for a 64-bit system. The `cvt.u64.u32` operation is made through a zero extension (zext) oper-

ation [31] which may not be the heaviest operation, but as explained still adds to the total instruction count.

When modifying the array copy kernel by using multiple `as usize` conversions instead of a single one, as seen in Listing A.7, we see an increase in instructions executed. Table 4.11 shows the difference where we see that both the ISETP and IMAD instructions have increased drastically.

| **Instruction** | **Single `as usize`** | **Multiple `as usize`** |
|---|---|---|
| ISETP | 98,304 | 131,072 |
| IMAD | 98,304 | 131,072 |
| ULDC | 65,536 | 65,536 |
| S2R | 65,536 | 65,536 |
| EXIT | 65,536 | 65,536 |
| STG | 32,768 | 32,768 |
| SHF | 32,768 | 32,768 |
| LDG | 32,768 | 32,768 |
| BRA | 32,768 | 32,768 |

**Table 4.11:** Difference of instruction count for the array copy kernel when using a single and multiple as usize conversions.

# 5

# Discussion

As we could see in the previous chapter, translating line-by-line from a C++ CUDA kernel to Rust does not bring the performance to a good enough level to compete with C++. The biggest culprit was the bounds checks that standard Rust indexing adds when fetching from global memory. The feature itself is beneficial as it ensures that data cannot be fetched from memory addresses that are outside the memory scope of the program. Hence, it is a feature that ensures that the program works as intended regarding the memory it is supposed to use, which goes hand in hand with the memory safety that the Rust language wants to achieve. When it comes to HPC, this is a luxury that is too costly to be used in production, as we have seen during the different experiments. To be able to have high performance while still keeping the benefits of the bounds check, different performance optimizations might be a good option, where the default during development includes the bounds checks, meaning notice early if there is an illegal memory access. When code should be transitioned into production, the option to change to a higher level of performance optimization which removes these checks would be a nice-to-have feature. However, it is highly unlikely that this will become a feature in the Rust language since it goes against one of the core principles of the language, memory safety. This makes it vital for programmers who want to create high-performing GPGPU programs in Rust to know that these bounds checks exist, and how to avoid them. While seeming like an easy thing, it might become a problem in larger codebases where multiple contributors work, since it requires proper on-boarding of new employees and regular screening of the code to avoid mistakes like these to appear.

This is also an indication that writing high-performance Rust GPGPU kernels without them being marked as `unsafe` will never be feasible. The reasoning for this is that as we saw in Section 4.1, we don't achieve good enough performance while having bounds checks, but to remove them we have to use a function that requires `unsafe` Rust. Since it is highly unlikely that the Rust language will remove these bounds checks in any way, the kernels will have to keep using `unsafe`. This does not mean in any way that `unsafe` Rust is more unsafe than regular C++ code, it only means that the goal of memory safety won't be achievable for these kernels.

Since `Rust-CUDA` has a requirement of using the *nightly-2021-12-04* it means that it is stuck on a daily version that quickly will become out of date. This means that different security patches and bug fixes might be missed, which is not optimal. This requirement only exists for the codegen, but it might be hard for developers

to enforce this specific version on only one part of the program, causing the entire program to run on the version. Hopefully, this requirement will be dropped in the future as the maturity of `Rust-CUDA` increases.

Something interesting with `Rust-CUDA` is that it operates on generated PTX, meaning that it is possible to use existing C++ CUDA kernels compiled down to PTX together with Rust *host* code. Rewriting existing codebases is therefore possible by a step-by-step approach, reusing the existing *device* code while converting the *host* code to Rust. This is important since it lowers the entry barriers for companies to start using `Rust-CUDA` which will benefit the community and the development of the library.

The results shown in Chapter 4 are all captured on machine 1 (see Table 3.1), however, the same experiments were conducted on the more high-end machine 2 (see Table 3.2) in order to validate the results and find potential differences. It was clear from the results that the kernels had the same relative performance on the two machines, both in terms of *execution time* and *energy consumption*. This is not surprising as the machines used the same drivers on graphics cards from the same generation. Using a more modern or outdated graphics card could potentially lead to different results due to potentially different interpretations of PTX code which could potentially indicate that the Rust implementations using `Rust-CUDA` are more suitable for modern GPUs. The results captured on machine 2 can be found in Appendix B.2.

As with most research, some limitations that influence the quality of the results are present in this work. The `Rust-CUDA` that all the Rust programs are implemented with is, although actively in development and being one of the more prominent GPU programming frameworks for Rust, is still in its early days and has not yet matured the way GPU programming in C++/native CUDA has. There might be missing or erroneous optimizations that negatively impact the performance of applications, which can potentially be solved in future updates.

The kernels compared in this work were carefully implemented to be comparable, but due to the nature of dealing with two different programming languages, this is not a trivial task. There are differences in the kernels imposed by syntax and compiler constraints that ultimately lead to different PTX code and performance. This is not negative per se, as the performance differences are what is being researched. However, it can't be ruled out with 100% certainty that differences exists because of programmer error.

As each experiment is non-deterministic in nature, the results vary from execution to execution. Deviations can occur due to other processes running on the machine influencing the performance. By running the experiments in isolation and by averaging the results of each experiment over twenty runs, these effects are mitigated. The number twenty was chosen as it was found that the outliers in the results were diminished effectively while still keeping a reasonable time to run the experiments.

The ethical considerations for this thesis are minimal. As the work will be based on

data fabricated and collected by the authors, no breach of privacy or other sensitive information can be realized. Moreover, the practice of high-performance computing consumes large amounts of energy with the powerful hardware utilized. Although this project aims to improve energy efficiency of GPU computations, energy consumption for calculations is something to consider in general from an environmental point of view.

# 6

# Conclusion

Even though GPU programming in Rust is relatively new, this thesis has shown that it is highly possible to create high-performing CUDA kernels written in Rust. Benchmarking these kernels and comparing them to C++ CUDA kernels it was found that when having line-by-line comparable code, Rust had much lower performance, mainly because of automatically added bounds checks when accessing global memory. By removing these, the Rust kernels started to perform at the same level or sometimes even better than the C++ counterpart.

By tying back to the research questions of the thesis, we have found multiple capabilities and limitations in GPU programming using Rust that bring differences regarding performance. The first and most impactful was the added bounds checks to the generated PTX when using regular indexing operations. These had a large negative impact on performance. By removing these we could achieve much better performance. Rust also requires the usage of `usize` as parameters in some function calls, while the return type of some `Rust-CUDA` functions return `u32`. Having multiple conversions between these types increases the number of instructions run. Finally, we found that Rust performs more loop unrolling than the C++ counterpart, reducing the number of instructions executed which in turn increases the performance.

We see that the relative performance of `Rust-CUDA` programs compared to programs made with C++-CUDA is quite competitive. As long as the programmer is aware of the bounds checks and the steps to avoid them, the created programs will according to our results perform at least at the same level as the C++ counterpart, but in a few cases even better.

When it comes to the different available libraries for executing Rust GPU code, we have found that there exists a few different projects available that work in different ways. `Rust-CUDA` which was the focus of this thesis, executes CUDA kernels without acting as a wrapper. `arrayfire-rust` and `rust-gpu` are other prominent libraries available. Kernels written in `rust-gpu` compile down to SPIR-V and has a higher focus on graphics programming in Rust and not on GPGPU. `arrayfire-rust` on the other hand is a wrapper for ArrayFire, which enables CUDA, OpenCL, and CPU HPC programming in Rust. No performance profiling was made for either of these two libraries and could be a potential future work to compare with the performance achieved with `Rust-CUDA` .

Having worked with `Rust-CUDA` we feel that it is a great candidate for CUDA programming and we are excited to see Rust grow in high-performance computing in the future.

# Bibliography


[1] A. Bychkov and V. Nikolskiy, "Rust Language for Supercomputing Applications," *Communications in Computer and Information Science*, vol. 1510 CCIS, pp. 391–403, 2021. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-92864-3_30

[2] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis, "GPU Programming in Rust: Implementing High-Level Abstractions in a Systems-Level Language," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 5 2013, pp. 315–324.

[3] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures," *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2 2014.

