GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Rust Programming for HPC Application

Hoang Nam Nguyen

MatrNr: 21968578

Supervisor: Dr. Patrick Höhn

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2026

Abstract

The growing demand for high-performance computing infrastructure has intensified the need for software that is both efficient and correct. Programming languages that are traditionally used in HPC, such as C, C++ and Fortran, prioritize performance but place the full burden of memory safety and correctness on the developer, leading to bugs that are difficult to detect and costly to fix. Rust is a programming language designed to provide compile-time memory safety guarantees without sacrificing performance. However, its use in HPC applications and especially GPU-accelerated computing remains largely unexplored.

This report investigates whether Rust is a viable alternative to C++ for CUDA-based GPU programming by implementing and comparing Naive GEMM (General Matrix Multiplication) and Tiled GEMM in both languages, with cuBLAS as an optimized reference baseline. The results show that Rust achieves comparable or slightly superior performance to C++ for the Naive GEMM implementation in terms of execution speed and memory usage. However, the Rust-CUDA compiler backend produces significantly less optimized code for the Tiled GEMM implementation, unrolling the inner loop only 4 times compared to 32 times for NVCC. This results in up to only half the tiling speedup compared to the C++ implementation over the naive approach. The near-identical cuBLAS performance at larger matrix sizes suggests that the gap is attributable to compiler immaturity rather than fundamental language limitations and that Rust is a promising but not yet production-ready alternative for GPU HPC workloads.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☑ To write individual passages, altogether to the extent of 5% of the entire text
- ☑ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☑ Further, namely: Initial understanding of the subject material and code parts

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

BLAS Basic Linear Algebra Subprograms

CUDA Compute Unified Device Architecture

FFI Foreign Function Interface

GEMM General Matrix Multiplication

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

HPC High-Performance Computing

NVCC NVIDIA CUDA Compiler

NVVM NVIDIA Virtual Machine

PTX Parallel Thread Execution

SASS Streaming ASSembler

SCC Scientific Computing Cluster

SIMT Single Instruction, Multiple Thread

SLURM Simple Linux Utility for Resource Management

SM Streaming Multiprocessor

Gitlab Repository

The source code for this project, including all implementations and benchmark scripts, is publicly available at:

`https://gitlab.gwdg.de/hoangnam.nguyen01/scap-cuda-with-rust`

1 Introduction

In recent years there has been a massive increase in constructions of data centers around the globe, fueled mainly by the need for more computing power for high performance computing, artificial intelligence and also cloud infrastructure[Byl]. These constructions do not only require faster hardware than ever before but also software that can take advantage of the increasing computing power. The main aspects for HPC software are: Scalability, efficiency, correctness and reliability. To fulfill these requirements the language should be chosen carefully. Programming languages that are currently often used include C, C++ and Fortran which do not rely on many safety features in exchange for high potential computing advantages. The concerns about correctness and reliability are mainly put on the developer.

Rust is a systems programming language that combines performance with compile-time safety guarantees and therefore reduces the burden on the developer to account for all potential shortcomings of the written code. Prior work has explored Rust as an alternative to C for HPC workloads, finding comparable performance with reduced programming effort [Cos+21; Par+23]. In this report we will discuss the largely unexplored area of GPU programming using the language Rust with CUDA compared to the same implementation written in C++.

Currently there are many discussions about using Rust for these kinds of workloads but paradoxically at the same time little exemplary resources available on it. And the resources that are available are often outdated due to the young age of the language and the regular changes of both the Rust language but also open-source projects that are still in early development. This work is meant to be a contribution to the currently very early stages of CUDA development in Rust.

We will first introduce backgrounds in Rust and also CUDA, before continuing with the methodology. Afterwards we will explain the details of our implementations and the metrics used and follow that up with the results section. Lastly we discuss our findings, limitations, challenges and then we conclude the report.

2 Background

In this section we briefly explain the fundamental concepts which the report is based on. The first part is about the Rust programming language, followed by an explanation of the concepts of GPU Programming and CUDA. Then we look at the Rust-CUDA project which is an implementation of CUDA for Rust and the reason for this report. Lastly we will briefly introduce the commonly used General Matrix Multiplication benchmark for our performance comparison.

2.1 Rust

Rust is a systems programming language that has been in development since 2006, with its first stable release (version 1.0) in 2015 [MK14]. It was designed with three primary goals in mind: performance, reliability, and productivity. The language is used by major software companies including Mozilla, Google, Microsoft, and Amazon, and in 2025 it

became an official development language for the Linux kernel[Cor25]. This makes Rust the second high-level language used for writing Linux kernel next to C.

The main distinguishing safety feature of Rust is its **ownership model**: a compile-time memory management system based on three rules. Every value has exactly one owner, there can only be one owner at any time, and when the owner goes out of scope, the value is dropped and its memory is freed. The main benefit of this is that it prevents use-after-free vulnerabilities and memory leaks without requiring the user to maintain special attention to these possible errors.

Closely related is the concept of **borrowing and references**. Rather than transferring ownership, Rust code may borrow a value via a reference. The borrow checker is a component of the Rust compiler that enforces at compile time that at any moment a value may only either be immutably borrowed by any number of references, or mutably borrowed by exactly one reference, but not both simultaneously. This guarantees the absence of data races by construction[Jun+17]. For HPC applications, where race-conditions and memory bugs are a common problem, these compile-time guarantees are of significant value.

Beyond only safety, Rust achieves performance comparable to C and C++ through zero-cost abstractions. High-level language features such as iterators and closures compile to the same machine code a careful programmer would write by hand [MK14]. The language also provides a C-compatible foreign function interface (FFI), enabling straightforward interoperability with existing C and C++ codebases.

