Paris-Saclay University

Master 2 High Performance Computing, Simulation
End-of-studies internship report

Rust and GPU programming

Author :
Gabriel Dos Santos

Supervisors :
Cédric Chevalier
Soraya Zertal

Abstract

In recent years, the field of High Performance Computing (HPC) has known significant advances in hardware technology. With the advent of heterogeneous architectures, these improvements imply an ever-increasing need for hardware accelerator programming, which is essential to harness the computational performance of exascale supercomputers.
To meet these new requirements, software engineering and programming languages are evolving to improve programmers' control and safety over parallel applications. Rust is a modern programming language focused on performance, memory safety, and concurrency. Its features make it an ideal choice for guaranteeing the robustness and efficiency of compute-intensive codes, particularly when parallelizing certain forms of data-flow algorithms.

In this context, the CEA uses Rust to develop several applications and tools, some of which could benefit from the accelerations offered by GPUs. This internship aims to explore the capabilities of the Rust language for programming hardware accelerators. This work consists of a detailed overview of the current state of the art, performance analysis depending on the chosen code generation method, and a proof of concept by porting partitioning algorithms from CEA's `coupe` library on NVIDIA GPUs.

CEA, DAM, DIF, F-91297, Arpajon, France

September 01, 2023

Table of contents

Acknowledgments

I want to express my most sincere appreciation for my supervisor at the CEA, Cédric Chevalier, who supported me throughout this internship and always provided me with good advice on how to improve. His deep technical insight into efficient software engineering and expert knowledge of modern hardware architectures were of tremendous help when I got stuck on challenging problems.

I am also especially grateful to my co-supervisor, Hugo Taboada, for his help and advice whenever I had questions. He accompanied me throughout this internship and this endeavor would not have been possible without him.

Thanks should also go to Thao, Hélène, Nathalie, and all the Teratec Campus staff who welcomed me daily for their availability and cheerfulness.

I am also thankful to my family who supported me during those six months. A special thanks to my girlfriend who put her graphic design skills to good use and helped me make some of the diagrams and figures presented in this report.

Finally, I would like to have a word for my fellow interns who have paced my days and with whom I was able to have deeply geeky conversations that sparked brilliant ideas for my work. Many thanks for all the coffee breaks and the intense board games after lunch!

Host institution

The CEA

As a significant actor in research, development and innovation, the French Atomic Energy and Alternative Energies Commission (CEA) operates in four fields:

- defense and security;
- low-carbon energies (nuclear and renewable);
- technological research for industry;
- fundamental research (material and life sciences).

Drawing on its recognized expertise, the CEA is involved in setting up collaborative projects with numerous academic and industrial partners. The CEA is based in 9 centers throughout France. It is developing numerous partnerships with other research organizations, local authorities and universities. As such, the CEA is a stakeholder in the national alliances coordinating French research in the fields of energy (ANCRE), life sciences and health (AVIESAN), digital sciences and technologies (ALLISTENE), environmental sciences (AllEnvi) and human and social sciences (ATHENA). Recognized as an expert in its fields of competence, the CEA is fully integrated into the European research space and has a growing presence at the international level. The CEA employs 19,925 technicians, engineers, researchers and staff with a budget of 5 billion euros (figures published at the end of 2018).

The Department of Military Applications (DAM)

A division dedicated to deterrence

CEA's Military Applications Division (DAM) designs, manufactures, maintains and dismantles the nuclear warheads used by France's airborne and naval nuclear forces. The DAM is responsible for designing and producing reactors and nuclear cores for French Navy vessels, submarines and aircraft carriers. It supports the French Navy in the in-service monitoring and maintenance of its reactors. The DAM is also responsible for the supply of strategic nuclear materials for deterrence purposes. In a world undergoing profound upheaval, the DAM also contributes to national and international security through the technical support it provides to the authorities in the fight against nuclear proliferation, terrorism and disarmament. Since the transfer of the Gramat center in 2010 from the Direction Générale de l'Armement (DGA) to the CEA, the DAM has been providing its expertise to the French Defense Ministry in the field of conventional armaments.

A division open to research

The national and international sharing of knowledge (where possible), exposure to external scientific assessment, and integration into networks of expertise all guarantee scientific credibility. Each year, DAM teams produce around 2,000 publications and scientific papers. The DAM's open approach also involves making its experimental resources available to the research community and enabling its teams to contribute to other research programs.

A division driving France's industrial policy

The DAM essentially shares its activities with the French industry: over two-thirds of its budget is spent on purchases from the latter, with the remaining third divided between staff salaries (one-fifth) and taxes.
DAM's industrial policy is unique in more ways than one:

- firstly, because the DAM retains overall prime contractor ship for the vast majority of the systems for which it is responsible: it thus ensures the right balance between the major defense industrial groups and the often innovative SMEs by contracting directly with the latter, thus enabling them to receive fair remuneration for their production;
- secondly, because an explicit distribution of work underpins the distribution of its budget: the DAM conducts research in its laboratories thanks to its high-level scientific and technological staff.

Once a product has been defined, the DAM transfers the definition and the processes to the industrialists, who then develop and produce it.

The DAM also aims to ensure that its centers participate in local economic life through their involvement in competitive clusters. Outside its own field of application, the DAM promotes its research by transferring technology to industry and registering numerous patents.

The format

DAM comprises five centers with homogeneous missions, whose activities are divided between basic research, development and manufacturing:

- DAM Ile-de-France (DIF), at Bruyères-le-Châtel, carries out weapons physics, numerical simulation and nuclear counter-proliferation activities. DIF is also the center responsible for engineering at DAM. Finally, the INBS-Propulsion Nucléaire at the CEA/Cadarache center, in the Provence Alpes-Côte d'Azur region, is attached to the DIF center and houses the onshore testing facilities and part of the nuclear propulsion manufacturing;
- Cesta, in the Aquitaine region, is dedicated to weapons architecture and environmental testing. It is also home to the Megajoule Laser, a major simulation facility;
- Valduc, in Burgundy, is dedicated to nuclear materials and the Epure experimental facility of the Simulation program;
- Le Ripault, in the Centre region of France, dedicated to non-nuclear materials (chemical explosives, etc.);
- Gramat, (formerly DGA) in the Midi-Pyrénées region, conducts system vulnerability and weapons effectiveness activities on behalf of the French Defense Ministry.

The DAM Île-de-France center

CEA/DAM - Île de France (DIF) is one of DAM's operational divisions. The DIF site employs around 2,000 CEA staff and welcomes around 600 employees from outside companies daily. It is located in Bruyères-le-Châtel, about 40 km south of Paris, in the Essonne department.
DIF's missions include :

- the design and guarantee of nuclear weapons, thanks to the Simulation program. The challenge is to reproduce the different phases in the operation of a nuclear weapon and to compare these results with measurements from past nuclear firings and experimental results obtained on current facilities (radiographic machine, power lasers, particle accelerators);
- the effort against proliferation and terrorism, in particular by contributing to the Non-Proliferation Treaty safeguards program and by providing French technical expertise for the implementation of the Comprehensive Nuclear Test Ban Treaty (CTBT);
- scientific and technical expertise for the construction and dismantling of complex structures, as well as for environmental monitoring and earth sciences;
- alerting the authorities, an operational mission carried out 24 hours a day, 365 days a year, in the event of nuclear tests, earthquakes in France or abroad, and tsunamis in the Euro-Mediterranean zone. The DIF provides the authorities with related analyses and technical summaries.

Since 2003, the DAM Île-de-France center has been home to the CEA's scientific computing facilities, which bring together all the CEA's supercomputers:

- the EXA1 supercomputer for the CEA/DAM Simulation program, successor to TERA 1000, with 23.2 petaflops computing power, i.e., capable of performing 23.2 million billion floating-point operations per second.
- Computers at the Centre de Calcul pour la Recherche et la Technologie (CCRT), open to the research community and industry, for a total power of 8.8 petaflops.
- The 22 petaflops Joliot-Curie supercomputer, the second in a network of petaflops-class supercomputers for researchers in the European scientific community. This supercomputer is housed at the TGCC (Très Grand Centre de Calcul) and operated by CEA teams, thus contributing to France's participation in the PRACE (Partnership for Advanced Computing in Europe) project.

1 Introduction

In the last ten years, HPC has experienced a dramatic shift in computer architecture, slowly moving away from general-purpose central processing units (CPU) and instead turning towards heterogeneous systems with specialized hardware designed to accelerate computations. The use of graphical processing units (GPU), field programmable gate arrays (FPGA), or even application-specific integrated circuits (ASIC) has increased significantly in modern supercomputers, often outnumbering CPUs by a factor of four in the systems that have most recently entered the TOP500 ranking. This change leads to a growing need for efficient software that exploits the computational performance unlocked by such accelerators. Even more recently, the surging of artificial intelligence (AI) has pushed performance requirements even further with extensive reliance on GPU architectures, which are especially well-suited for these workloads. This leads to a rapid convergence between HPC and AI, in which both fields depend on similar hardware but accommodate different computational demands.

To meet these new criteria and take advantage of the performance improvements offered by GPUs, the software has to change, which involves rewriting significant portions of existing applications. This endeavor is not trivial, and fully exploiting these accelerators requires comprehensive knowledge of GPU architecture. Moreover, rewrites often induce complex communications between CPU and GPU to transfer the data between their respective memory space. These come at a high cost that is difficult to offset, as memory latency and bandwidth have improved very slowly compared with the hardware computing performance. In the pursuit of efficiency, programming languages offer modern tools to work with accelerators, either by offering low-level control of the device (e.g., CUDA C++, OpenCL C, etc.) or by providing higher-level concepts that abstract over architectural details, sometimes sacrificing performance in favor of better code portability (e.g., SYCL, Kokkos, OpenMP, etc.).

The Rust programming language is a newcomer in the field of high-performance compiled languages, with its first stable release in 2015. It aims to solve most of the memory and type safety issues that exist in C and C++ while maintaining equivalent performance. It also puts a significant accent on correctness in concurrency contexts by eliminating an entire class of data race bugs thanks to its borrow checker. Rust thus provides robust safety guarantees without performance penalties, packed in a modern syntax with many functional features that align well with the current trends in software engineering.