[4] "Programming Guide :: CUDA Toolkit Documentation." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model

[5] "NVCC :: CUDA Toolkit Documentation." [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#cuda-programming-model

[6] "PTX ISA Version 7.6 Introduction." [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#introduction

[7] "History of C++ - C++ Information." [Online]. Available: https://www.cplusplus.com/info/history/

[8] "A Brief Description - C++ Information." [Online]. Available: https://www.cplusplus.com/info/description/

[9] "Mozilla Welcomes the Rust Foundation." [Online]. Available: https://blog.mozilla.org/en/mozilla/mozilla-welcomes-the-rust-foundation/

[10] "Rust Programming Language." [Online]. Available: https://www.rust-lang.org/

[11] “Unsafe Rust - The Rust Programming Language.” [Online]. Available: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

[12] “Channels - The rustup book.” [Online]. Available: https://rust-lang.github.io/rustup/concepts/channels.html

[13] “ rust-gpu GtitHub.” [Online]. Available: https://github.com/EmbarkStudios/rust-gpu

[14] “arrayfire-rust GitHub.” [Online]. Available: https://github.com/arrayfire/arrayfire-rust

[15] “SPIR-V - OpenGL Wiki.” [Online]. Available: https://www.khronos.org/opengl/wiki/SPIR-V

[16] “ArrayFire Documentation.” [Online]. Available: https://arrayfire.org/docs/installing.htm

[17] “Rust-GPU/Rust-CUDA: Ecosystem of libraries and tools for writing and executing fast GPU code fully in Rust.” [Online]. Available: https://github.com/Rust-GPU/Rust-CUDA

[18] “Rust-CUDA/faq.md at master · Rust-GPU/Rust-CUDA.” [Online]. Available: https://github.com/Rust-GPU/Rust-CUDA/blob/master/guide/src/faq.md

[19] “NVVM IR :: CUDA Toolkit Documentation.” [Online]. Available: https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html

[20] “Rust-CUDA/features.md at master · Rust-GPU/Rust-CUDA.” [Online]. Available: https://github.com/Rust-GPU/Rust-CUDA/blob/master/guide/src/features.md

[21] “Rust-CUDA/getting_started.md at master · Rust-GPU/Rust-CUDA.” [Online]. Available: https://github.com/Rust-GPU/Rust-CUDA/blob/master/guide/src/guide/getting_started.md

[22] “Rust-CUDA/gpu_computing.md at master · Rust-GPU/Rust-CUDA.” [Online]. Available: https://github.com/Rust-GPU/Rust-CUDA/blob/master/guide/src/cuda/gpu_computing.md

[23] “RAII - Rust By Example.” [Online]. Available: https://doc.rust-lang.org/rust-by-example/scope/raii.html

[24] “std::result - Rust.” [Online]. Available: https://doc.rust-lang.org/std/result/

[25] “NVIDIA Nsight Compute | NVIDIA Developer.” [Online]. Available: https://developer.nvidia.com/nsight-compute

[26] “NVIDIA Visual Profiler | NVIDIA Developer.” [Online]. Available: https://developer.nvidia.com/nvidia-visual-profiler

[27] “Developer Blog: How to Migrate to the Latest Versions of the Nsight Suite | NVIDIA Technical Blog.” [Online]. Available: https://developer.nvidia.com/blog/migrating-to-the-latest-versions-of-the-nsight-suite/

[28] “NVCC :: CUDA Toolkit Documentation.” [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#ptxas-options-opt-level

[29] “Customizing Builds with Release Profiles - The Rust Programming Language.” [Online]. Available: https://doc.rust-lang.org/book/ch14-01-release-profiles.html

[30] “Nsight Compute CLI :: Nsight Compute Documentation.” [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html#command-line-options-profile

[31] “PTX ISA :: CUDA Toolkit Documentation.” [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#type-conversion

# A

# Appendix A

In this appendix the kernel functions that was used during the benchmarking can be found.

```
#[kernel]
#[allow(improper_ctypes_definitions, clippy::missing_safety_doc)]
pub unsafe fn matMulCUDA(a: &[f32], b: &[f32], c: *mut f32) {
    let a_shared_pointer = shared_array![f32; BLOCK_WIDTH*BLOCK_WIDTH];
    let b_shared_pointer = shared_array![f32; BLOCK_WIDTH*BLOCK_WIDTH];

    let thread_idx = thread::thread_idx();
    let block_idx = thread::block_idx();

    let row_idx = block_idx.y * BLOCK_WIDTH_U32 + thread_idx.y;
    let col_idx = block_idx.x * BLOCK_WIDTH_U32 + thread_idx.x;


    let mut computed_value: f32 = 0.0;

    for m in 0..(MATRIX_WIDTH_U32 +BLOCK_WIDTH_U32-1)/BLOCK_WIDTH_U32{
        //copy to shared memory
        if m * BLOCK_WIDTH_U32 + thread_idx.x < MATRIX_WIDTH_U32 && row_idx < MATRIX_WIDTH_U32{
            let mut a_write = &mut *a_shared_pointer.add((thread_idx.y* BLOCK_WIDTH_U32 + thread_idx.x) as usize);
            //*a_write = *a.get_unchecked(( row_idx*MATRIX_WIDTH_U32 + m* BLOCK_WIDTH_U32+thread_idx.x ) as usize);
            *a_write = a[( row_idx*MATRIX_WIDTH_U32 + m*BLOCK_WIDTH_U32+thread_idx.x ) as usize];
        }
        else{
            let mut a_write = &mut *a_shared_pointer.add((thread_idx.y* BLOCK_WIDTH_U32 + thread_idx.x) as usize);
            *a_write = 0.0;
        }

        if m*BLOCK_WIDTH_U32 + thread_idx.y < MATRIX_WIDTH_U32 && col_idx < MATRIX_WIDTH_U32 {
            let mut b_write = &mut *b_shared_pointer.add((thread_idx.y* BLOCK_WIDTH_U32 + thread_idx.x) as usize);
            //*b_write = *b.get_unchecked(((m*BLOCK_WIDTH_U32 + thread_idx.y)* MATRIX_WIDTH_U32+col_idx) as usize);
            *b_write = b[((m*BLOCK_WIDTH_U32 + thread_idx.y)*MATRIX_WIDTH_U32+ col_idx) as usize];
        }
        else{
            let mut b_write = &mut *b_shared_pointer.add((thread_idx.y* BLOCK_WIDTH_U32 + thread_idx.x) as usize);
            *b_write = 0.0;
        }

        //sync to make sure all data is available in shared memory before computations
        thread::sync_threads();
```

```
40
41          for k in 0..(BLOCK_WIDTH_U32){
42              let mut a_read = &mut *a_shared_pointer.add((thread_idx.y*
    BLOCK_WIDTH_U32 + k) as usize);
43              let mut b_read = &mut *b_shared_pointer.add((k*BLOCK_WIDTH_U32 +
    thread_idx.x) as usize);
44              computed_value += *a_read * *b_read;
45          }
46
47          //sync to ensure all threads finished using shared memory before we move
48          thread::sync_threads();
49      }
50
51      if row_idx < MATRIX_WIDTH_U32 && col_idx < MATRIX_WIDTH_U32{
52          let elem = &mut *c.add((row_idx * MATRIX_WIDTH_U32 + col_idx) as usize);
53          *elem = computed_value;
54      }
55  }
```

**Listing A.1:** Rust Tiled Matrix Multiplication CUDA Kernel

```
1  __global__ static void matMultCUDA(const float* a, const float* b, float* c, int n)
       {
2      __shared__ float a_shared[BLOCK_WIDTH][BLOCK_WIDTH];
3      __shared__ float b_shared[BLOCK_WIDTH][BLOCK_WIDTH];
4
5      int b_x = blockIdx.x;
6      int b_y = blockIdx.y;
7      int t_x = threadIdx.x;
8      int t_y = threadIdx.y;
9
10     int Row = b_y * BLOCK_WIDTH + t_y;
11     int Col = b_x * BLOCK_WIDTH + t_x;
12
13     float computed_value = 0;
14     for (int m = 0; m < (n+BLOCK_WIDTH-1)/BLOCK_WIDTH; m++){
15
16         //copy to shared memory
17         if(m*BLOCK_WIDTH + t_x < n && Row < n)
18             a_shared[t_y][t_x] = a[Row * n + m*BLOCK_WIDTH + t_x];
19         else
20             a_shared[t_y][t_x] = 0;
21
22         if(m*BLOCK_WIDTH + t_y < n && Col < n)
23             b_shared[t_y][t_x] = b[(m*BLOCK_WIDTH + t_y) * n + Col];
24         else
25            b_shared[t_y][t_x] = 0;
26         __syncthreads(); //sync to make sure all data is available in shared memory
        before computations
27         for (int k = 0; k < BLOCK_WIDTH; ++k){
28             computed_value += a_shared[t_y][k] * b_shared[k][t_x];
29         }
30         __syncthreads(); //sync to ensure all threads finished using shared memory
       before we move
31     }
32     if(Row < n && Col < n)
33         c[Row * n + Col] = computed_value;
34 }
```