As for the last goal the language designers aim to provide high productivity through an extensive documentation and resources for studying. Rust's integrated package manager and build system **Cargo** handles dependency management and compilation. The compiler also produces not only descriptive error messages, but also suggests fixes, reducing development time[MK14].

2.2 GPU Programming

In recent years GPUs have become an important part of parallel programming as they enable substantially superior performance compared to CPUs in certain workloads. Starting out originally for graphics applications, they have been used for highly parallelizable applications thanks to their high number of cores [Owe+08].

Unlike CPUs, which are optimized for low-latency sequential execution with a small number of powerful cores, GPUs contain thousands of simpler cores designed for high-throughput parallel computation. For example, the NVIDIA RTX PRO 6000 contains 24,064 CUDA cores. GPUs execute work according to the **Single Instruction, Multiple Thread (SIMT)** model: threads are grouped into **warps** of 32, and all threads within a warp execute the same instruction in lockstep. When one warp stalls on a memory access, the hardware can immediately switch to another warp that is ready to execute, effectively hiding memory latency through massive parallelism [Owe+08].

The GPU memory hierarchy is central to achieving high performance. **Global memory** (device DRAM) is large but has comparatively high access latency. **Shared memory** is a fast, low-latency on-chip memory that is shared among all threads within the same thread block, enabling efficient data reuse and inter-thread communication. Individual threads also have access to **registers**, which are the fastest but most limited storage tier. Efficiently exploiting this hierarchy, mainly by loading data from global memory into shared memory and reusing it across threads, is key to obtaining optimal throughput

on GPU hardware.

2.3 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model introduced by NVIDIA in 2007 [Nic+08]. It extends C, C++, and Fortran with additional constructs for executing code on NVIDIA GPUs. Given NVIDIA's dominant share of the discrete GPU market, CUDA has become the de facto standard for GPU programming in scientific computing and HPC.

CUDA exposes GPU parallelism through a three-level hierarchy [NVI25]. Individual **threads** are grouped into **thread blocks**, and thread blocks are organized into a **grid**. All threads within the same block execute on the same Streaming Multiprocessor (SM) and can communicate via shared memory and synchronize using barriers. Blocks are scheduled independently across all available SMs, allowing CUDA programs to scale automatically to the available hardware.

CUDA code is compiled via a two-stage pipeline. The NVIDIA compiler NVCC first compiles the source code to **PTX** (Parallel Thread Execution), a virtual, architecture-independent instruction set designed by NVIDIA [NVI25]. At runtime, the NVIDIA driver then compiles PTX to **SASS** (Streaming ASSembler), the native instruction set of the specific target GPU. This design ensures forward compatibility: PTX code compiled for one GPU generation can be compiled and executed on future architectures. PTX is human-readable and provides direct insight into how the compiler has structured a computation, which makes it a useful artifact for comparing the code generation quality of different compilers, which is a focus of the analysis in this report.

CuBLAS is NVIDIA's GPU-accelerated implementation of the Basic Linear Algebra Subprograms (BLAS) specification, a widely adopted API for linear algebra operations[Don+90]. It provides highly optimized routines for operations such as General Matrix Multiplications, tuned specifically for each NVIDIA GPU architecture. In this report, cuBLAS serves as a performance reference baseline, and it is expected to substantially outperform the custom implementations due to these architecture-specific optimizations.

2.4 Rust-CUDA Project

The Rust-CUDA project is an open-source effort, currently maintained under the Rust-GPU organization on GitHub, that aims to enable writing and executing CUDA GPU kernels entirely in Rust [Rus23]. Its stated goals are to compile Rust code to PTX, provide Rust bindings for existing CUDA libraries, and offer a memory-safe and efficient alternative to C, C++, and Fortran for GPU programming.

The project provides several key crates. The `rustc_codegen_nvvm` crate operates as a custom compiler backend, intercepting the Rust compilation process and emitting NVVM IR, NVIDIA's LLVM-based intermediate representation, rather than standard LLVM IR. This NVVM IR is then compiled by NVIDIA's `libnvvm` library to PTX. The `cuda_std` crate provides GPU-side device functions and utilities analogous to CUDA's device standard library, while `cust` covers CPU-side CUDA runtime functionality such as device management, memory allocation, and kernel launching. Additional crates extend the ecosystem to specific domains: `cudnn` wraps GPU-accelerated deep learning primitives, and `optix` provides hardware ray tracing support.

A notable characteristic of writing GPU kernel code in Rust-CUDA is that kernel

code must be written in `unsafe` blocks, since raw pointer arithmetic and direct memory access cannot be verified by the borrow checker at compile time. This stands in contrast to standard Rust code, where the borrow checker enforces memory safety at compile time and shows that the safety guarantees Rust provides are not fully extended to GPU kernel code.

As of the time of this report, the project is still in early development and has only resumed development in early 2025. The documentation explicitly notes that bugs, safety issues, and missing features are to be expected [Rus23]. The toolchain requires a specific pinned nightly version of the Rust compiler and supports only part of the full CUDA API. These constraints have direct consequences for the implementations described in Section 6.2, and understanding them is part of the motivation for this work.

2.5 General Matrix Multiplication

A common benchmark for GPU programming is General Matrix Multiplication (GEMM), since it is often a fundamental part of larger GPU-based applications, such as neural networks. In the BLAS specification GEMM is defined as $C = \alpha \cdot (A \cdot B) + \beta \cdot C$. The parameters α and β are scalar parameters. In its simplest form, with $\alpha = 1$ and $\beta = 0$, the formula is reduced to the standard matrix product $C = A * B$.

3 Methodology

In this report we compare two different algorithms for GEMM, namely Naive GEMM and Tiled GEMM, implemented in both C++ and Rust to verify whether Rust is a viable alternative for HPC applications with a focus on CUDA. CuBLAS was used as a reference baseline.