This internship aims to evaluate the viability of Rust as a GPGPU programming language in the context of scientific computing and HPC. In particular, the goal is to determine if we can leverage some of the language's properties to guarantee the robustness, memory and thread safety of GPU codes developed at the CEA.

2 Context of the internship

2.1 Hardware accelerators

In this section, we give an overview of hardware accelerators. We introduce the architecture of a GPU, illustrated with current NVIDIA cards, and how they integrate into modern heterogeneous systems. Then, we present the existing programming models used to write code targeting such hardware. Finally, we cover their performance benefits and relevant use cases in HPC.

2.1.1 GPU architecture

While CPUs are optimized to compute serial tasks as quickly as possible, GPUs are instead designed to share work between many small processing units that run in parallel. They trade reduced hardware capabilities in program logic handling for much higher core counts, emphasizing parallel data processing. As a result, GPUs prioritize high throughput over low latency, allowing them to outperform CPUs in compute-intensive workloads that can be trivially parallelized, making them particularly well suited to compute-bound applications.

In this section, we will use the Ampere architecture as an example, as it is described in NVIDIA's whitepaper [1]. We will also provide the terminology for equivalent hardware components on AMD GPUs.

Figure 1: Block diagram of the full NVIDIA GA100 GPU implementation

Figure 1 shows the compute and memory resources hierarchy available on NVIDIA's data center GA100 GPU, designed for HPC and machine learning workloads.

Figure 2 presents the Streaming Multiprocessor (SM) of the GA100 GPU. SMs are the fundamental building block of NVIDIA GPUs and are comparable to the Compute Unit (CU) in AMD terminology. Each SM is a highly parallel processing unit containing multiple CUDA cores (or shader cores on AMD) and various specialized hardware units. It achieves parallel execution through the Single Instruction, Multiple Thread (SIMT) technique, allowing multiple CUDA cores within the SM to execute the same instruction on different data simultaneously. Threads are scheduled and executed in groups of 32, called "warps" (or "wavefronts" in AMD terminology), thus promoting data parallelism. The SM also manages various types of memory, including fast on-chip registers, instruction and data caches, and shared memory for intra-thread block communication. Additionally, it provides synchronization mechanisms to coordinate threads within a block.

Starting from the Volta architecture in 2017 [2], NVIDIA SMs introduced an acceleration unit called the Tensor Core, purposefully built for high-performance matrix multiplication and accumulation operations (MMA), which are crucial in AI and machine learning workloads. However, these specialized cores only provide noticeable performance improvements for mixed-precision data types, reducing their usefulness for most HPC applications that work with double-precision (64-bit) floating-point values. On the full implementation of NVIDIA's GA100 GPU, there are 128 SMs and 64 single-precision (32-bit) floating-point CUDA cores per SM, enabling the parallel execution of up to 8192 threads.

Figure 2: Streaming Multiprocessor (SM) of NVIDIA A100 GPU

NVIDIA GPUs expose multiple levels of memory, each with different capacities, latencies and throughputs. We present them hereafter from fastest to slowest:

1. **Registers** are the fastest and smallest type of memory available on the GPU, located on the individual CUDA cores (SM units), providing very low latency and high-speed data access. Registers store local variables and intermediate values during thread execution.
2. **L1 Cache/Shared Memory** is a small, fast, on-chip memory shared among threads within the same thread block (see Section 2.1.2.1 and Figure 3). This cache level can either be managed automatically by the GPU or managed manually by the programmer and treated as shared memory between threads. This allows them to communicate and cooperate on shared data. Shared memory is particularly useful when threads need to exchange information or access contiguous data with reduced latency compared to accessing global memory.
3. **L2 Cache** is a larger on-chip memory that serves as a cache for global memory accesses. It is shared among all the SMs in the GPU. L2 cache helps reduce global memory access latency by storing recently accessed data closer to the SMs.
4. **Global Memory** is the largest and slowest memory available on the GPU as it is located off-chip in the GPU's dedicated Video Random Access Memory (VRAM). Global memory serves as the main memory for the GPU and is used to store data that needs to be accessed by all threads and blocks. However, accessing global memory has higher latency than on-chip memories described above. Global memory generally comprises either Graphic Double Data Rate (GDDR) or High-Bandwidth Memory (HBM), which provides higher throughput in exchange for higher latencies.
5. **Host Memory** refers to the system's main memory, i.e., the CPU's RAM. Data transfers between the CPU and the GPU are necessary for initializing data, transferring results back to the host, or when data does not fit within the GPU's global memory. Data transfers between host and GPU memory often involve much higher latencies because of the reduced bus bandwidth between the two hardware components (implemented using, e.g., PCIe buses).

Figure 3 showcases how these kinds of memory are typically organized on an NVIDIA chip.

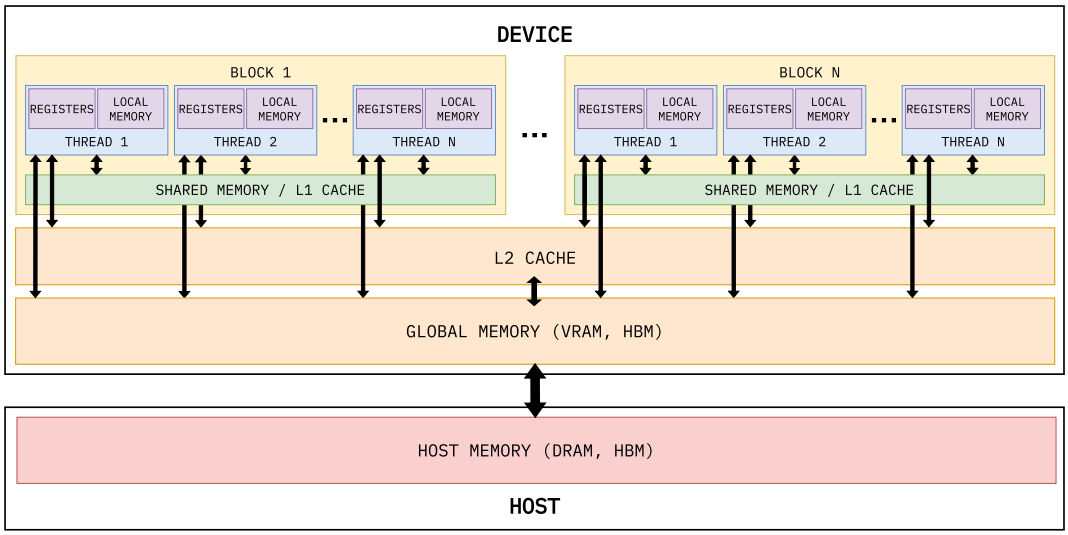

Figure 3: Generic memory hierarchy of NVIDIA GPUs

The increased integration of GPUs in modern HPC systems requires fast interconnect networks that enable the use of distributed programming models. As most supercomputers use a combination of 2-4 GPUs per CPU (or per socket), there need to be two levels of interconnect fabric:

1. Inter-GPU networks, generally comprised of proprietary technologies (e.g., NVLink on NVIDIA, Infinity Fabric on AMD, etc.), ensuring the fastest possible data transfers between nearby GPUs.
2. Inter-node networks allowing fast, OS-free Remote Direct Memory Accesses (RDMA) between faraway GPUs.

2.1.2 Programming models

GPU programming models refer to the different approaches and methodologies used to program and utilize GPUs for general-purpose computation tasks beyond their traditional use cases in graphics rendering. This section introduces some of the programming models used in HPC based on their abstraction level. Firstly, we present low-level models that closely map to the underlying hardware architecture. Secondly, we showcase higher-level programming styles that offer more expressiveness and portability, often at the expense of a high degree of fine-tuned optimization.

We start by introducing common concepts for GPU programming. We define the terms starting from the most high-level view and gradually refine them toward smaller components of GPU programming.

2.1.2.1 Definitions

1. **Kernel:** A kernel is a piece of device code, generally composed of one or multiple functions, that leverages data parallelism (SIMD) and is meant to execute on one or multiple GPUs. A kernel must be launched from the host code, although it can be split into multiple smaller kernels that are called from the device.
2. **Grid:** A grid is the highest-level organizational structure of a kernel's execution. It encompasses a collection of blocks, also called work groups and manages their parallel execution. Kernels are launched with parameters that define the configuration of the grid, such as the number of blocks on a given axis. Multiple grids — i.e., kernels — can simultaneously exist on a GPU to efficiently utilize the available resources.
3. **Block:** A block, also called thread block (CUDA), workgroup (OpenCL), or team/league (OpenMP), is a coarse-grain unit of parallelism in GPU programming. It is the main component of grids and represents a collection of threads working together on parts of the data operated on by

a kernel. Like grids, the dimensions of blocks can be configured when launching a kernel. Threads in a block can share memory in the L1 cache (see Figure 3), which enables better usage of this resource by avoiding expensive, repeated reads to the GPU's global memory.

4. **Warp:** A warp (also called wavefront in AMD terminology) is a fine-grain unit of parallelism in GPU programming, very much related to the hardware implementation of a GPU. However, it also appears at the software level in some programming models. On NVIDIA devices, threads inside a block are scheduled in groups of 32, which programmers can take advantage of in their kernels (e.g., warp-level reductions).
5. **Thread**: A thread is the smallest unit of parallelism in GPU programming. They are grouped in blocks and concurrently perform the operations defined in a kernel. Each thread executes the same instruction as the others but operates on different data (SIMD parallelism).

Figure 4 summarizes these structures as exposed to programmers in a CUDA-style programming model, which we introduce in the next section.