**Listing A.2:** C++ Tiled Matrix Multiplication CUDA Kernel

```
1  #[kernel]
2  #[allow(improper_ctypes_definitions, clippy::missing_safety_doc)]
3  pub unsafe fn matMulCUDA(a: &[f32], b: &[f32], c: *mut f32) {
4      let thread_idx = thread::thread_idx();
5      let block_idx = thread::block_idx();
6
7      let row = (block_idx.y * BLOCK_WIDTH + thread_idx.y) as usize;
8      let column = (block_idx.x * BLOCK_WIDTH + thread_idx.x) as usize;
```

```
9
10      if row < MATRIX_WIDTH && column < MATRIX_WIDTH {
11          let mut p_value: f32 = 0.0;
12          for k in 0..MATRIX_WIDTH {
13              //p_value += a.get_unchecked(row*MATRIX_WIDTH+k)*b.get_unchecked(k*
    MATRIX_WIDTH+column);
14              p_value += a[row*MATRIX_WIDTH+k]*b[k*MATRIX_WIDTH+column];
15          }
16
17          let write = &mut *c.add(row*MATRIX_WIDTH+column);
18          *write = p_value;
19      }
20  }
```

**Listing A.3:** Rust Untiled Matrix Multiplication CUDA Kernel

```
1   __global__ static void matMultCUDA(const float* a, const float* b, float* c, int n)
        {
2       int Row = blockIdx.y*blockDim.y+threadIdx.y;
3       int Col = blockIdx.x*blockDim.x+threadIdx.x;
4
5       if ((Row < n) && Col < n){
6           float PValue = 0;
7           for (int k = 0; k < n; k++){
8               PValue += a[Row*n+k]*b[k*n+Col];
9           }
10          c[Row*n+Col] = PValue;
11      }
12  }
```

**Listing A.4:** C++ Untiled Matrix Multiplication CUDA Kernel

```
1   #[kernel]
2   #[allow(improper_ctypes_definitions, clippy::missing_safety_doc)]
3   pub unsafe fn copy(a: &[f32], c: *mut f32) {
4       let thread_idx = thread::index() as usize;
5       if thread_idx < a.len() {
6           let elem = &mut *c.add(thread_idx);
7           //*elem = *a.get_unchecked(thread_idx);
8           *elem = a[thread_idx];
9       }
10  }
```

**Listing A.5:** Rust Array Copy CUDA Kernel

```
1   __global__ void copy_array(float *a, float *c)
2   {
3       int thread_idx = blockDim.x * blockIdx.x + threadIdx.x;
4       if (thread_idx < ARRAY_SIZE){
5           c[thread_idx] = a[thread_idx];
6       }
7   }
```

**Listing A.6:** C++ Array Copy CUDA Kernel

```
1   // Code differences for the two version with different amount of usize.
2
3   // With 1 as usize
4   #[kernel]
5   #[allow(improper_ctypes_definitions, clippy::missing_safety_doc)]
6   pub unsafe fn copy(a: &[f32], c: *mut f32) {
7       let thread_idx = (thread::block_dim().x * thread::block_idx().x + thread::
    thread_idx().x) as usize;
8       if thread_idx < ARRAY_SIZE {
9           let elem = &mut *c.add(thread_idx);
10          *elem = a[thread_idx];
11      }
12  }
```

```
13
14 // With multiple as usize
15 #[kernel]
16 #[allow(improper_ctypes_definitions, clippy::missing_safety_doc)]
17 pub unsafe fn copy(a: &[f32], c: *mut f32) {
18     let grid_x = thread::block_dim().x as usize;
19     let block_x = thread::block_idx().x as usize;
20     let thread_x = thread::thread_idx().x as usize;
21     let thread_idx: usize = grid_x * block_x + thread_x;
22     if thread_idx < ARRAY_SIZE {
23         let elem = &mut *c.add(thread_idx);
24         *elem = a[thread_idx];
25     }
26 }
```

**Listing A.7:** Rust Array Copy CUDA Kernel differences with different amounts of usize calls.

# B

# Appendix B

In this appendix graphs for the benchmarked power usage, energy consumption and execution time can be found for both of the machines used.

## B.1 Machine 1

In Figures B.1, B.2 and B.3 graphs depicting the average, minimum and maximum power usage for tiled matrix multiplication can be found. In Figures B.4, B.5 and B.6 graphs depicting the average, minimum and maximum power usage for array copy can be found.

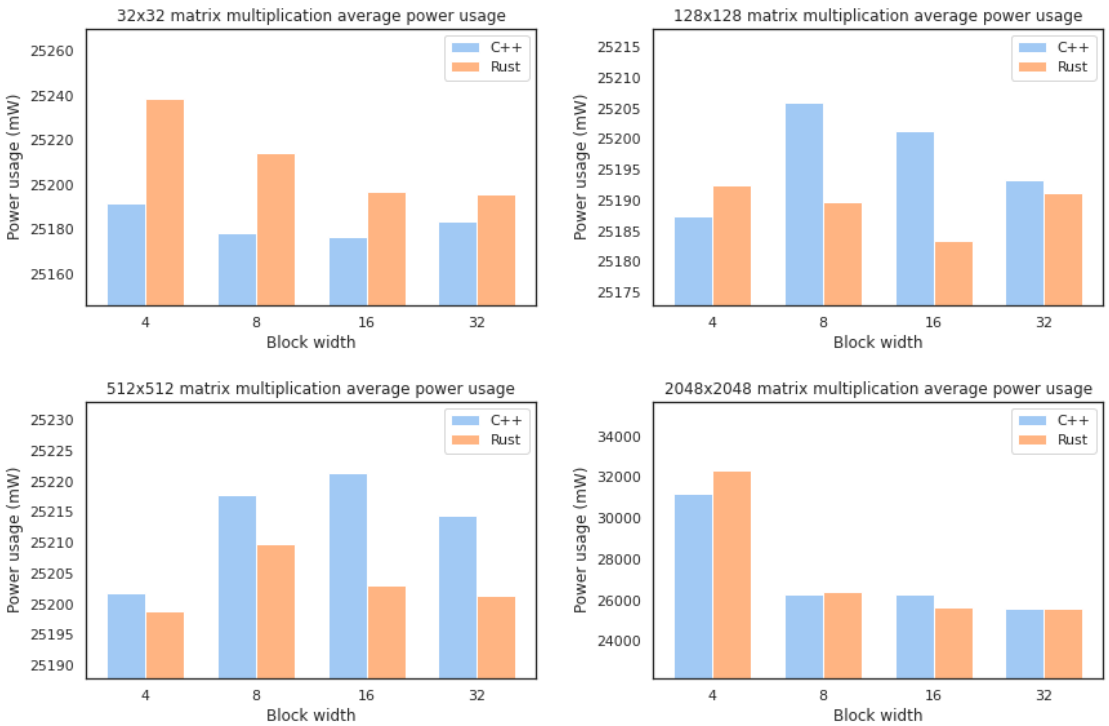


**Figure B.1:** Difference in average power usage for Rust and C++ for tiled matrix multiplication with different block widths.