As for the experimental setup, the comparisons were performed on the Scientific Computing Cluster (SCC) of the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG), with benchmarks submitted via SLURM's *sbatch*, each job requesting a single NVIDIA A100 GPU. The build was containerized using Apptainer, based on Nvidia's official CUDA 12.8.1 image on Ubuntu 24.04. The C++ CUDA code was compiled using NVCC 12.8.1 with GCC 12 as the host compiler directly on the SCC. An example dockerfile by the Rust-CUDA project was used as a foundation for the Apptainer image.

The Rust code was compiled using the pinned nightly toolchain *nightly-2025-08-04*, as required by the Rust-CUDA project. Notably, the Rust-CUDA codegen backend required a custom build of LLVM 7.1.0, which is now quite outdated, as that was mandated by the libnvvm dependency. As the nightly toolchain could not be loaded within the Apptainer environment on the SCC, the Rust binary was compiled locally and copied into the image prior to transferring it to the SCC for deployment.

Further instructions on running the programs can be found in the readme file of the repository of this project.

The benchmark evaluated GEMMs for square input matrices with dimensions of 32×32 to 8192×8192, doubling each step. A total of 20 executions were performed for each size, and the results were subsequently averaged. The primary metric measured was time in milliseconds that was recorded using CUDA events placed before and after the kernel launch the operations have been performed. This does not include the warmup performed

before the operations or the transfer of data to the GPU, and therefore only measures the kernel execution time. Additionally the usage of GPU memory was also tracked for the memory usage comparisons later on to identify any significant differences.

The final part of the analysis is comparing the PTX code generated by the different compilers of both languages. For the Rust implementation the PTX code is generated as part of the building process of cargo. The PTX output file can be found in the rust directory of the project under `GEMM/target/cuda-builder/nvptx64-nvidia-cuda/release/gemm\_kernels.ptx`. The PTX code of the C++ implementation on the other hand needs to be generated manually with `nvcc -ptx kernels.cu -o kernels.ptx`.

4 Implementation

In this section the implemented algorithms will be briefly explained, while also mentioning the minor differences between the Rust and C++ implementations. Generally the source logic is the same with minor language-specific differences. The C++ implementations were guided by a reference C++ CUDA implementation by Gautam [Gau23] and adapted for the benchmarks. The Rust implementations were developed by adapting these approaches alongside the code examples provided by the Rust-CUDA project [Rus23].

4.1 Naive GEMM

In the Naive GEMM implementation, each thread computes exactly one element of the output matrix C. Thread indices map directly to the row and column of the output element. Each thread then iterates over the full shared dimension k, accumulating the dot product of one row of A and one column of B. The result is written back using the full BLAS definition: $C[row, col] = \alpha \cdot sum + \beta \cdot C[row, col]$ (see Listing 1).

```
int row = blockDim.y*blockIdx.y + threadIdx.y;
int col = blockDim.x*blockIdx.x + threadIdx.x;

...

C[row,col] = alpha * sum + beta * C[row,col]
```

Listing 1: Mapping of elements for Naive GEMM in C++ CUDA

Both implementations use fixed 32x32 thread blocks, giving 1024 threads per block. Out-of-bounds threads are checked by a `if(row < m && col<n)` before write. The algorithm is considered naive because every thread independently reads its entire row from A and column from B directly from global memory, with no data reuse across threads in the same block. For k = 8192 this amounts to 16,384 global memory reads per thread. There are no optimizations done for the memory access, making global memory bandwidth the primary bottleneck.

The C++ and Rust kernels are structurally identical, differing only in syntax. C++ uses plain pointer arithmetic, while Rust passes raw pointers and performs arithmetic through unsafe pointer operations, as required for Rust CUDA kernel code.

4.2 Tiled GEMM

Instead of reading from global memory once per element of the dot product, threads cooperatively load a `TILE_SIZE` × `TILE_SIZE` sub-block of A and B into shared memory. The outer loop advances through the k-dimension in steps of `TILE_SIZE`. With each step, every thread loads one element into the shared tile, then all threads compute a partial dot product using only the fast shared memory. This reduces the number of global memory accesses.

Both implementations used `TILE_SIZE = 32` matching the block dimensions ($32 \times 32 = 1024$ threads). This is the largest `TILE_SIZE` that worked with the Rust implementation of Tiled GEMM.

Two barriers were required per phase to synchronize the threads. The first barrier, placed after tile loading, ensures that all threads have finished writing their element into shared memory, before any thread begins reading. The second barrier, placed after the dot product computation, ensures all threads have finished reading shared memory before the next phase overwrites it.

Between the two implementations, mainly the declaration of shared memory differs. In C++ the shared array is simply declared as a `__shared__ float`, while Rust requires a `MaybeUninit<f32>` with `#[address_space(shared)]` and raw pointer access via `addr_of_mut!`, since Rust does not allow uninitialized shared memory to be declared directly without `MaybeUninit`.

4.3 Metrics

The first metric, kernel execution time is measured by using CUDA events (`cudaEventRecord` / `cust::event::Event`) placed around the kernel launch, giving the time elapsed for the GPU kernel execution. For each matrix size, 2 warmup runs are performed before timing to ensure the GPU is in steady state and caches are warm and working optimally. The result time is the total time recorded divided by the number of total runs executed (in this case the number is always set to 20 executions).

GPU Memory usage is sampled after each run using `cudaMemGetInfo` / `cust::memory::mem_get_info`, which returns the total allocated memory on the device. The total GPU memory used is returned for further analysis.

5 Results

In this section we go over the results of our experiments and the performance comparisons between the implementations. First we go over the quantitative metrics of computing speed and memory usage. Then we analyze the PTX code generated by both compilers. The full results can be found in Table B1 and Figure A1.