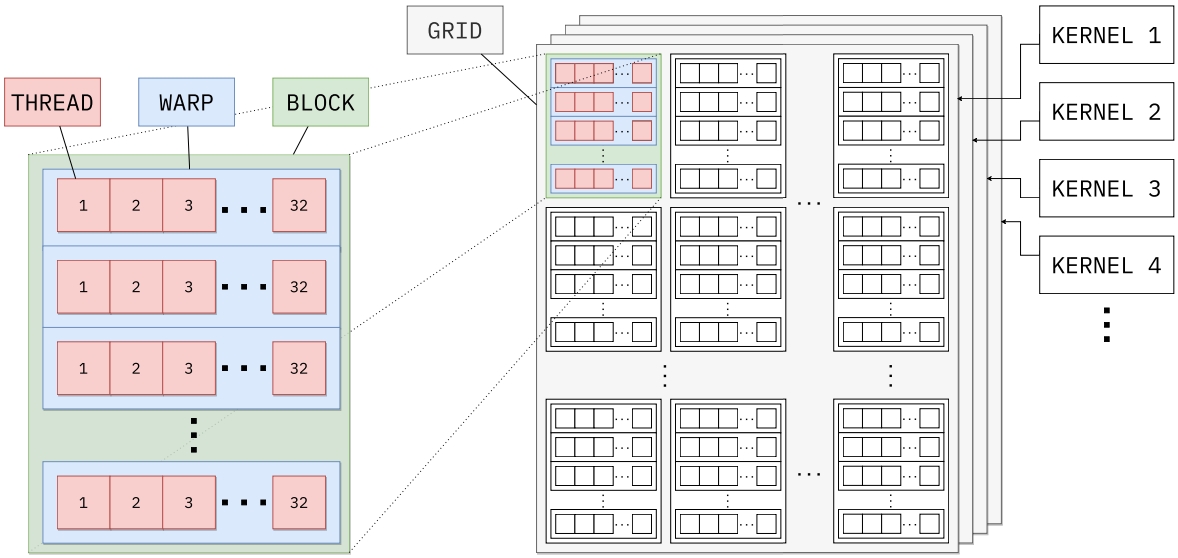

Figure 4: General compute model for CUDA-style GPU programming

2.1.2.2 Low-level

Low-level programming models strive to operate as closely as possible to the underlying hardware. Consequently, kernels are articulated using specialized subsets of programming languages like C or C++. Such frameworks provide developers with the essential tools and abstractions to accurately represent the accelerator's architecture, enabling them to write highly optimized kernels.

Computed Unified Device Architecture (CUDA) [3] is NVIDIA's proprietary tool for GPU programming. Using a superset of C/C++, it provides a parallel programming platform and several APIs, allowing developers to write kernels that will execute on NVIDIA GPUs, locking users into the vendor's ecosystem. However, CUDA is one of the most mature environments for GPU programming, offering a variety of highly optimized computing libraries and tools. Thanks to the vast existing codebase, CUDA is often the default choice for GPU programming in HPC.

Heterogeneous-Compute Interface for Portability (HIP) [4] is the equivalent of CUDA for AMD GPUs. It is part of the RadeonOpenCompute (ROCm) software stack. Contrary to its NVIDIA equivalent, HIP is open-source, making it easier to adopt as it does not lock users into a specific vendor ecosystem. It provides basic compute libraries (BLAS, DNN, FFT, etc.) optimized for AMD GPUs and several tools to port CUDA code to HIP's syntax automatically. It is quickly gaining traction as AMD is investing a lot of resources into its GPU programming toolkit in order to catch up with NVIDIA in this space. From an HPC standpoint, its hardware advantage over NVIDIA is also enabling AMD to improve adoption in the domain.

Open Compute Language (OpenCL) [5] is developed as an open standard by Khronos Group for low-level, vendor-agnostic hardware-accelerator programming. Unlike CUDA and HIP, OpenCL supports offloading to GPU and FPGA and can even fall back to the CPU if no devices are available. It provides APIs and tools to allow programmers to interact with devices, manage memory, launch parallel executions on the GPU, and write kernels using a superset of C's syntax. The standard defines common mechanisms that make the programming model portable, similar to what Figure 4 showcases. Because of its focus on portability, OpenCL implementations can have performance limitations compared to specialized programming models such as CUDA or HIP. Most GPU vendors (NVIDIA, AMD, Intel) supply their own implementation of the OpenCL standard optimized for their hardware. Some open-source implementations of OpenCL and OpenCL compute libraries also exist.

2.1.2.3 High-level

In contrast to low-level programming models, high-level programming models focus on portability, ease of use and expressiveness. They are much more tightly integrated into the programming language they are used in — generally C++ — and offer an intuitive way of writing GPU code. Kernels' syntax and structure closely resemble typical CPU code, simplifying the process of porting them to various target architectures. Most of the "hardware mapping" (i.e., translating CPU constructs to suit the architecture of hardware accelerators) and optimization work is delegated to the compiler and runtime.

SYCL [6] is a recent standard developed by the Khronos Group that provides high-level abstractions for heterogeneous programming. Based on recent iterations of the C++ standard (C++17 and above), it aims to replace OpenCL by simplifying the process of writing kernels and abstracting the usual low-level compute model (see Figure 4). Kernels written in SYCL look very much like standard CPU code does. However, they can be offloaded to hardware accelerators as the user desires, in an approach similar to what programmers are used to with OpenCL.

Open Multi-Processing (OpenMP) [7] is an API specification for shared-memory parallel programming that also supports offloading to hardware accelerators. It is based on compiler directives for C/C++ and Fortran programming languages. Similarly to SYCL, standard CPU code can be automatically offloaded to the GPU by annotating the relevant section using OpenMP `pragma omp target` clauses. OpenMP is well-known in the field of HPC as it is the primary tool for fine-grain, shared-memory parallelism on CPUs.

Kokkos [8] is a modern C++ performance portability programming ecosystem that provides mechanisms for parallel execution on hardware accelerators. Unlike other programming models, Kokkos implements GPU offloading using a variety of backends (CUDA, HIP, SYCL, OpenMP, etc.). Users write their kernels in standard C++ and can choose their preferred backend for code generation. Kokkos also provides useful memory abstractions, tools, and compute libraries targeted for HPC use cases.

2.1.3 Performance benefits and HPC use cases

Historically, GPUs have primarily been used for graphics-intensive tasks like 3D modeling, rendering, or gaming. However, their highly parallelized design makes them appealing for HPC workloads, which often induce many computations that can be performed concurrently. Applications that are primarily compute-bound can benefit from significant performance improvements when executed on GPUs. Modern HPC systems have already taken advantage of GPUs by tightly incorporating them into their design. Around 98% of the peak performance of modern supercomputers such as Frontier (#1 machine on the June 2023 TOP500 ranking [9]) comes from GPUs, making it crucial to use them efficiently. Moreover, nine systems in the top 10 are equipped with GPUs, further demonstrating their importance.

The convergence between HPC and AI contributes to the hardware-accelerator trend, especially in exascale-class systems. However, it is essential to note that while both fields can benefit from GPUs, their hardware uses are entirely different. Indeed, most AI workloads can profit from reduced floating-point (FP) precision. Notably, they can leverage specialized tensor cores found in modern GPUs to

enhance the performance of AI workloads even further, which predominantly depend on dense matrix operations. This is not the case for HPC, which, most of the time, requires the use of double-precision floating-point arithmetic. As AI continues to gain significant momentum and attract a growing user base, it may impact the design of the next generation of GPUs. This influence could lead to a shift towards prioritizing more tensor cores and reduced floating-point precision, potentially at the expense of HPC's interests.

As massive reliance on hardware accelerators is becoming the norm within heterogeneous systems, it is crucial to efficiently program GPUs to exploit the performance benefits they offer correctly. To this end, the industry is investing a considerable amount of resources in software engineering to encourage and facilitate the development of GPU-accelerated applications. We are witnessing significant efforts, particularly in the field of programming languages, that concentrate on ensuring the safety and performance of massively parallel code. The Rust programming language targets those goals and will be our focus in the next section.

2.2 The Rust programming language

This section introduces the Rust programming language, its notable features, and its possible usage in HPC software that leverages hardware accelerators like GPUs.

2.2.1 Language features

Rust is a compiled, general-purpose, multi-paradigm, imperative, strong, and statically typed language designed for performance, safety and concurrency. Its development started in 2009 at Mozilla, and its first stable release was announced in 2015. As such, it is a relatively recent language that aims at becoming the new gold standard for systems programming. Its syntax is based on C and C++ but with a modern twist and heavily influenced by functional languages.

Rust's primary feature distinguishing it from other compiled languages is its principle of ownership and *borrow-checker*. Ownership rules [10] state the following:

1. Each value has an owner.
2. There can only be *one* owner at a time.
3. When the owner goes out of scope, the value's associated memory is *automatically* dropped.

Contrarily to C++, Rust is a move-by-default language. This means that instead of creating implicit deep copies of heap-allocated objects, Rust destructively moves the data between objects — i.e., any subsequent use of a moved value is detected and rejected by the compiler (see Listing 1). Furthermore, variables are constant by default and must be explicitly declared mutable using the `mut` keyword.

```
let s1 = String::from("foo");
let s2 = s1; // `s1` ownership is moved to `s2`, which now owns the value "foo"
println!("{s1}"); // ERROR! value of `s1` has been moved to `s2`
```

Listing 1: Rust's ownership in action

In Listing 1, declaring `s2` by assigning `s1` to it takes ownership of the value held by `s1` (ownership rule #2). This invalidates any later use of `s1`, and the Rust compiler can statically catch such mistakes. This guarantees that the compiled code cannot contain use-after-free bugs.

In order to share values between multiple variables, the language also provides references that implement a borrowing mechanism. There are two kinds of references in Rust:

- **Immutable (shared) references** are read-only. Multiple immutable references to the same value can exist simultaneously.
- **Mutable references** allow modifying a value that has been borrowed. However, a mutable reference is unique. There cannot be other references (mutable or shared) to a value that has been mutably borrowed while the reference remains in scope.