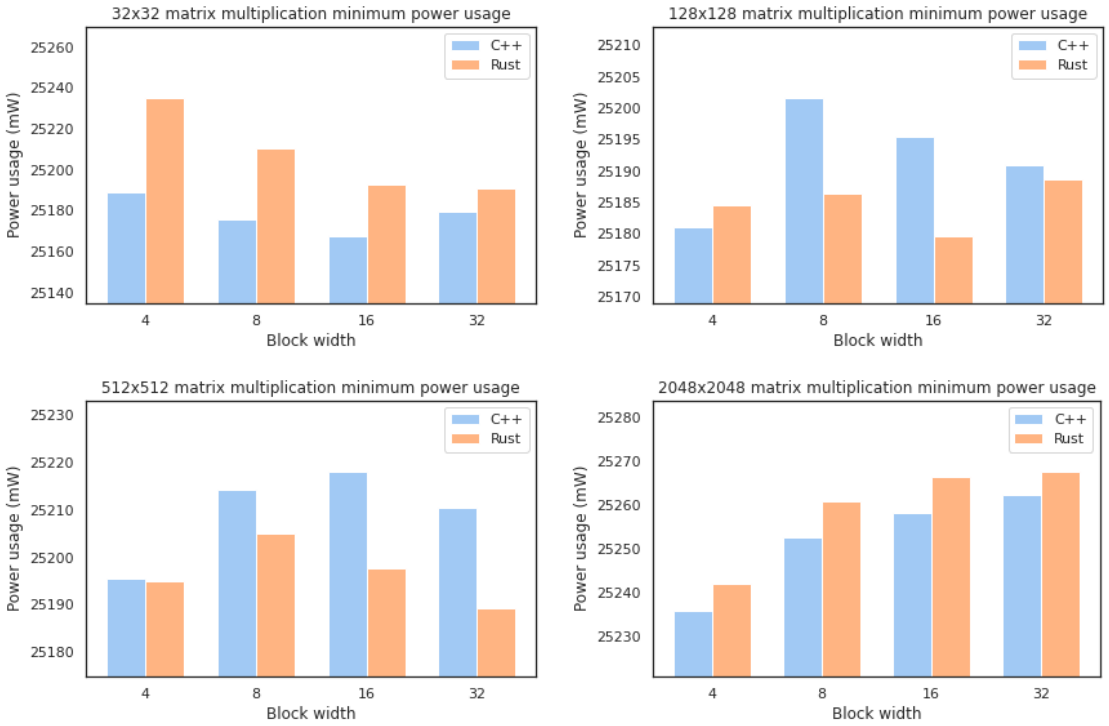


**Figure B.2:** Difference in minimum power usage for Rust and C++ for tiled matrix multiplication with different block widths.

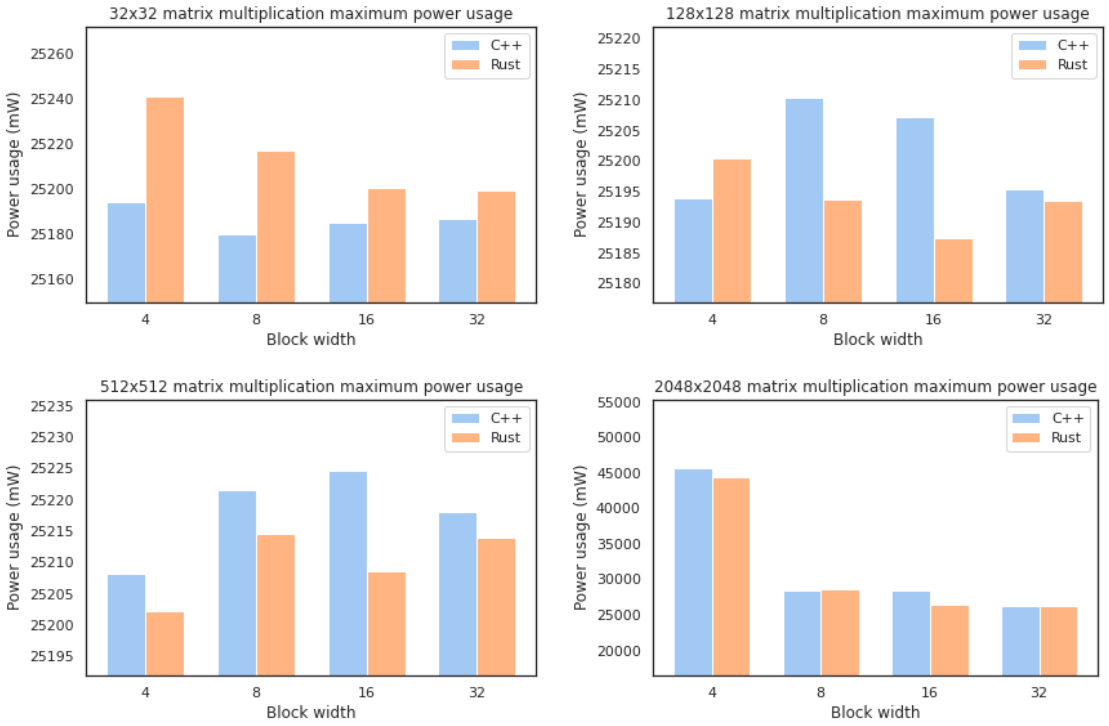


**Figure B.3:** Difference in maximum power usage for Rust and C++ for tiled matrix multiplication with different block widths.