5.1 Performance Comparisons

In this subsection of the report we present the performance results of each benchmark by directly comparing the observed differences between both implementation of each algorithm.

5.1.1 Computing Speed

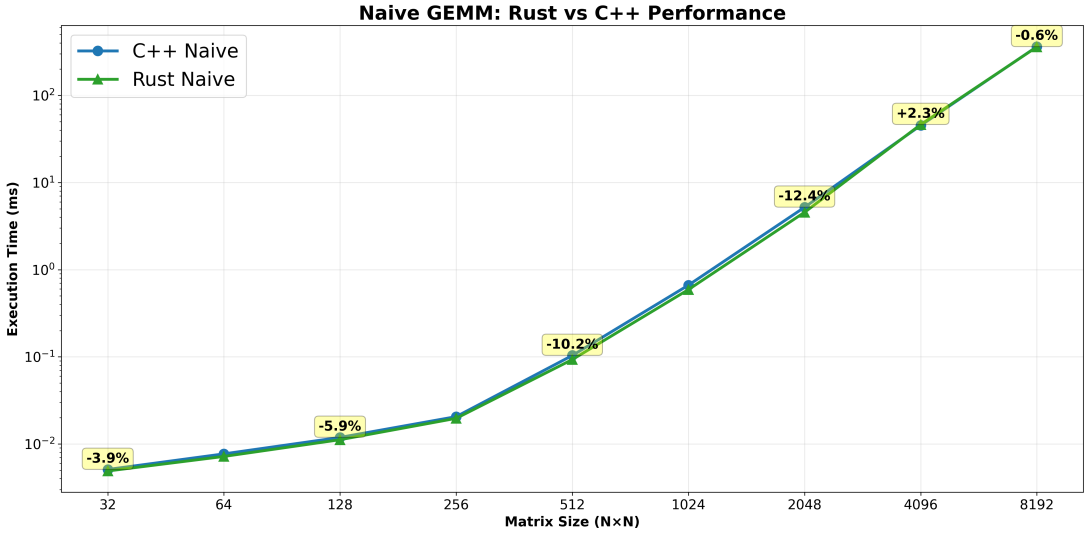

Figure 1: Computing speed comparison of Naive GEMM. Blue line for C++ and green line for Rust implementation.

First we analyze the results of the Naive GEMM implementations. The results of both algorithms can be seen in Figure 1. The Rust implementation is ahead of the C++ implementation by 3.9% on the smallest matrix size and reaches up to 11% for size 2048. At larger sizes starting with 4096 both reach near parity in performance. This suggests a small fixed overhead in the C++ implementation that becomes negligible relative to total execution time at larger matrix sizes.

Figure 2 depicts the performance gap between the Tiled GEMM implementation of Rust and C++. Contrary to expectations, the Tiled GEMM implementations in Rust performed almost the same as the Naive GEMM implementation until matrix size 512 where Tiled GEMM reaches a speedup of 10.3%. This performance increases to 38% at size 4096 and decreases to 34.9% at size 8192, which suggests diminishing or even reversing returns.

The C++ Tiled GEMM showed much better performance gains with consistently increasing gains across all matrix sizes, starting with 8.5% and monotonically increasing to 70% speedup at the largest size. At the largest size the performance increase of the C++ implementation is almost twice that of the Rust implementation.

Potential reasons for the discrepancy will be discussed in Section 6.1.

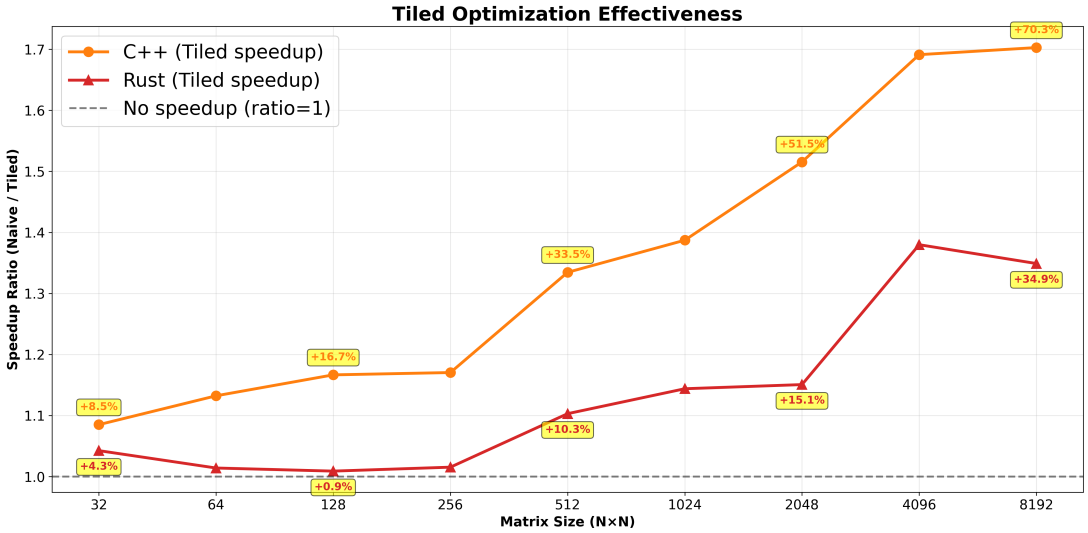

Figure 2: Comparison of speedup ratio of Naive over Tiled GEMM for Rust and C++.

Lastly the performance of native cuBLAS library in C++ and the integration into Rust is compared in Figure 3. As explained in Section 2.3 cuBLAS is an optimized library for linear algebra and we use it as our baseline. The results roughly match the findings of the observations for the Naive GEMM. The Rust integration of cuBLAS performs approximately 8% faster than the implementation in C++ for size 32 to size 2048. At sizes 4096 and 8192 the execution time is essentially the same. This also supports the idea that C++ has a small fixed overhead that becomes more negligible relative to the total execution time at larger matrices.