The Rust "borrow-checker" enforces these rules at compile-time (see Figure 18 and Figure 19 in Section 6). It can also check that any given reference remains valid while in use (i.e., the object it points to is still in scope). This statically guarantees no dangling pointers/references in the code. Listing 2 demonstrates these rules annotated with comments summing up the compiler errors, where relevant.

```
let s2;
{
    let s1 = String::from("bar");
    s2 = &s1;          // Borrowing the value held by `s1`
    println!("{s1}"); // OK! `s1` has only been borrowed; thus it is still valid
    println!("{s2}"); // OK! `s2` holds a reference to "bar" but does not own it
}
println!("{s2}");     // ERROR! `s2` held a reference that is not valid anymore
                      // because the owner `s1` went out of scope
```

```
let mut s1 = String::from("Hello, "); // `s1` needs to be declared as mutable
                                      // so we can mutably borrow it
{
    let s2 = &mut s1;      // Mutably borrowing the value held by `s1`
    let s3 = &s1;          // ERROR! cannot borrow because `s2` is a mutable
                           // reference in scope, later used to modify `s1`
    s2.push_str("world!"); // OK! Modifying `s1` through `s2`
} // `s2` falls out of scope, and the mutable reference is dropped
let s3 = &s1;              // OK! There are no mutable references to `s1`
println!("{s3}");          // Prints "Hello, world!"
```

Listing 2: Rust's borrowing in action

Rust's ownership and borrowing rules eliminate the need for a garbage collector, thus maintaining high performance comparable to other compiled languages such as C, C++ or Fortran. Moreover, they also prevent an entire class of memory safety bugs that plague C/C++ codebases, often causing crashes, memory leaks, or even opening vulnerabilities to cyber-attacks.

The advantages of these features do not stop there either. By leveraging the rules of ownership and Rust's strict type system, the compiler can catch most concurrency-related bugs, such as race conditions or data races.

Listing 3 implements a simple parallel vector sum using C++ standard library threads and a lambda function. The vector contains a million values, all initialized to `1`. Compiling the following code using the following command does not produce any warnings whatsoever:

```
$ g++ -std=c++17 -Wall -Wextra parallel_vector_sum.cpp
```

However, when running the code a few times, we get the following results:

```
$ for i in $(seq 0 5); do ./a.out; done
RESULT: 639810
RESULT: 641278
RESULT: 619719
RESULT: 1235839
RESULT: 590743
```

We obtain different results for each run and never get the expected `1,000,000` result. This is because Listing 3 contains a race condition that happens when we try to increment the `result` variable.

```cpp
constexpr size_t NELEMENTS = 1'000'000;
constexpr size_t NTHREADS = 8;
constexpr size_t QOT = NELEMENTS / NTHREADS;
constexpr size_t REM = NELEMENTS % NTHREADS;
std::vector<int> vector(NELEMENTS, 1);
std::vector<std::thread> threads(NTHREADS);

int result = 0;
for (size_t t = 0; t < NTHREADS; ++t) {
    size_t const start = t * QOT;
    size_t const end = t == NTHREADS - 1 ? start + QOT + REM : start + QOT;
    threads[t] = std::thread([&]() {
        for (size_t i = start; i < end; ++i) {
            result += vector[i];
        }
    });
}
for (auto& t: threads) {
    t.join();
}

printf("RESULT: %d\n", result);
```

Listing 3: Multi-threaded vector sum in standard C++17

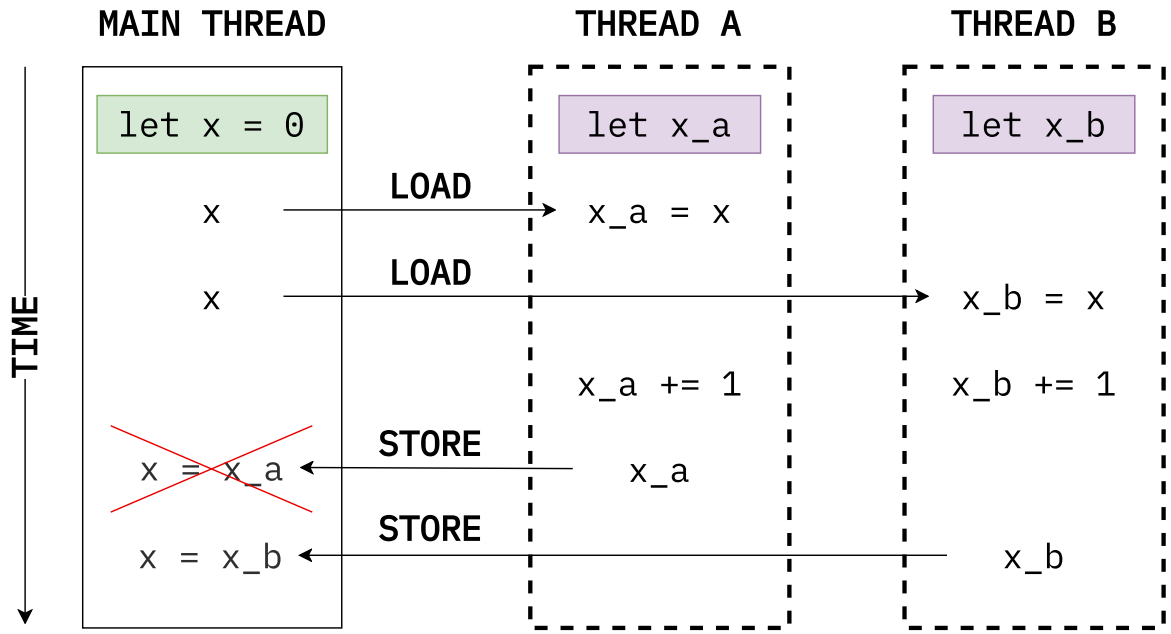

Figure 5: Illustration of a race condition

Figure 5 shows how two threads, A and B, can cause a race condition while trying to update the value of `result` concurrently. Both threads load the same value and increment it before storing it again. Thread B overwrites the value stored by thread A without considering thread A's changes, therefore losing information and producing the wrong sum.

In contrast, Rust's ownership rules allow the compiler to notice such race conditions, making it reject the following equivalent code:

```rust
const NELEMENTS: usize = 1_000_000;
const NTHREADS: usize = 8;
const QOT: usize = NELEMENTS / NTHREADS;
const REM: usize = NELEMENTS % NTHREADS;
let vector = vec![1; NELEMENTS];
let mut threads = Vec::with_capacity(NTHREADS);

let mut result = 0;
for t in 0..NTHREADS {
    let start = t * QOT;
    let end = if t == NTHREADS - 1 {
        start + QOT + REM
    } else {
        start + QOT
    };
    threads.push(std::thread::spawn(|| {
        for i in start..end {
            result += vector[i]; // thread `t` mutably borrows `result`
        }
    }));
}
for t in threads {
    t.join().unwrap();
}

println!("RESULT: {result}");
```

Listing 4: Multi-threaded vector sum in standard Rust

Indeed, in Listing 4, although Rust automatically infers that it must mutably borrow `result` in the thread's lambda (called "closures" in Rust), it cannot guarantee that the thread will finish executing before `result` goes out of scope. Furthermore, when a thread `t` mutably borrows `result`, it prevents the other threads from borrowing it, resulting in a compiler error. The entire error message is available in Listing 7 in the Section 6.
In some cases, Rust can even propose the relevant changes to make the code valid. E.g., Listing 4 can be fixed by either:

- Making the `result` variable atomic to guarantee shared-memory communication of the updated value between threads;
- Wrapping the `result` variable with a lock (e.g., a mutex) to ensure that increments of the value are protected atomically.

Rust's ownership and borrowing rules make it an excellent fit for parallel programming, as the compiler can assert that the code is semantically correct and will produce the expected behavior. Being a compiled language, it is able to match and even sometimes surpass the performance of its direct competitors, C and C++.

Hereafter, we exhaustively list other valuable features that the language includes but that are not worth exploring in detail in the available space of this report:

- Smart pointers and Resource Acquisition Is Initialization (RAII) mechanisms for automatic resource allocation and deallocation;
- Powerful enum types that can both encode meaning and hold values, paired with pattern-matching expressions to handle variants concisely;
- Optional datatypes that enable compact error handling for recoverable and unrecoverable errors;
- A generic typing system, and *traits* that provide ways to express shared behaviors between objects;
- A robust documenting, testing, and benchmarking framework integrated into the language;
- A broad standard library that provides an extensive set of containers, algorithms, OS and I/O functionalities, asynchronous and networking primitives, concurrent data structures and message passing features, etc.

Rust also comes with a vast set of tools to aid software development:

- A toolchain manager that handles various hardware targets, `rustup`;
- A package manager and build system, `cargo`;
- A registry for sharing open-source libraries, `crates.io`;
- A comprehensive language and standard library documentation, `docs.rs`;
- First-class programming tools for improved development efficiency: a language server, `rust-analyzer`, a linter `clippy`, a code formatter `rustfmt`, etc.

2.2.2 HPC use cases

Rust's accent on performance, safety and concurrency makes the language a fitting candidate for becoming a first-tier choice for HPC applications. Its focus on thread safety, in particular, empowers programmers to write fast, robust, and safe code that will be easily maintainable and improvable over its lifetime. Rust avoids many of the pitfalls of C++, especially in terms of language complexity, and its modern features and syntax make it a lot easier to work with than Fortran. Its adoption into many of the top companies that operate in fields related to HPC (Amazon, Google, Microsoft Azure, etc.), and its acceptance as the second language of the Linux kernel helped it gain a lot of traction in low-level, high-performance programming domains. Not only is it well suited to writing scientific software that relies on efficient parallelism, Rust is also a formidable language for writing HPC tools, such as profilers, debuggers, or even low-level libraries that power abstractions in higher-level languages (e.g., Python, Julia, etc.).

Rust's robust memory and thread safety features position it as an excellent candidate for GPGPU programming. Should the language's properties ensure the elimination of common bugs in parallel programming (e.g., race conditions, data races, or accesses to invalid memory regions within GPU kernels), Rust emerges as a highly attractive choice for developing the next generation of scientific applications harnessing the heterogeneous architecture of modern supercomputers.