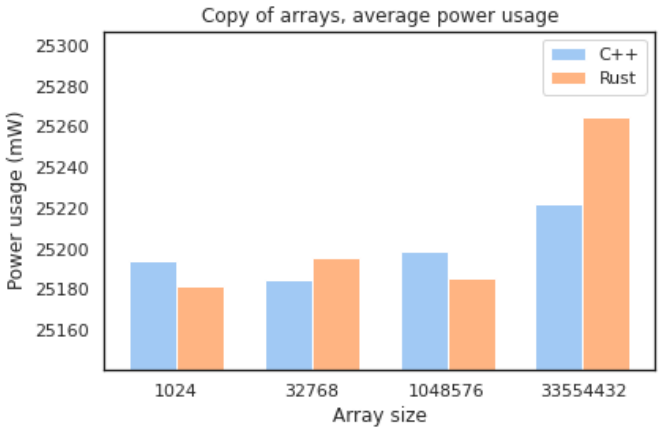


**Figure B.4:** Difference in average power usage for Rust and C++ for array copy with different array sizes.

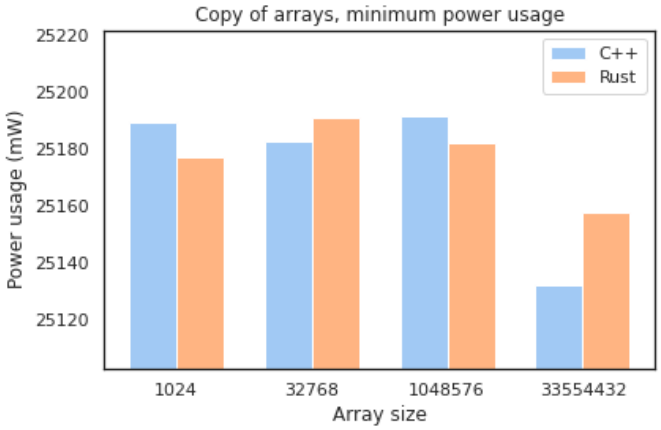


**Figure B.5:** Difference in minimum power usage for Rust and C++ for array copy with different array sizes.

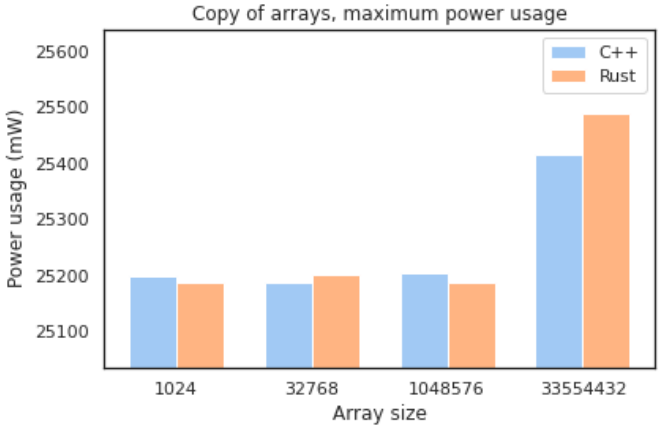


**Figure B.6:** Difference in maximum power usage for Rust and C++ for array copy with different array sizes.

### B.1.1 Energy consumption without bounds checks

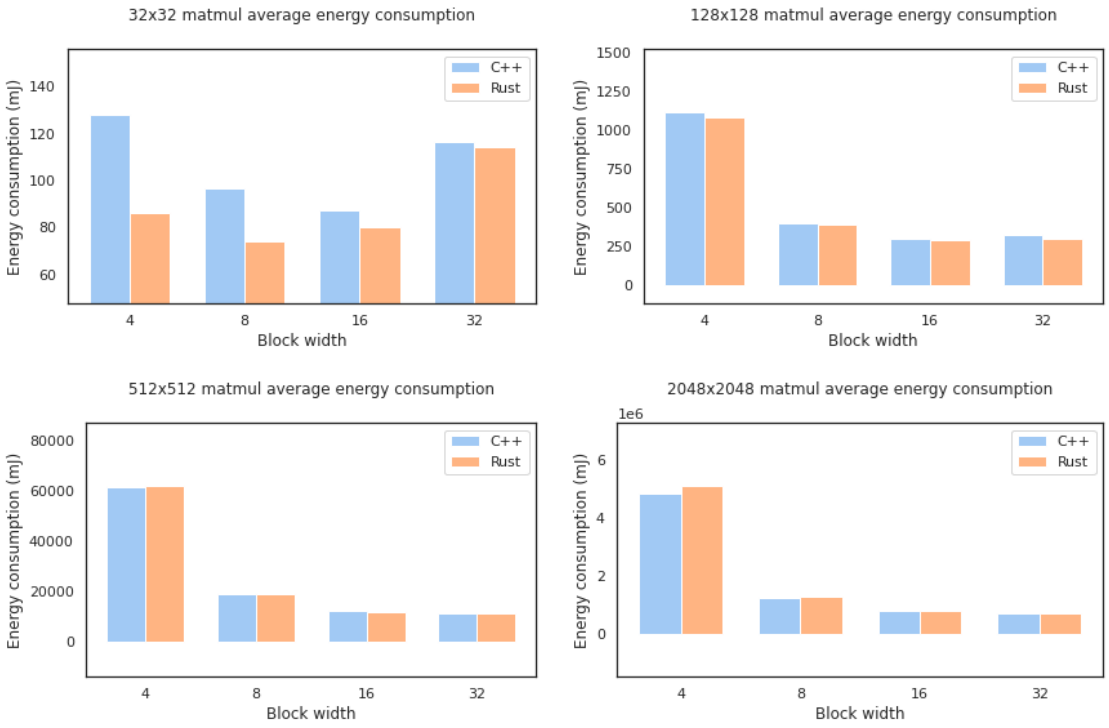


**Figure B.7:** Tiled matrix multiplication energy consumption difference between C++ and Rust without bounds checks.

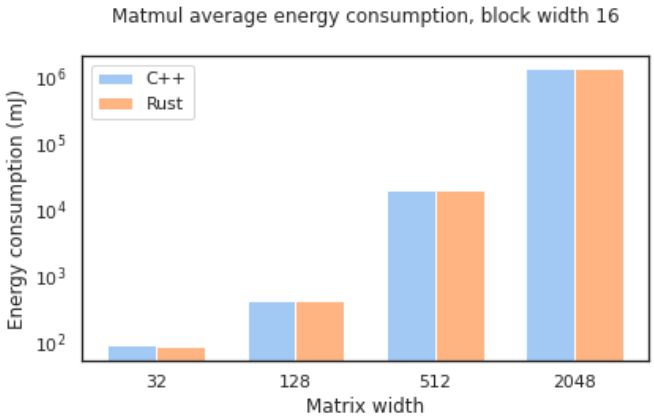


**Figure B.8:** Untiled matrix multiplication energy consumption difference between C++ and Rust without bounds checks. Y-axis in logarithmic scale

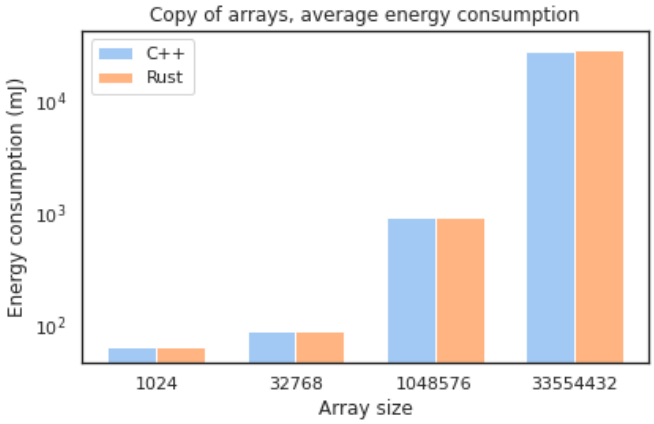


**Figure B.9:** Copy kernel energy consumption difference between C++ and Rust without bounds checks.

## B.2 Machine 2

All the captured data from benchmarks ran on machine 2.

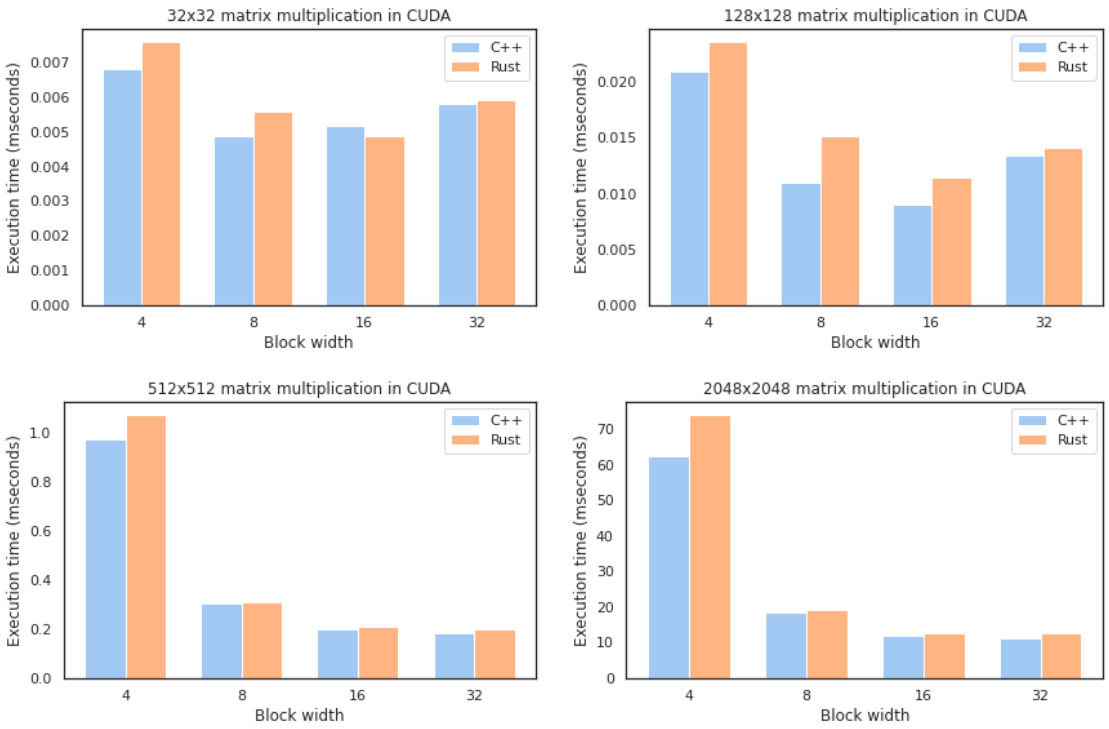


**Figure B.10:** Difference in execution time for Rust and C++ for tiled matrix multiplication with different matrix and block widths, Machine 2.