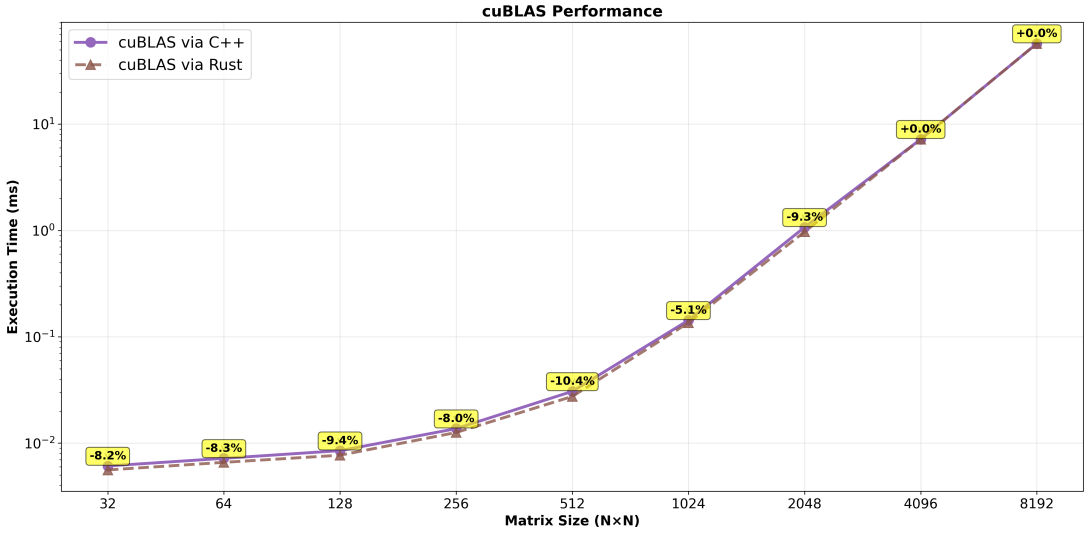

Figure 3: CuBLAS computing speed comparison.

Overall, the results indicate that Rust achieves comparable or slightly better performance to C++ for naive kernel implementations and the cuBLAS integration, but falls significantly behind once the tiling optimizations are applied. Despite the optimizations performed cuBLAS vastly outperforms all of our custom implementations at this point (see Figure A1). At the largest size the cuBLAS integration is 3.7 times faster than the C++ Tiled GEMM implementation and that one is 1.7 times faster than the Naive

GEMM implementations. Notably the cuBLAS integrations are slower than the custom implementations at matrix size 32.

5.1.2 Memory Usage

Generally the Rust GEMM implementation used less GPU memory than the implementation in C++ (see Table 1). On the left side of the table is the matrix size and how much memory the three matrices A, B, and C should take in MB. When subtracting the memory used for the three matrices from the total GPU memory used, we find that Rust has a constant 424–426 MB overhead while C++ has 508 MB overhead. This is the case for both Tiled GEMM and Naive GEMM as they both use the same amount of memory. This results in a consistent difference in memory usage between the two implementations of 82–84MB. This difference increases marginally from 82 MB to 84 MB for matrix sizes of 1024 and above. The cuBLAS implementations require approximately 10MB more than the respective custom implementations for both Rust and C++. The results will be further discussed in Section 6.1.

Size	Matrices (MB)	Rust (MB)	C++ (MB)	Δ C++−Rust (MB)
32	0.01	426.1	508.1	+82.0
64	0.05	426.1	508.1	+82.0
128	0.19	426.1	508.1	+82.0
256	0.75	426.1	508.1	+82.0
512	3.0	428.1	510.1	+82.0
1024	12.0	436.1	520.1	+84.0
2048	48.0	472.1	556.1	+84.0
4096	192.0	616.1	700.1	+84.0
8192	768.0	1192.1	1276.1	+84.0

Table 1: GPU memory usage for Rust and C++ GEMM implementations.

5.2 PTX Analysis

In this section we compare the PTX code generated by both the NVCC for C++ CUDA and compiler crate for Rust. Overall they both generally constructed correct code that achieves the same calculations and results, albeit with a difference in structure due to optimizations that each compiler found to be more optimal.

The naive version of both implementations produced an identical 4x unrolling of the inner loop in `L__BB0_4`. Overall the PTX code is similar, which also results in roughly the same performance.

Figure 4 shows the PTX code of the Rust Tiled GEMM implementation. This block is responsible for loading the values from the matrices A and B and calculating the partial dot product of the output matrix C. A total of 4 values are computed before the loop condition is checked. If the end condition has not been reached, the thread is redirected to the beginning of the loop to process the next tile. Otherwise, execution is finished. This shows that the LLVM backend recognizes the loop and partially unrolls it to 4 iterations, meaning that multiple iterations are performed in a single pass before the loop condition is rechecked, reducing the total number of condition checks per tile computation.