2.3 Goals

This internship aims to establish an exhaustive state of the art for the capabilities of Rust in GPU programming. The goal is to explore what the language is currently able to support natively and what are the existing frameworks or libraries for writing GPGPU software.

As the CEA is involved in developing critical applications for simulation purposes, Rust's focus on high performance and its guarantees in type, memory and thread safety are compelling assets for writing fast, efficient, and robust code. As a primary actor in research and industry, the CEA could benefit from using Rust for hardware acceleration purposes in scientific computing. Several crates, e.g., `rayon` [11], enable trivial parallelization of code sections for CPU use cases, similar to OpenMP's ease of use in C, C++ and Fortran. This library provides parallel implementations of Rust's standard iterators that fully leverage the language thread-safety features. Code is guaranteed to be correct at compile time and unlocks the processor's maximum multi-core performance. Rayon also implements automatic work-stealing techniques that keep the CPU busy, even when the application's load balancing is not optimal. We want to investigate if something similar exists for GPU computing and, if not, to determine what the limitations would be if we tried to.
In a secondary stage, we want to assess Rust's ability to keep up with C and C++ GPGPU programming performance. This comparison would be primarily based on common compute kernels and should aim at evaluating the best options for GPU code generation in Rust.
Finally, we would like to research the limits of Rust for GPU computing by porting parts of real-world CEA applications. This work involves evaluating both the effort necessary for such ports, and the performance improvements that we can expect for industrial-grade software.

This work's ultimate purpose is to determine if it is possible to leverage Rust's properties for writing efficient code whose concurrent correctness is asserted by the compiler.

3 Contributions

This section details the work that has been conducted during the internship. We start by establishing the current state of the art for Rust's present capabilities in GPU programming. Then, we present the open-source contributions that have been made as part of the Rust-CUDA project. We continue by offering a detailed overview of a tool for profiling the performance of hardware-accelerated Rust code. Finally, we discuss the process of porting a partitioning algorithm from a CEA application on NVIDIA GPUs.

3.1 Establishing the state of the art

The first goal of the internship was to establish a comprehensive state of the art for programming GPUs with Rust. First, we investigate the state of the language's native support. Second, we look at libraries that provide capabilities for writing GPU code through shading languages or existing external frameworks. Third, we present Rust bindings to the OpenCL 3 API. Finally, we explore CUDA support specifically for NVIDIA GPUs.

3.1.1 Native language support

The Rust programming language officially supports NVIDIA's `nvptx64` architecture as a "tier 2" compiler target [12] [13]. This includes support for the following :

- Writing kernels directly in Rust
- Intrinsics for retrieving a thread's unique identifier
- Synchronization primitives for block-level scheduling

However, this initial support is minimal compared to writing standard Rust. Indeed, kernels cannot depend on Rust's standard library. Kernels must be declared as `unsafe` functions, which reduces the compiler's ability to assert the GPU code's correctness. Moreover, one of Rust's most useful abstractions, slices, are not usable inside functions compiled for the `nvptx64` target. This forces the use of raw pointers to interact with memory buffers. As pointer arithmetic is forbidden in Rust, it is necessary to use the core `add` method to correctly offset a pointer's address before de-referencing it. These shortcomings result in highly verbose kernel code, which is hard to read and write.

```rust
#![no_std]  // Disable access to the standard library
#![no_main] // Remove the requirement for a main function
#![feature(abi_ptx, core_intrinsics)] // Enable PTX ABI and access to its intrinsics
use core::arch::nvptx; // Import the `nvptx` namespace

#[no_mangle] // Prevent the compiler from mangling the function's name
// Define the function as "unsafe" and use the PTX ABI
pub unsafe extern "ptx-kernel" fn daxpy_kernel(
    n: usize, alpha: f64, x: *const f64, y: *mut f64
) -> {
    let idx = nvptx::_thread_idx_x() as usize; // Retrieve the thread's index
    if idx < n { // Assert that the index is not out of bounds
        // Get a mutable borrow of the output vector's target index/address
        let item = &mut *y.add(idx);
        // De-reference the target index/address to perform the AXPY operation
        *item += alpha * &*x.add(idx);
    }
}

// Necessary code to tell the compiler what to do in case of a fatal error
#[panic_handler]
unsafe fn breakpoint_panic_handler(_: &::core::panic::PanicInfo) -> ! {
    core::intrinsics::breakpoint();
    core::hint::unreachable_unchecked();
}
```

Listing 5: Minimal example for writing a native Rust DAXPY GPU kernel

As Listing 5 demonstrates, a kernel as simple as DAXPY is unnecessarily verbose to write. This makes GPU code exceedingly challenging to work with in native Rust due to the high amount of complexity implied by working with no language abstractions.
Furthermore, the current Rust compiler (rustc v1.72.0) cannot produce a valid executable of the above code snippet. The CUDA runtime will throw an error stating that the provided PTX assembly (NVIDIA's proprietary high-level assembly language) is invalid when trying to load it.

There is an open tracking issue for PTX code generation problems [14], but there has not been any contribution since March 2022. Rust's efforts for GPU programming native support seem to be at a stop currently.

We did not retain Rust's native support for GPU programming as a suitable approach, as it is currently unusable. Consequently, we did not conduct any performance evaluation with it.

3.1.2 Compute shaders and external libraries

Shading languages are the most popular approach for programming GPUs using the Rust language. Multiple actively maintained crates offer support for writing GPU code through compute shaders using Rust as a wrapper.
The three most relevant and active libraries are the following:

- `EmbarkStudios/rust-gpu` [15]
- `gfx-rs/wgpu` [16]
- `vulkano-rs/vulkano` [17]

Although compute shaders are a reliable way to program GPUs, they miss the point of leveraging Rust's compiler abilities to prevent a large class of parallelism-related bugs. Indeed, these libraries require the user to write the kernels using shading languages, such as GLSL/WGSL [18] or SPIR-V [19]. Utilizing a foreign language to express GPU computations prevents using Rust's strict type system and unique memory management techniques to assert that the code does not contain any use-after-free, dangling pointers or race condition kinds of bugs.

Moreover, writing scientific computing applications requires a high degree of control, especially regarding memory layout, to best optimize the code for a given target hardware. Most compute shaders lack this ability as they are primarily designed for graphics use cases (e.g., rendering, web interfaces, video games, etc.).

External C and C++ libraries, such as Arrayfire [20], also provide Rust bindings. Although the GPU code can be entirely written concisely using Rust, these bindings are too high-level for our purpose. In the case of Arrayfire, computations are expressed using an array-based notation. This makes the code much more compact but means we must rely on the library's backend code generation to do all the heavy lifting regarding optimizations.

While compute shaders are the most popular way of programming GPUs in Rust, they do not align with the uncompromising demands of HPC and scientific computing. Consequently, we did not consider benchmarking their performance, deeming them an impractical approach for our purposes.

3.1.3 OpenCL

As mentioned in Section 2.1.2.2, OpenCL is a low-level GPU programming model. Two Rust crates provide bindings to the OpenCL 3 API: `cogciprocate/ocl` and `kenba/opencl3`. Both crates feature APIs that fully leverage Rust's RAII principles and concise error handling using the `?` operator. However, kernels cannot be written directly in Rust. They must be written in OpenCL C (an extension of C99) and loaded at compile-time into the Rust code, either via a macro or by directly pasting the kernel as a string into the Rust program. Similarly to the compute shaders and external libraries presented in the previous section, this prevents the Rust compiler from guaranteeing GPU kernels' type, memory, and thread safety. Although this appears limiting for our purpose, it is easier to integrate Rust code in an existing HPC code base that uses OpenCL as their hardware-accelerator programming language (e.g., for code portability reasons).

In the rest of this subsection, we will assume the use of the `cogciprocate/ocl` crate [21].

The `ocl` library provides all the necessary abstractions to call functions from the OpenCL API concisely. It can manage platforms, devices, programs and queues, kernels, memory allocations on the GPU, and data transfers between the host and the device. This can be expressed in highly succinct code, thanks to Rust's elegant syntax for handling errors and automatic resource deallocation.

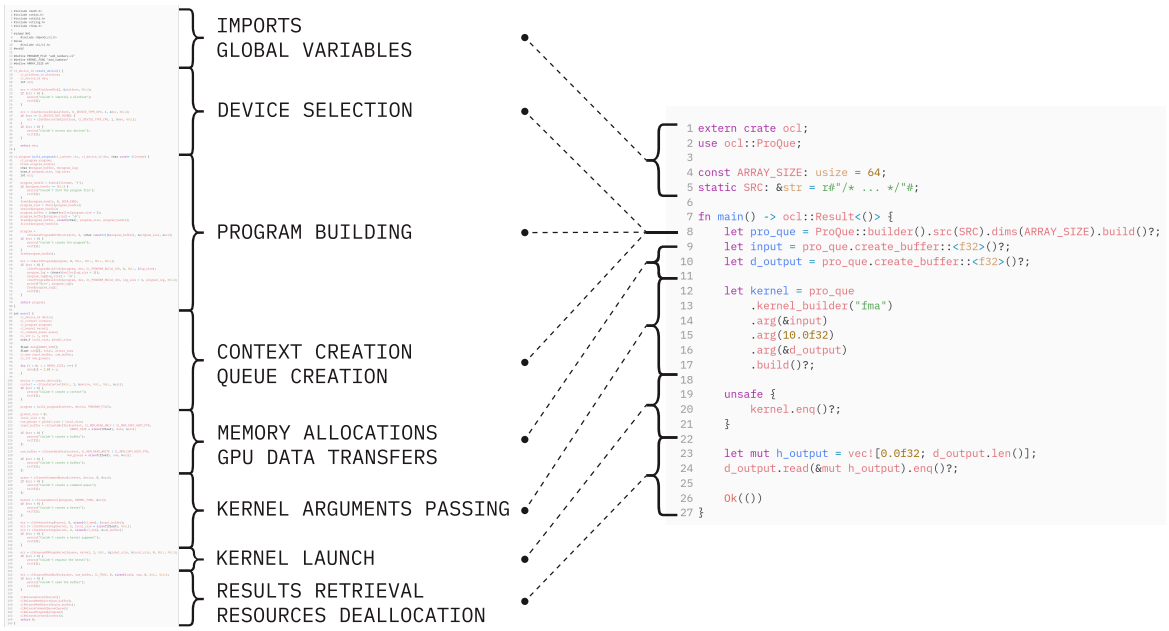

Figure 6: C vs. Rust comparison of minimal code example for launching a kernel on a GPU.