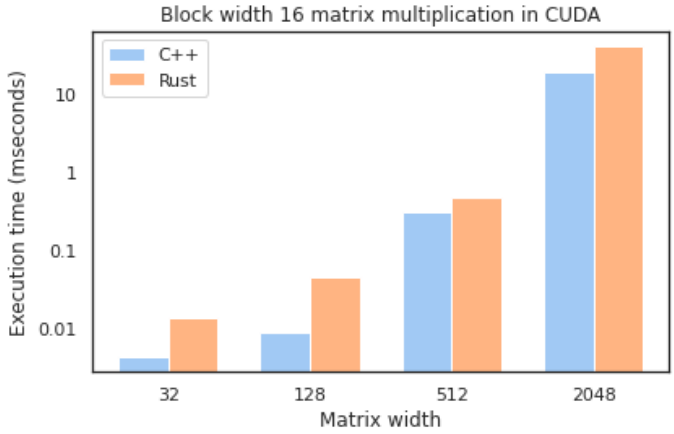


**Figure B.11:** Difference in execution time for untiled matmul, Machine 2. Y-axis in logarithmic scale

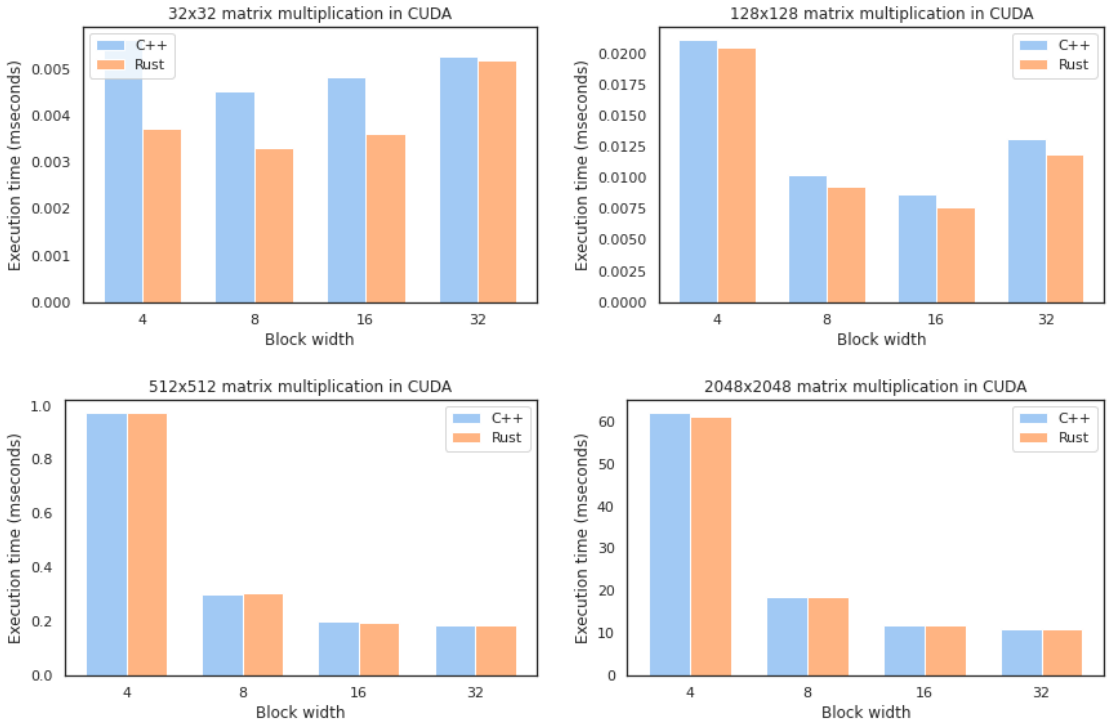


**Figure B.12:** Difference in execution time for Rust and C++ for tiled matrix multiplication without bounds checks with different matrix and block widths, Machine 2.

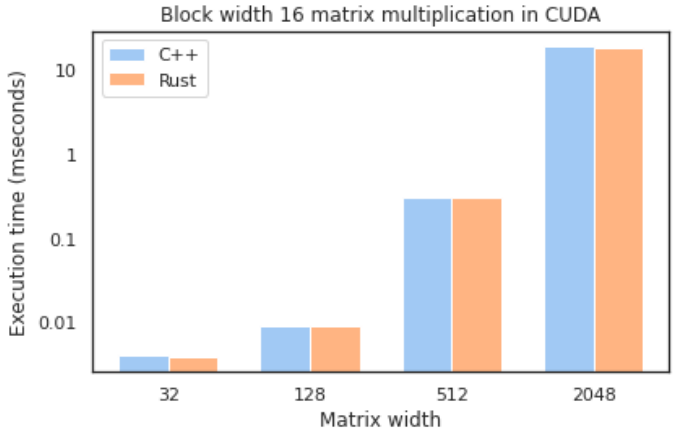


**Figure B.13:** Difference in execution time for Rust and C++ for untiled matrix multiplication without bounds checks with different matrix widths, Machine 2. Y-axis in logarithmic scale.

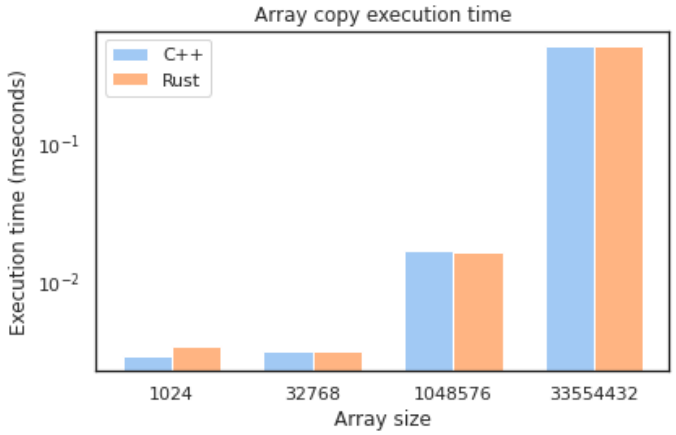


**Figure B.14:** Difference in execution time for array copy without bounds checks, Machine 2.

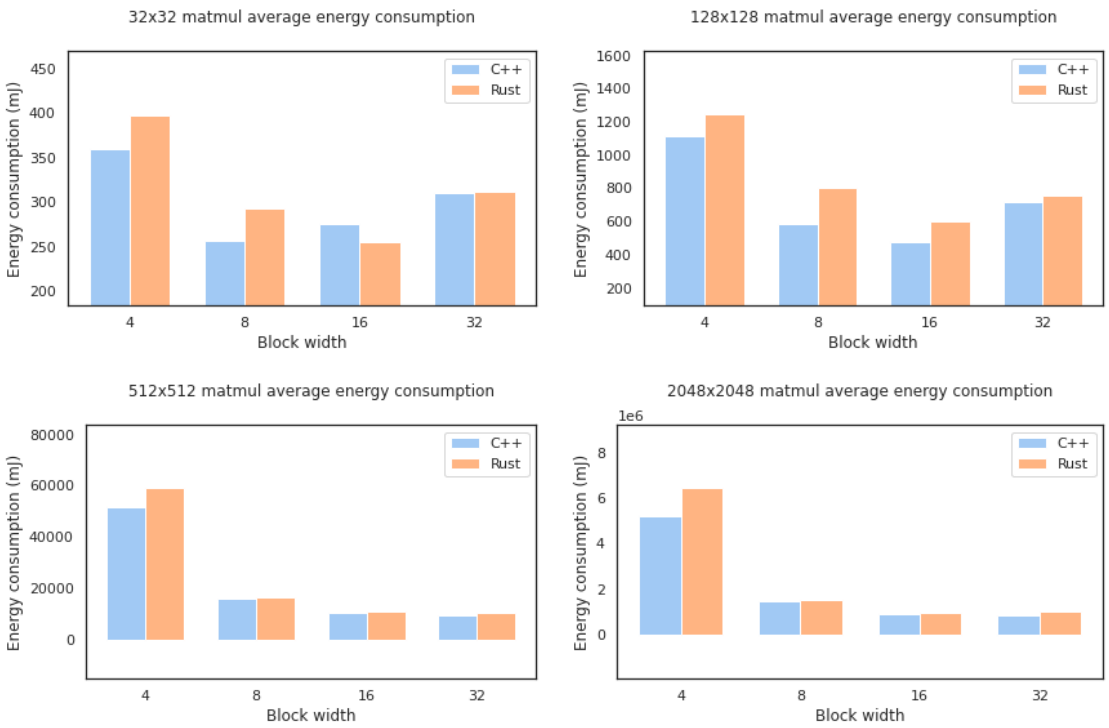


**Figure B.15:** Difference in average energy consumption for Rust and C++ for tiled matrix multiplication with different block widths, Machine 2.

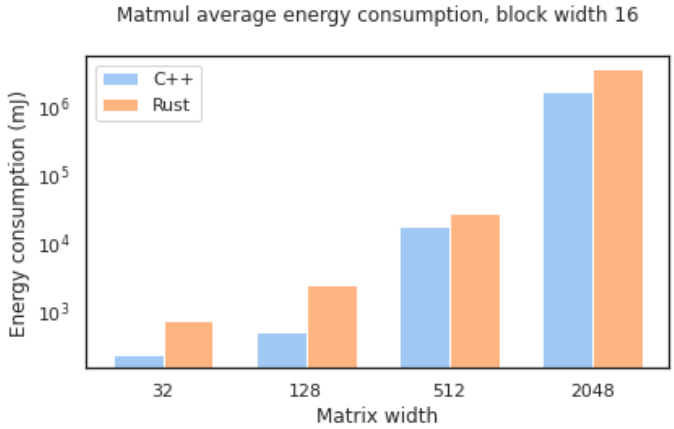


**Figure B.16:** Difference in average energy consumption for Rust and C++ for untiled matrix multiplication with different matrix sizes, Machine 2. Y-axis in logarithmic scale.