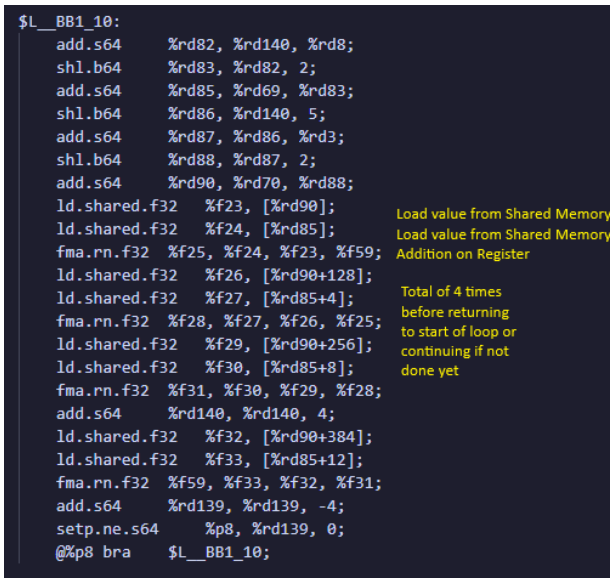

Figure 4: PTX code of Tiled GEMM in Rust highlighting 4 times unroll

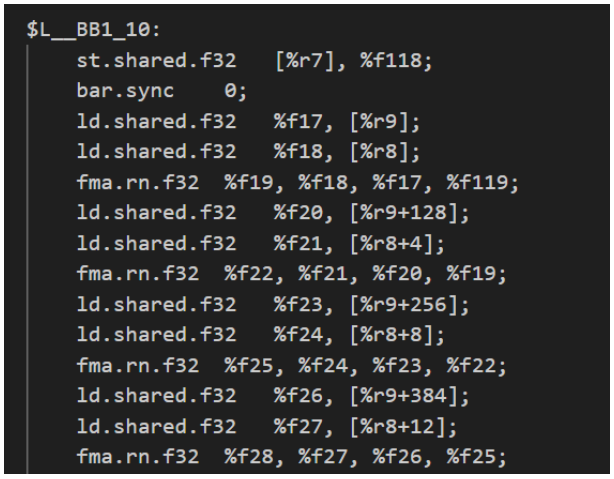

Figure 5: Snippet of the PTX code of Tiled GEMM in C++ highlighting unrolling. The full code unrolled the entire loop with 32 register additions

Figure 5 shows part of the PTX code created by the C++ implementation. NVCC managed to recognize the loop and aggressively unrolled the loop 32 times instead of only 4 times for the Rust compiler and therefore matching exactly the tile size of 32 of the Tiled GEMM algorithm. This results in fewer loop condition checks and therefore faster execution.

Additionally the PTX code generated by the Rust compiler often uses 64-bit values and registers, seen for instance by the `s64` or `%rd82` in contrast to the 32-bit registers referenced by `%r7` in the PTX code generated by the C++ implementation[NVI].

Another noteworthy difference is that both PTX files specify a target GPU architecture. For the C++ implementation it is `sm_52` and for Rust it is a newer `sm_75` referencing GPUs released with compute capabilities 5.2 and 7.5 respectively. Both implementations of Tiled GEMM source code did not specify which GPU architecture the kernel code was targeted for. There may be performance differences that relate to this as the used NVIDIA A100 is based on a different compute capability version.

6 Discussion

In this section we interpret the differences found between the two implementations and highlight the limitations and challenges found during the research period of this project.

6.1 Differences between Rust and C++

The results reveal several performance differences between the Rust and C++ implementations. We largely attribute them to the compiler's maturity and runtime differences rather than fundamental language limitations.

The Naive GEMM results show that Rust performs consistently faster than C++ by a small margin for smaller matrix sizes, with the gap closing at larger sizes. This correlates with the memory overhead findings in Section 5.1.2. The C++ CUDA runtime consistently allocates approximately 82–84 MB more than the Rust runtime regardless of matrix size. This suggests a heavier C++ runtime initialization that may manifest as a fixed overhead. At larger sizes this overhead becomes negligible relative to total execution time. The cuBLAS results further support this interpretation, as the Rust and C++ cuBLAS integration show a similar pattern.

The most significant finding is the performance gap between the Tiled GEMM implementations. The PTX analysis in Section 5.2 supports an explanation. NVCC unrolls the inner loop more aggressively to exactly 32 iterations, matching the tile size and therefore effectively eliminating the loop condition check within one run through the code. The Rust-CUDA backend only unrolls the loop 4 times which results in the loop condition being checked 8 times per tile. This may already explain most of the performance gap between the two implementations.

Nevertheless, it should be noted that the differing target GPU architectures specified in the generated PTX code may have influenced performance. The PTX code generated by the Rust implementation targets a newer GPU architecture that is more similar to the GPU used. Both differ from the actual architecture of the A100 used for benchmarking, but this might have led to differences. In principle, this difference should have minimal impact, as the PTX is compiled to native SASS instructions for the target GPU at runtime. However, the target architecture can influence the instructions and optimizations available at the PTX level, and this variable was not controlled in the current experimental setup. Future work should explicitly set a consistent target architecture across both compilers to eliminate this as a potential confound

The Rust implementation also often uses 64-bit registers for computations that the C++ compiler handles with 32-bit registers. Therefore Rust might be using unnecessarily large registers that could be used more efficiently with 32-bit registers.

Overall, these findings suggest that the performance gap between Rust and C++ is not inherent to the languages themselves, but mainly a consequence of the relative immaturity of the Rust-CUDA compiler backend compared to the highly optimized NVCC. The near-identical cuBLAS performance at large matrix sizes supports this conclusion. When both languages call the same underlying optimized libraries then the measured performance is the same.

6.2 Limitations

There are several limitations for this report, that should be considered when interpreting the results. Regarding generalizability, the benchmarks were performed exclusively on a single NVIDIA A100 GPU. Different GPU architectures may show different performance characteristics.

Furthermore, only GEMM was used as the benchmark workload. While GEMM is a representative and widely used benchmark in HPC, it is a single operation and does not capture the full range of computational patterns found in real HPC applications.

A concrete technical limitation encountered during implementation was the tile size restriction. The Rust-CUDA implementation of Tiled GEMM was limited to a `TILE_SIZE` of 32, as larger tile sizes caused the implementation to fail. No such restriction was encountered in the C++ implementation. The cause of this limitation could not be conclusively determined. This prevented exploring more optimal tile configurations.