Figure 6 demonstrates how much more compact it is to write OpenCL using Rust as a "frontend" rather than C or C++. In this example, the original C code is 165 lines long. Although it correctly handles all possible errors, it only frees the allocated resources at the end of the program, which can lead to memory leaks in case of an early caused by an error. In contrast, the Rust is only 27 lines long. All the error handling and resource deallocation logic is tightly packaged through the use of the `?` operator on each `ocl` function call. If a given call returns an error, the stack is automatically unwinded to free allocated memory before returning the error to the callee. The complete code for both OpenCL versions can be found in the Section 6 at Listing 8 and Listing 9, respectively.

Since OpenCL already occupies an essential role in hardware-accelerator programming for HPC, owing to its emphasis on fine-grained control and cross-vendor portability, we selected it as a viable option for Rust-based GPU programming. As such, we conducted performance evaluations on OpenCL as part of the later stages of the internship, which we present in a subsequent section of this report.

3.1.4 CUDA

As introduced in Section 2.1.2.2, CUDA is a low-level, proprietary GPU programming model designed specifically for NVIDIA hardware accelerators. While CUDA is C++-based, most of its internals are language-agnostic and solely work based on PTX (Parallel-Thread eXecution) [22] and/or cubin (CUDA binary) files. PTX is NVIDIA's proprietary low-level, human-readable ISA (Instruction Set Architecture), and the penultimate state of a kernel's representation before being lowered to SASS (Streaming ASSembler) format and ultimately turned into a cubin file. Consequently, this means that we are not bound to use C++ for writing GPU kernels and that it is possible to utilize Rust instead, as long as we are able to compile it into PTX code.

The `Rust-GPU/Rust-CUDA` open-source project [23] tries to do that by offering Rust first-class CUDA programming capabilities instead of C++. It consists of a complete software stack, providing code generation targeting NVIDIA GPUs, management of the CUDA environment, and bindings to most NVIDIA libraries aimed at HPC/AI workloads. Although it is limited to CUDA hardware, the Rust-CUDA project is, at the moment of writing, the most advanced way of natively programming GPUs in Rust.

The `Rust-CUDA` project comprises multiple sub-projects, some of which are independent from the others. In the remainder of this section, we will present the most relevant ones for using Rust-CUDA in an HPC environment.

`cust` acts as the Rust equivalent of the CUDA C++ Runtime library. It provides all the basic tools to manage the environment surrounding GPU code execution, e.g., creating streams, allocating device-side buffers, handling data transfers between CPU and GPU memory, launching kernels, etc. In order to improve control over contexts, modules, streams, and overall performance, `cust` is implemented using bindings to the CUDA Driver API. This actually comes as a requirement, as Rust-CUDA kernels that have been compiled into PTX or cubin/fatbin files must be dynamically loaded as modules at runtime, which are only supported in the Driver API. `cust` can be used independently of the other sub-projects described here and currently is the only library that can launch CUDA kernels from Rust (i.e., it is a required dependency for executing GPU code written using the Rust's compiler native support, as discussed in Section 3.1.1). Moreover, `cust` can also be used to launch kernels written in CUDA C++, as long as they are a PTX or cubin/fatbin module that can be loaded at runtime, as described previously.

`cuda-std` is the GPU-side "standard" library for writing Rust-CUDA kernels. It provides all of the usual CUDA functions and primitives (e.g., getting a thread's index, synchronizing threads at the block level, etc.) and also a wide variety of low-level intrinsics for math functions, warp-level manipulations, or address-space casting. `cuda-std` also provides macros for allocating shared memory, which we can extensively use for kernel performance optimizations.

`rustc-codegen-nvvm` is a custom backend for the Rust compiler that produces PTX code and the most crucial component of the Rust-CUDA project for enabling Rust as a first-class CUDA programming language. It leverages NVIDIA's libNVVM to offload the code generation and most of the optimization work. The NVVM IR (Intermediate Representation) is a proprietary compiler internal representation based, at the time of writing, on LLVM 7 IR. The `rustc-codegen-nvvm` module is responsible for generating valid PTX from Rust's inner Mid-level Intermediate Representation (MIR). It first lowers MIR to LLVM 7 IR, then feeds it into libNVVM before getting the final, optimized PTX. Figure 7 showcases the complete compilation process for generating a cubin/fatbin from a Rust-CUDA kernel using `rustc-codegen-nvvm`.

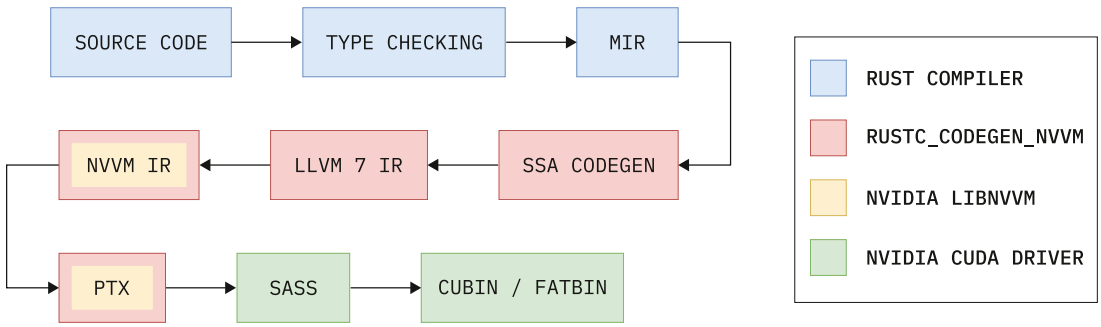

Figure 7: Complete compilation pipeline of a Rust-CUDA kernel

`ptx-compiler` is a small tool that allows the compilation of PTX files into cubin or fatbin files. This allows us to avoid JIT compilation of PTX upon loading it as a module when using `cust`.

`cuda-builder` is another small tool that allows us to build our Rust-CUDA kernels into PTX or cubin/fatbin files in `build.rs` scripts, thus helping to automatize the build process GPU code in Rust projects.

Unfortunately, the Rust-CUDA project has not been maintained since the end of 2021. As the Rust compiler is constantly evolving, with new releases every six weeks, the current version (v.1.72.0 at the time of writing) is now incompatible with Rust-CUDA. Likewise, the latest version of CUDA is also incompatible due to breaking changes in the NVVM library used by `rustc-codegen-nvvm`. However, this has been mitigated as part of the internship, as explained in Section 3.2.

It is essential to mention that Rust-CUDA is not a project officially endorsed by Rust or NVIDIA. It is purely an open-source piece of work developed by a computer science student who does not have the time nor the will to continue maintaining it. Moreover, from his point of view, this project should be integrated upstream, directly as part of the `rustc` compiler, to really gain traction and benefit from more contributions. Ideally, it could replace the current backend implementation for PTX code generation as it is more complete and should offer better performance, thanks to the use of NVIDIA's proprietary NVVM IR. Some bindings to HPC libraries (cuBLAS, cuSPARSE, cuFFT) are incomplete and could probably be improved by using more idiomatic Rust wrappers.

While Rust-CUDA only aims at GPU programming on NVIDIA hardware, it is the advanced and complete way of writing kernels using native Rust syntax. As a result, we selected it for subsequent benchmarking experiments carried out later in this internship.

3.2 Open-source work on the Rust-CUDA project

To support the latest major release of the NVIDIA CUDA Toolkit (version 12), we had to fix the breaking changes introduced between this version and the last stable version with which Rust-CUDA was compatible, CUDA 11.8. As mentioned in Section 3.1.4, the Rust-CUDA project uses a custom compiler backend, `rustc-codegen-nvvm`, which depends on NVIDIA's libNVVM. As part of the CUDA 12 Toolkit update, a new version (v2.0) of the NVVM IR specification [24] was released.

NVVM 2.0 introduced the following breaking changes:

- Removed address space conversion intrinsics.
- Stricter error checking on the supported data layouts.
- Older style loop unroll pragma metadata on loop back edges is no longer supported.
- Shared variable initialization with non-undefined values is no longer supported.

CUDA 12 also adds support for Hopper and Ada Lovelace architecture while dropping support for Kepler and deprecating Maxwell architectures.

We forked Rust-CUDA and fixed the breaking changes presented above, as well as updated the minimum architecture requirements for using the project. We also added support for NVIDIA's newest architectures: Hopper (HPC/AI/server-focused) and Ada Lovelace (consumer-targeted). As part of this endeavor, we took the time to enhance some of the project's documentation. We also added improved code examples that leverage more advanced, previously undocumented, features of `cuda-std`. These include using shared memory and tiling programming techniques applied within an optimized General Matrix Multiply (GEMM) kernel.

We are currently working on a draft Pull Request (PR) to merge these changes into the upstream Rust-CUDA to benefit more people and hopefully kickstart a resumption of the project's maintenance.

3.3 Hardware-Accelerated Rust Profiling

After establishing an exhaustive state of the art for GPU programming in Rust, we chose the most relevant code generation methods and set out to benchmark their performance. To do this, we implemented an open-source tool that evaluates the performance of GPU-accelerated Rust. HARP (Hardware-Accelerate Rust Profiling) is a CEA project hosted at the CEA-HPC organization on GitHub.

3.3.1 Implementation details

HARP is a simple profiler for evaluating the performance of hardware-accelerated Rust code. It aims at gauging the capabilities of Rust as a first-class language for GPGPU programming, especially in the field of scientific computing.