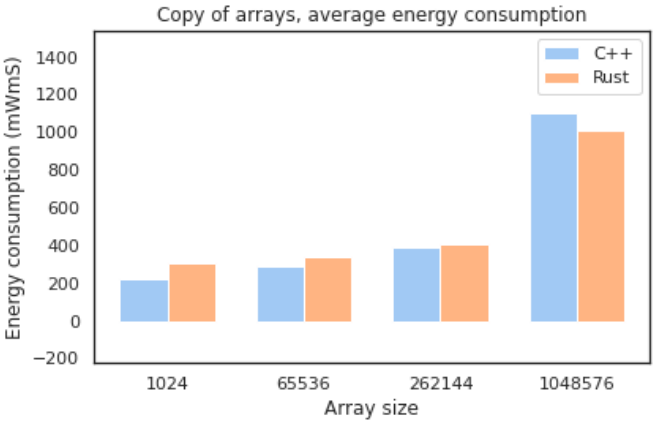


**Figure B.17:** Difference in average energy consumption for array copy, Machine 2.

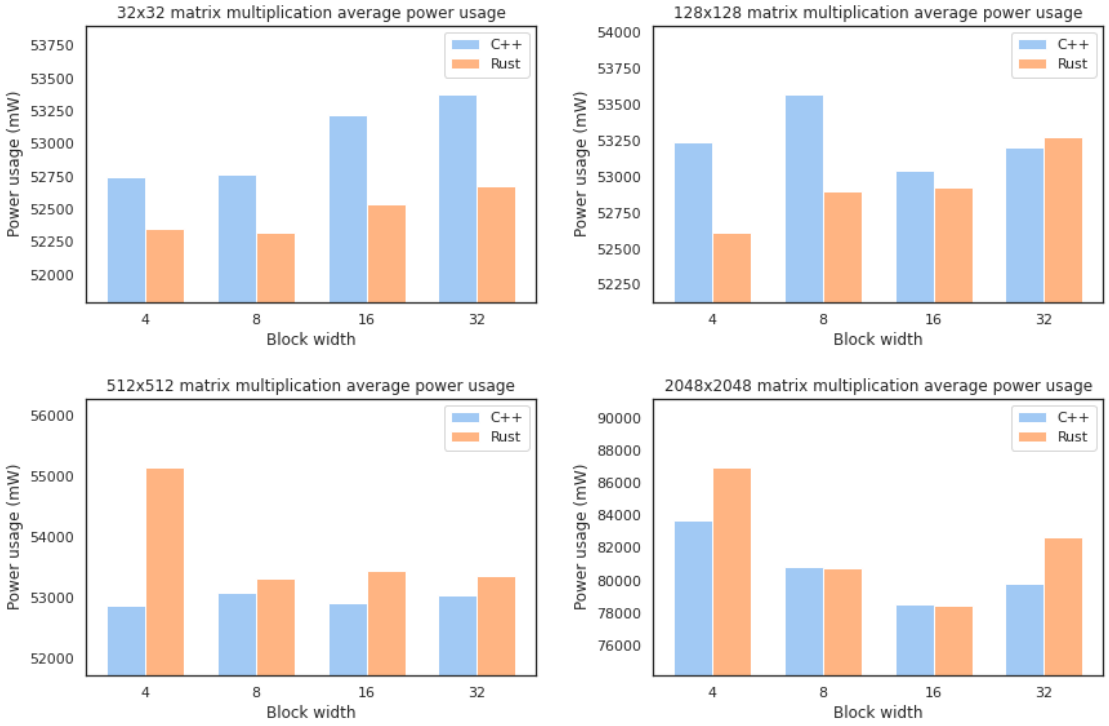


**Figure B.18:** Difference in average power usage for Rust and C++ for tiled matrix multiplication with different block widths, Machine 2.

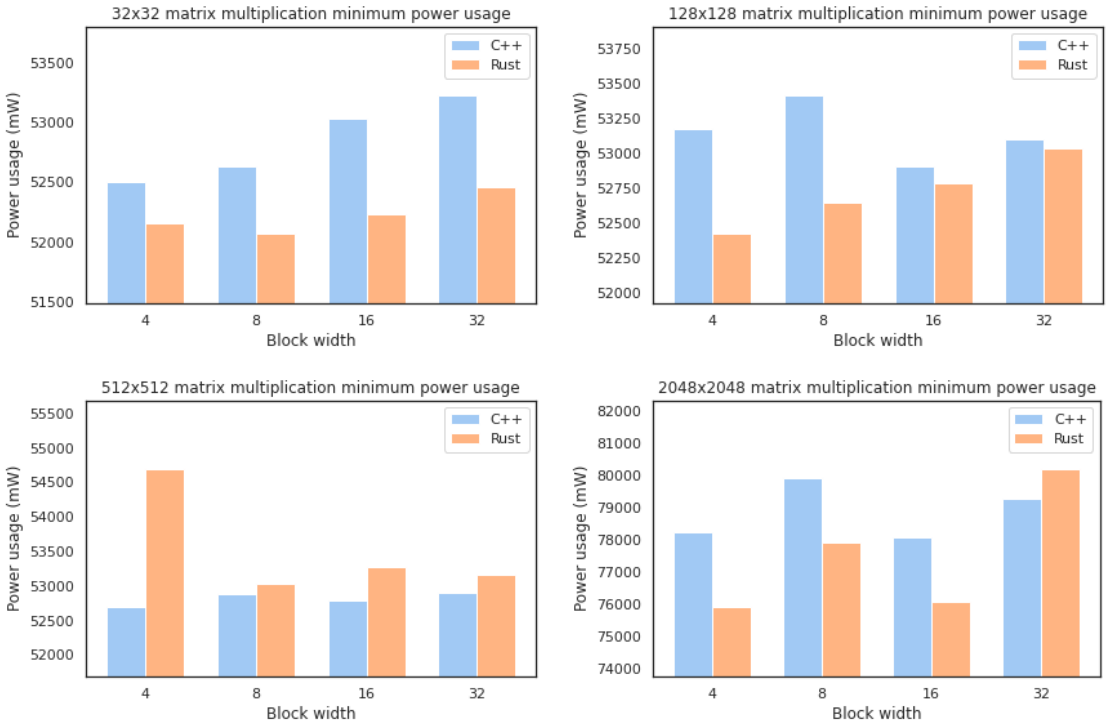


**Figure B.19:** Difference in minimum power usage for Rust and C++ for tiled matrix multiplication with different block widths, Machine 2.

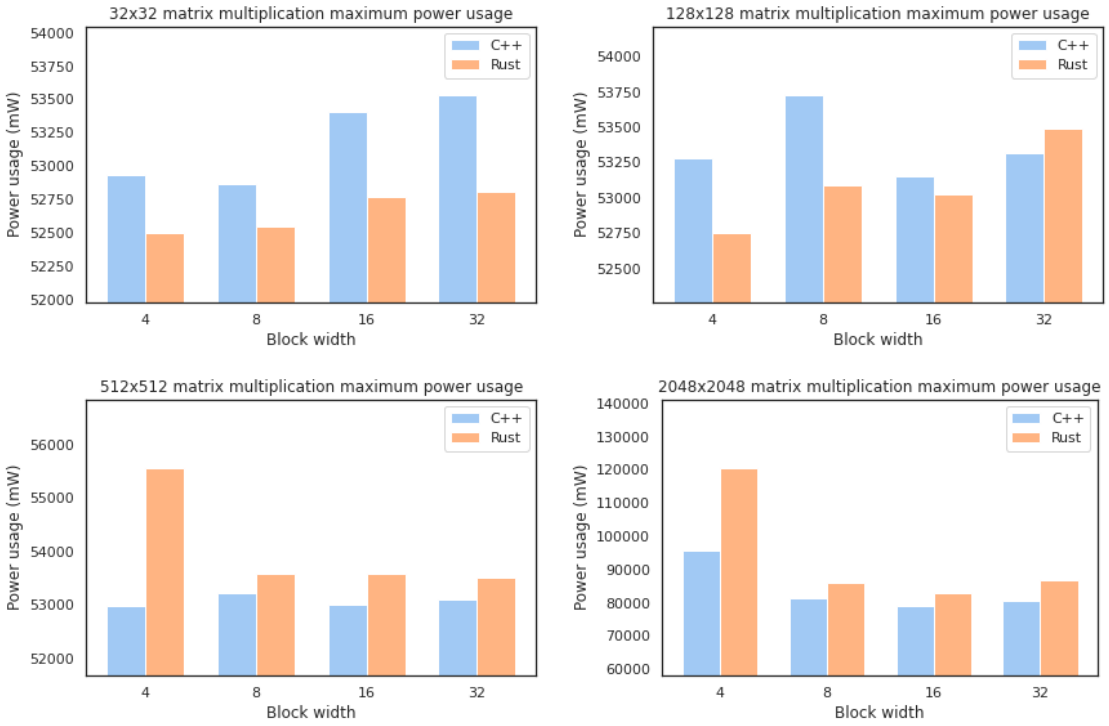


**Figure B.20:** Difference in maximum power usage for Rust and C++ for tiled matrix multiplication with different block widths, Machine 2.