The Rust-CUDA project itself introduces additional limitations. As noted in Section 2.4, the project is still in early development and explicitly warns of bugs, missing features, and safety issues. The toolchain also requires a specific nightly version of the Rust compiler, meaning that code written today may not compile with future Rust versions without modifications. This problem usually does not exist for the stable NVCC toolchain. Additionally, Rust-CUDA currently does not support Windows natively, limiting the accessibility for developers working on Windows-based systems.

The most significant structural limitation is that the Rust-CUDA project will likely always lag behind NVCC in terms of supported CUDA features. NVIDIA develops and maintains NVCC directly alongside new GPU architectures and CUDA releases, meaning new hardware features and compiler optimizations are available immediately in C++. Rust-CUDA is a community-maintained project and therefore must reverse-engineer or wrap these features afterwards, which introduces an inherent delay.

6.3 Challenges

The two most significant challenges were the scarcity of learning resources and the complexity of the toolchain setup. While CUDA programming in C++ has an extensive official documentation, textbooks, research papers and community resources, the Rust CUDA ecosystem offers very little beyond the project's own repository and their short code examples, which also included a recent docker file and a GEMM example, which was used as a foundation for the code of this project. This disparity is especially pronounced for optimization techniques. Resources on tiling, shared memory usage and register optimization are widely available for C++ CUDA but effectively non-existent for Rust-CUDA.

A public repository by the user Gautam [Gau23] was used as a guide for the optimization techniques, as it provided C++ CUDA implementations of tiling and shared memory optimization that were subsequently translated to Rust. Translating these optimization strategies from C++ to Rust often required trial and error to make it work with the Rust memory model.

From a toolchain perspective, the C++ CUDA setup was straightforward. NVCC and the CUDA runtime were available directly on the SCC through the apptainer image without issues. The Rust-CUDA toolchain, by contrast, required manual workarounds, as described in Section 3. These additional steps added significant development time and made iterating on Rust implementations considerably slower than the C++ side.

7 Conclusion

This report investigated whether Rust is a viable alternative to C++ for GPU-accelerated HPC applications, by using CUDA-based GEMM as a benchmark on the SCC on an NVIDIA A100 GPU. Two algorithms were implemented and compared in both languages. Naive GEMM, Tiled GEMM and cuBLAS were measured in terms of execution speed and memory usage.

The results show that Rust achieves comparable or slightly superior performance to C++ for the Naive GEMM implementation, with a consistent advantage of up to 11% for smaller matrix sizes that diminish at larger sizes. This advantage correlates with a consistently lower GPU memory usage of 82–84 MB compared to the C++ application.

However, the Tiled GEMM results reveal a significant performance gap. While the C++ implementation achieves up to 70% speedup over the naive approach, the Rust implementation reaches at most 38%. The PTX analysis identified the less aggressive loop unrolling by the Rust-CUDA compiler as the main difference, compared to the optimized full 32 times performed by the NVCC. Additionally the use of 64-bit registers for computations, while the C++ implementation only used 32-bit operations, might have had an influence.

In its current state, Rust-CUDA is not yet a production-ready replacement for C++ in GPU HPC workloads. The toolchain complexity, lack of documentation, limited CUDA API coverage, and immature compiler optimizations remain significant barriers. Nevertheless, the findings suggest that Rust has strong potential for HPC applications with its memory safety guarantees, lower runtime overhead and its comparable performance in the Naive GEMM and cuBLAS benchmarks.

References

[Byl] Byline. *Building data centers bigger, faster*. Accessed: 2026-03-29. URL: https://www.mckinsey.com/industries/private-capital/our-insights/scaling-bigger-faster-cheaper-data-centers-with-smarter-designs.

[Cor25] Jonathan Corbet. *The (Successful) End of the Kernel Rust Experiment*. https://lwn.net/Articles/1049831/. Accessed: 2026-01-17. 2025.

[Cos+21] Manuel Costanzo et al. "Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body". In: Oct. 2021, pp. 1–10. DOI: 10.1109/CLEI53233.2021.9640225.

[Don+90] J. J. Dongarra et al. "A set of level 3 basic linear algebra subprograms". In: *ACM Trans. Math. Softw.* 16.1 (Mar. 1990), pp. 1–17. ISSN: 0098-3500. DOI: 10.1145/77626.79170. URL: https://doi.org/10.1145/77626.79170.

[Gau23] Tushar Gautam. *CUDA-C*. Accessed: 2026-01-16. 2023. URL: https://github.com/tgautam03/CUDA-C.

[Jun+17] Ralf Jung et al. "RustBelt: securing the foundations of the Rust programming language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158154. URL: https://doi.org/10.1145/3158154.

[MK14] Nicholas Matsakis and Felix Klock. "The rust language". In: *ACM SIGAda Ada Letters* 34 (Oct. 2014), pp. 103–104. DOI: 10.1145/2692956.2663188.

[Nic+08] John Nickolls et al. "Scalable parallel programming with CUDA". In: *ACM SIGGRAPH 2008 Classes*. SIGGRAPH '08. Los Angeles, California: Association for Computing Machinery, 2008. ISBN: 9781450378451. DOI: 10.1145/1401132.1401152. URL: https://doi.org/10.1145/1401132.1401152.

[NVI] NVIDIA Corporation. *PTX ISA 9.2 documentation*. Accessed: 2025-12-19. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#parameterized-variable-names.

[NVI25] NVIDIA Corporation. *CUDA C++ Programming Guide*. Version 12.8. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[Owe+08] John D. Owens et al. "GPU Computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899. DOI: 10.1109/JPROC.2008.917757.