HARP can benchmark and profile the following set of kernels:

- AXPY (general vector-vector addition), of complexity $\mathcal{O}(n)$

$$y = \alpha x + y$$

- GEMM (general dense matrix-matrix multiplication), of complexity $\mathcal{O}(n^3)$

$$C = \alpha AB + \beta C$$

- Reduction (sum reduction), of complexity $\mathcal{O}(\log n)$ in parallel, $\mathcal{O}(n)$ otherwise

$$r = \sum_{i=0}^{n} x_i$$

- Prefix Sum (sum exclusive scan), of complexity $\mathcal{O}(\log n)$ if the processor (CPU or GPU) has at least n cores, $\mathcal{O}(n \log n)$ otherwise

$$\forall x_i \in x, x_i = \sum_{j=0}^{i-1} x_j$$

Please note that the Rust-CUDA implementation of the scan kernel currently does not work for unknown reasons. It appears to be caused by a memory issue (segmentation fault), but the identical CUDA C++ code works flawlessly. We suppose it is caused by a problem during the Rust-CUDA code generation step, maybe ABI-related, but we could not find a fix at the time of writing.

The AXPY and GEMM kernels were chosen because they are part of the traditional Basic Linear Algebra Software (BLAS) [25] routines. Measuring the performance of BLAS kernels is crucial in HPC applications as it enables the optimization of fundamental mathematical operations, which are prevalent in scientific computing workloads. BLAS performance benchmarking helps identify bottlenecks, improve computational efficiency, and optimize hardware utilization, particularly on specialized architectures such as GPUs and multi-core CPUs. Ultimately, they aid in algorithm selection, benchmarking HPC systems, assessing scalability, and achieving energy-efficient computations. We have paid the utmost attention to optimizing the GPU-based GEMM kernels, using a myriad of advanced optimization techniques, such as shared memory, tiling, and instruction-level parallelism, based on the ClBlast OpenCL BLAS library [26].
The reduction and scan kernels are fundamental building blocks for all sorts of algorithms and were needed to implement the RCB algorithm presented in Section 3.4.2. We took this opportunity to include them in the set of benchmarks provided by HARP. Similarly to what we have done for the BLAS kernels, we took care of optimizing their implementation using state-of-the-art GPGPU programming techniques [27] [28] [29].

Each of the kernels is available in several implementations:

- CPU: sequential naive (using C-style `for` loops), sequential idiomatic (using iterators constructs), and parallel (using the `rayon` crate);
- OpenCL;
- CUDA, using either Rust-CUDA or CUDA C++ code.

The CPU versions of the kernels serve as a baseline to compare the speedup GPUs offer. The GPU implementations of the GEMM kernel are available in two flavors: a naive version and an optimized one that leverages shared memory with SIMD memory loads and stores, as well as tiling techniques to use the underlying hardware architecture more efficiently.

Profiling can be done on both single-precision and double-precision floating-point formats (following IEEE 754 norm [30]). Currently, both the reduction and scan kernels only support 32-bit signed integers. This is due to time constraints that prevented the implementation of generic versions for floating-point arithmetic, which is more intricate to set up when using advanced warp-level intrinsic. The algorithmic results are validated with an accuracy requirement: a tolerance of 10^{-15} for double-precision implementations and 10^{-6} for single-precision counterparts.

The user must specify a kernel to benchmark and a set of dimensions on which to run the measurements (vector length for AXPY, reduction and scan, matrix size for GEMM). HARP then automatically performs all the benchmarking runs and generates a CSV file containing a report of the aggregated statistics for the kernel. A report includes the following information for each dimension specified in the HARP benchmark configuration:

- The target kind (either host or device);
- The implementation variant of the kernel;
- The number of elements per dimension;
- The allocates memory size in bytes;
- The total number of FP operations.

It also includes the following metrics about the kernel:

- The minimum and maximum recorded execution time;
- The median and mean (average) recorded execution time;
- The runtime standard deviation;
- The arithmetic intensity (in FLOP/Byte);
- The memory bandwidth (in GiB/s);
- The computational performance (in GFLOP/s).

HARP also provides a Python script that produces graphs from the performance reports using pandas and plotly libraries. It takes the CSV output from HARP and can be configured to output PNG images of the generated plots.

3.3.2 Benchmark methodology

In order to assert the stability and correctness of the measures, we developed a systematic approach to benchmarking the kernel implementations. Listing 6 gives a high-level overview of the algorithmic methodology used to measure the performance of Rust kernels.

The input dataset is randomly initialized for each specified dimension and remains invariant for all of the benchmark runs for that specific dimension. This guarantees that we do not fall into edge cases where the compiler or the CPU/GPU microarchitecture can aggressively optimize some computations (e.g., when doing operations with 1s or 0s). It also ensures that all implementations and their respective variants are compared using a consistent dataset.

The `MIN_REP_COUNT` constant allows us to repeat the measurements as many times as necessary to compute meaningful statistics about the kernel's performance. The default value is set to 31 (the same value used by the MAQAO HPC profiler [31]).

The `MIN_EXEC_TIME` constant serves a tight loop that ensures that the kernels run for a long enough period of time. This value depends on the clock's precision used to benchmark the kernels.

```
PROGRAM harp_benchmark
──────────────────────────────────────────────────────────────────────────────
INPUTS:
  kernel:          A kernel to benchmark
  implementations: A list of implementations to compare
  variants:        A list of variants for each implementation
  datatype:        The datatype to use
  dimensions:      A list of dimensions for generating the datasets
  rng_seed:        A seed for a randomized dataset generation
──────────────────────────────────────────────────────────────────────────────
OUTPUT:
  A list of statistics for each dimension/implementation/variant combination
  of the benchmarked kernel
──────────────────────────────────────────────────────────────────────────────
CONSTANTS:
  MIN_EXEC_TIME: Minimum execution time to validate a kernel execution
  MIN_REP_COUNT: Minimum number of benchmarks to perform for a given
                 dimension/implementation/variant combination
──────────────────────────────────────────────────────────────────────────────
VARIABLES:
  dataset:   A list of randomly generated values for each dimension
  samples:   A list of execution times for each
             dimension/implementation/variant combination
  exec_time: Execution time of a given kernel
             dimension/implementation/variant combination
──────────────────────────────────────────────────────────────────────────────
PROCEDURE:
FOR EACH dim IN dimensions
  dataset <- generate_dataset(datatype, dim, rng_seed)

  FOR EACH impl IN implementations
    FOR EACH var IN variants
      FOR EACH i IN [0, MIN_REP_COUNT]
        WHILE exec_time < MIN_EXEC_TIME
          exec_time <- chrono(kernel(impl, var, dataset))
        END WHILE
        samples[dim, impl, var, i] <- exec_time
      END FOR EACH
    END FOR EACH
  END FOR EACH
END FOR EACH

RETURN compute_statistics(samples)
```

Listing 6: Pseudo-code of the algorithm used to benchmark kernels in HARP

3.3.3 Results analysis

This subsection will present the results obtained from HARP and separate measurements designed to specifically compare GPU programming in Rust against other hardware-accelerator paradigms or libraries in the field of HPC.
All performance results presented hereafter were done on a workstation with the following characteristics:

- **CPU:** Intel (Alder Lake) i5-12600H, 12 cores (4 hyper-threaded P-cores, 8 E-cores) @ 4.5 GHz, 32 GB DDR5 RAM.
- **GPU:** NVIDIA T600 (Turing), 640 CUDA cores, 4 GB GDDR6 VRAM, 160 GB/s memory bandwith, 1.7 TFLOP/s computational performance in FP32.
- **Software stack:** NVIDIA GPU Driver v535.86.10, NVIDIA CUDA Toolkit Version 12.2, NVIDIA OpenCL SDK Version 12.2, NVIDIA HPC SDK Version 23.7.
- **Compilers**: `gcc` v11.4 and v13.1, `rustc` v1.59.0 and v1.72.0, `nvcc` v12.2.

In this subsection, we will focus on presenting results for the DGEMM kernel, which is the most relevant one in the context of HPC. In Section 6.3, we include the full plot outputs for all kernels available in HARP.

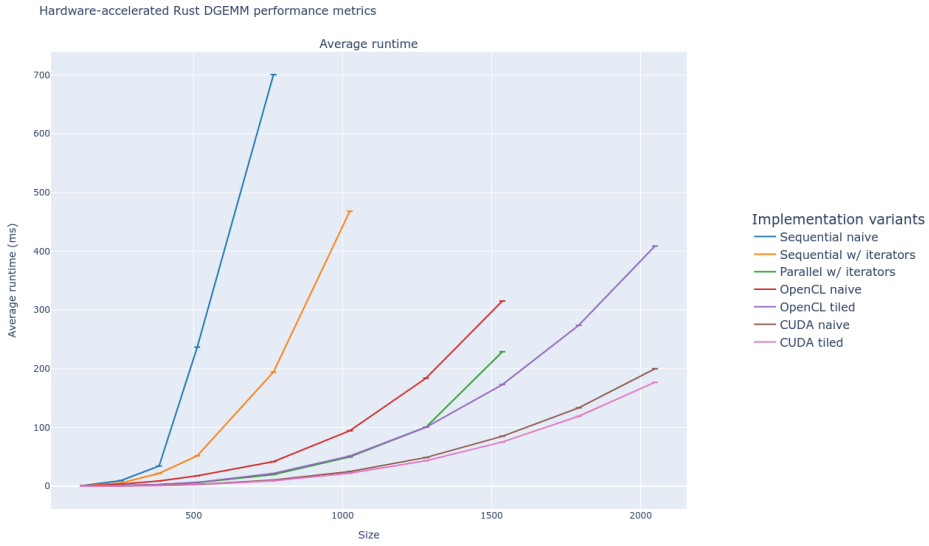

Figure 8: Average runtime performance for the DGEMM kernel

Figure 8 compares the average runtime of multiple Rust implementations of the DGEMM kernel (CPU and GPU) for increasing sizes of FP64 precision dense matrices. The graph includes error bars for each measurement obtained following the algorithm described in Listing 6. This graph clearly shows the performance dominance of hardware accelerators over traditional CPUs. Both sequential implementations (naive uses C-style `for` loops, the other uses Rust's iterator constructs) are at least twice as slow as the fastest GPU version on matrices that are twice as small. Both CUDA-based DGEMMs significantly outshine the OpenCL implementations.