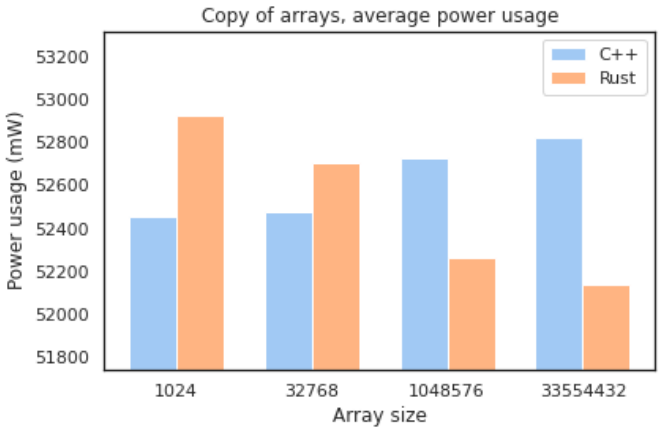


**Figure B.21:** Difference in average power usage for Rust and C++ for array copy with different array sizes, Machine 2.

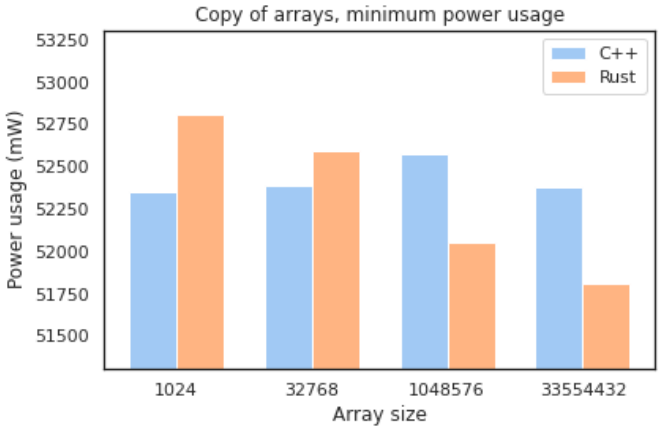


**Figure B.22:** Difference in minimum power usage for Rust and C++ for array copy with different array sizes, Machine 2.

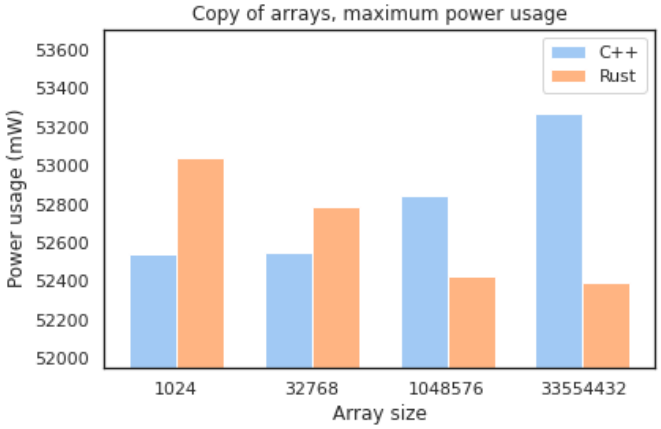


**Figure B.23:** Difference in maximum power usage for Rust and C++ for array copy with different array sizes, Machine 2.

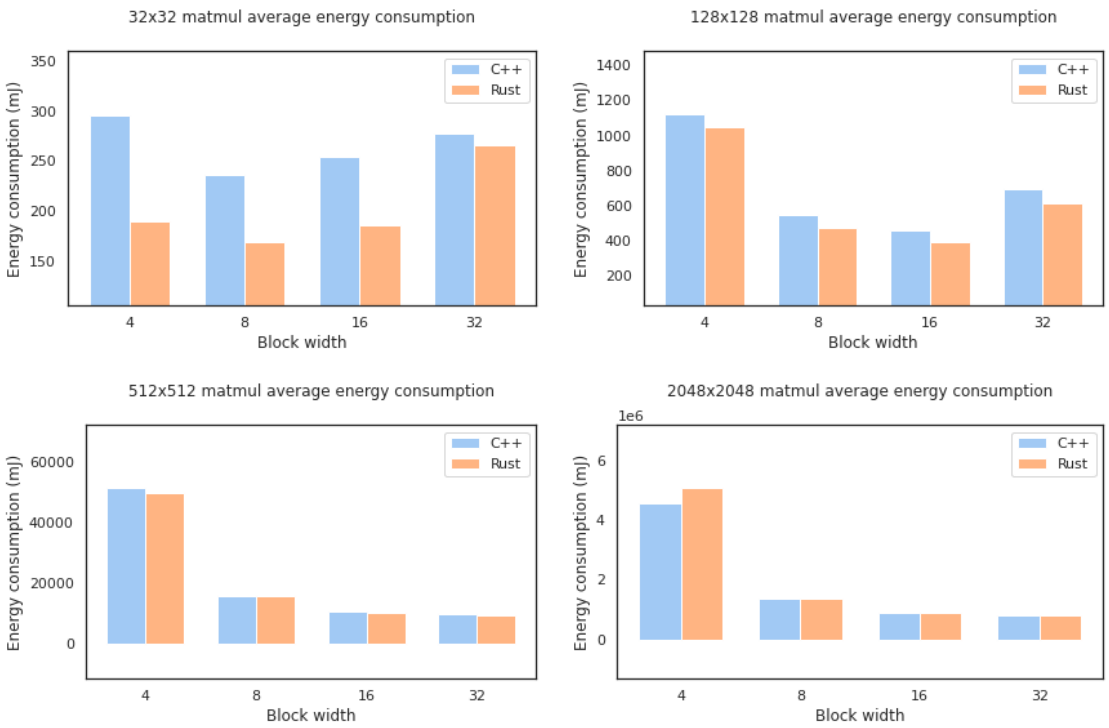


**Figure B.24:** Tiled matrix multiplication energy consumption difference between C++ and Rust without bounds checks, Machine 2.

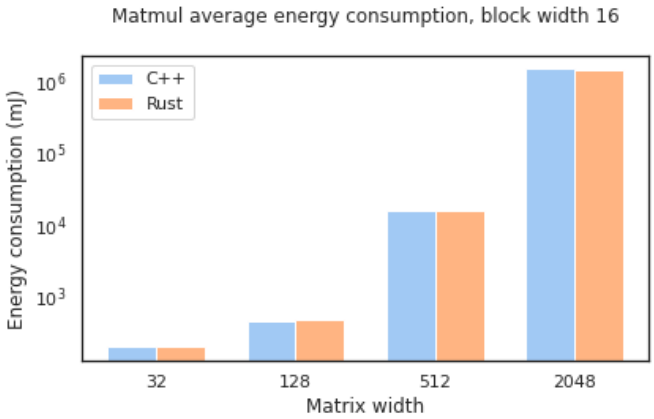


**Figure B.25:** Untiled matrix multiplication energy consumption difference between C++ and Rust without bounds checks, Machine 2. Y-axis in logarithmic scale.

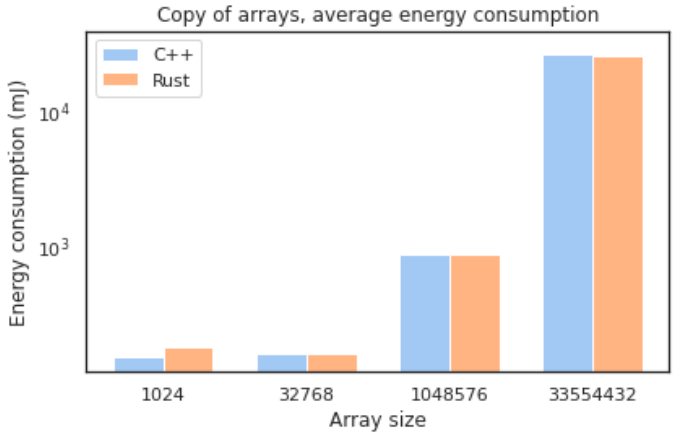


**Figure B.26:** Copy kernel energy consumption difference between C++ and Rust without bounds checks, Machine 2. Y-axis in logarithmic scale.