[Par+23] John Parrish et al. "Towards Safe HPC: Productivity and Performance via Rust Interfaces for a Distributed C++ Actors Library (Work in Progress)". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. MPLR 2023. Cascais, Portugal: Association for Computing Machinery, 2023, pp. 165–172. ISBN: 9798400703805. DOI: 10.1145/3617651.3622992. URL: https://doi.org/10.1145/3617651.3622992.

[Rus23] Rust-GPU Contributors. *Rust-CUDA: Ecosystem of Libraries and Tools for Writing and Executing Fast GPU Code Fully in Rust*. https://github.com/Rust-GPU/Rust-CUDA. Accessed: 2025-12-19. 2023.

A Plots

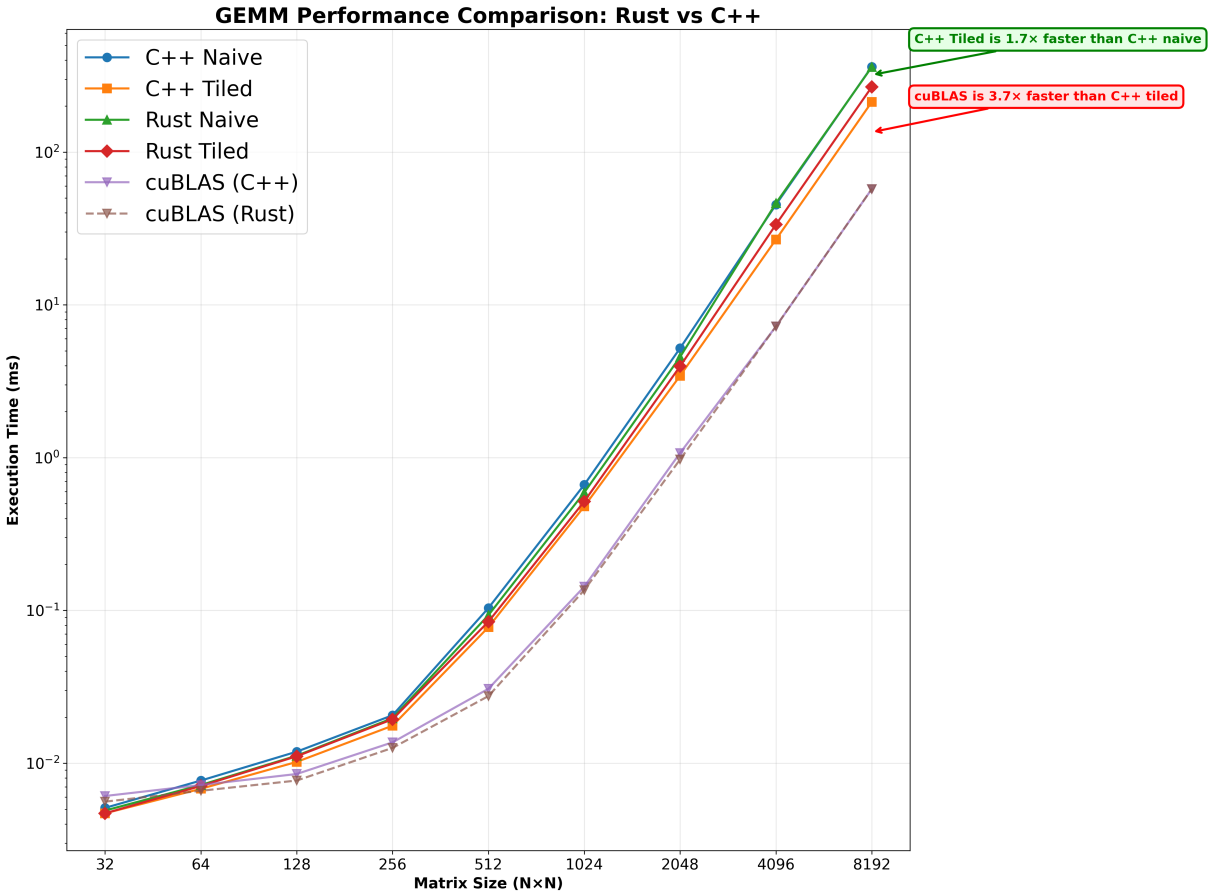

Figure A1: Overall performance comparison

B Tables

Table B1: GEMM Benchmark Results: C++ vs. Rust

		C++		Rust	
Method	**Size**	**Time (ms)**	**Memory (KB)**	**Time (ms)**	**Memory (KB)**
cuBLAS	32	0.0061	444480	0.0056	444480
	64	0.0072	520256	0.0066	444480
	128	0.0085	520256	0.0077	444480
	256	0.0137	520256	0.0126	444480
	512	0.0307	522304	0.0275	446528
	1024	0.1431	532544	0.1358	456768
	2048	1.0733	569408	0.9730	493632
	4096	7.2188	716864	7.2224	641088
	8192	57.3491	1306688	57.3599	1230912
gemm_naive	32	0.0051	444480	0.0049	436288
	64	0.0077	520256	0.0072	436288
	128	0.0119	520256	0.0112	436288
	256	0.0206	520256	0.0197	436288
	512	0.1037	522304	0.0931	438336
	1024	0.6649	532544	0.5913	446528
	2048	5.2059	569408	4.5607	483392
	4096	45.2565	716864	46.3058	630848
	8192	362.4268	1306688	360.2269	1220672
gemm_tiled	32	0.0047	520256	0.0047	436288
	64	0.0068	520256	0.0071	436288
	128	0.0102	520256	0.0111	436288
	256	0.0176	520256	0.0194	436288
	512	0.0777	522304	0.0844	438336
	1024	0.4793	532544	0.5169	446528
	2048	3.4352	569408	3.9633	483392
	4096	26.7608	716864	33.5581	630848
	8192	212.8484	1306688	267.0289	1220672