Figure 9 presents the computational performance for each kernel implementation using the same results as Figure 8. We can interpret this graph as the opposite of the previous one, with higher FLOP/s indicating increased performance. This plot gives a better visualization of the performance difference between implementations, with CUDA-based ones clearly ahead of OpenCL, with over 2x better performance. We can also notice that the GPU kernels are compute-bound; their performance continually increases until it reaches a plateau and stays constant regardless of the size of the matrix. On the other hand, the results of CPU implementations decrease as the matrixes get bigger, highlighting the fact that these kernels are memory-bound.

Figure 9: Computational performance for the DGEMM kernel

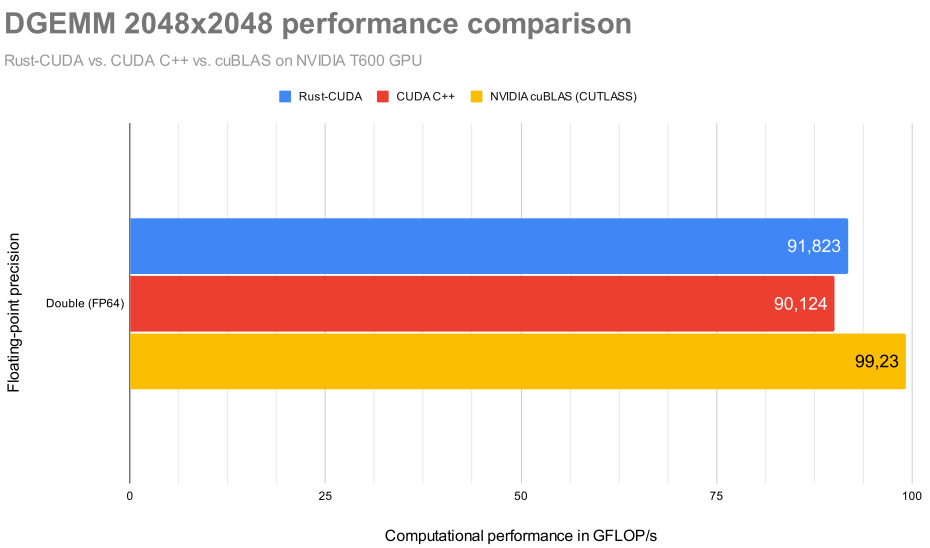

Figure 10: Performance comparison between DGEMM kernels on different CUDA-based implementations

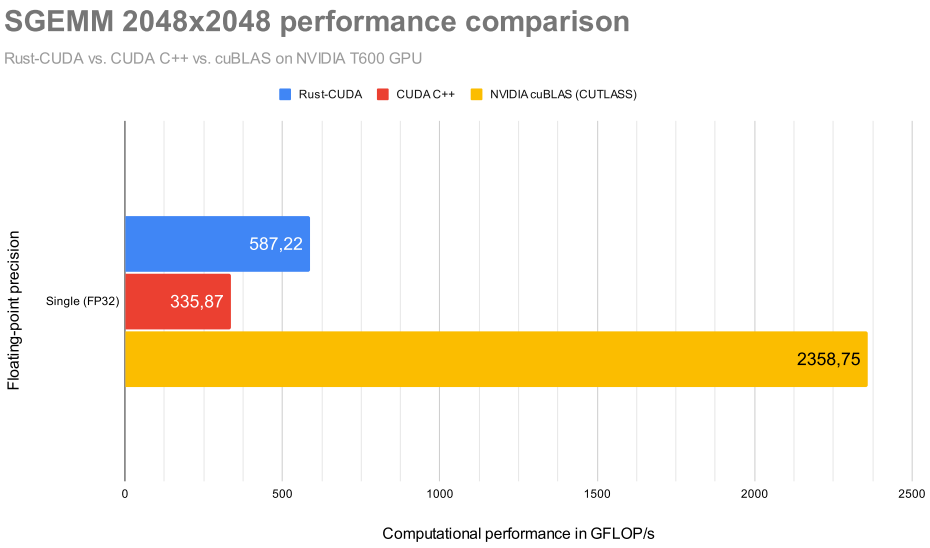

Figure 11: Performance comparison between DGEMM kernels on different CUDA-based implementations

Figure 10 and Figure 11 compare the computational performance of Rust-CUDA, CUDA C++ and cuBLAS implementations of the GEMM kernel in both single and double FP precision on an NVIDIA T600 GPU. The results presented are for matrices of size 2048, initialized with random values between 0 and 1 and non-null and non-one values for the α and β coefficients. The same benchmarking methodology used in HARP has been applied here, and the standard deviation of these results is under 5%. In double-precision floating-point, Rust-CUDA kernels performed slightly better than the CUDA C++ implementation (a 1:1 equivalent). Both of these manual implementations are behind the NVIDIA cuBLAS one by about 10%, which is a relatively small performance drop considering how highly optimized NVIDIA's libraries are. As both the Rust-CUDA and CUDA C++ kernels share the same code generation pipeline (NVVM IR $\rightarrow$ PTX, see Figure 7), Figure 10 demonstrates that the Rust compiler front-end can match, and even slightly edge, the C++ one in terms of optimizations made at the IR level.

Figure 11 highlights this even better, with the Rust-CUDA implementation achieving a massive 75% performance improvement over the CUDA C++ SGEMM kernel. However, this result seems overly pessimistic of the FLOP/s we expect from a CUDA C++ implementation. Historically, C and C++ compilers used to convert every floating-point operation to double precision, even if only single precision was required. Some of the arithmetic operations in the CUDA C++ implementation of the SGEMM kernel may be performed in FP64, which would explain the reduced performance. At the time of writing, we could not assert that this is what is actually happening. We are investigating at the binary level by analyzing the generated assembly (PTX and SASS) to confirm it.

On the other hand, Figure 11 shows that the cuBLAS implementation is largely out of reach when dealing with single-precision floating-point arithmetic. This is explained by the cuBLAS implementation using the GPU's hardware more efficiently than our hand-written kernels, notably through extensive reliance on tensor cores. These specialized cores are dedicated to accelerating matrix operations using a non-IEEE754 format (TensorFloat32, or TF32), which only uses 10 bits for the mantissa and is optimized to provide up to 8x speedups over standard FP32 precision. Although this affected the result slightly, it still achieved the 10^{-6} required accuracy to validate the benchmark. This matches the performance increase over our CUDA C++ implementation, which is roughly seven times slower than the cuBLAS one.

We also had the opportunity to compare Rust-CUDA and Kokkos (C++) GEMM kernels and obtained comparable performance between implementations.

3.4 Porting partitioning algorithms from a CEA application

The final stage of the internship involves porting parts of a real-world application to the GPU using Rust. This last step aims to push the boundaries of Rust GPGPU programming capabilities and explore the limits of the compiler's help in writing thread-safe kernels. Porting is done using the Rust-CUDA project, targeting NVIDIA GPUs.

3.4.1 `coupe`, a concurrent mesh partitioner

The application we chose to port is `coupe` [32], a modular, multi-threaded library for mesh partitioning written in Rust. It is developed at the CEA/DAM by the joint CEA — Paris-Saclay University LIHPC laboratory. Coupe implements multiple algorithms aimed at achieving optimal load balancing while minimizing communication costs through the use of geometric methods.
Hereafter, we list some of the partitioning algorithms available in the tool, some of which offer optimized variants for cartesian meshes:

- Space-filling curves: Z-curve, Hilbert curve
- Recursive Coordinate Bisection (RCB) and Recursive Inertial Bisection (RIB)
- Multi-jagged
- Karmarkar-Karp
- K-means

3.4.2 Recursive Coordinate Bisection (RCB)

The algorithm we have chosen to port is the Recursive Coordinate Bisection (RCB) [33] [34], one of the simplest geometric algorithms.

Given an N-dimensional set of points, select a vector n of the canonical basis $(e_0, ..., e_{n-1})$. Split the set of points with a hyperplane orthogonal to n, such that the two parts of the splits are evenly weighted. Recurse as many times as necessary by reapplying the algorithm to the two parts with another normal vector in each.

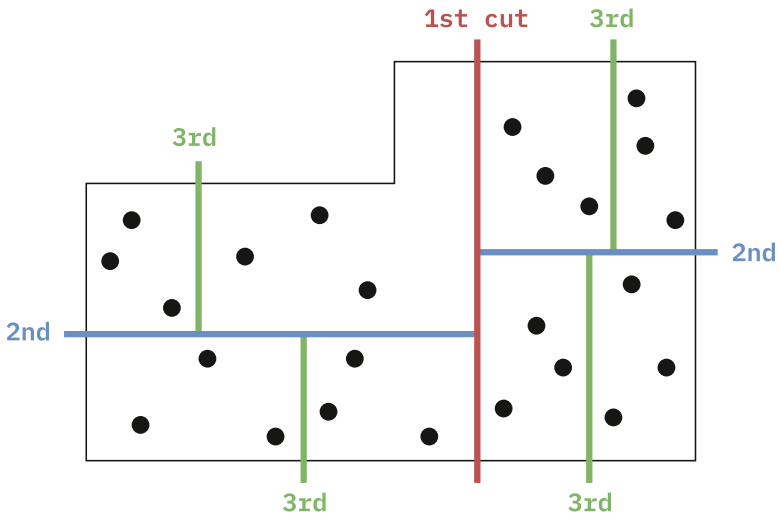

Figure 12: 3-step RCB algorithm visualized on a random set of points

Figure 12 showcases the partitioning of a set of points following the RCB algorithm. We have chosen it because of its straightforward approach and recursive nature, making it ideal for GPU use.

3.4.3 Observations

In practice, trying to port the RCB algorithm was exceedingly difficult. Indeed, most of it relies on a sequence of fundamental algorithms, such as reductions or prefix sums (exclusive scan). Because Rust-CUDA does not have access to library bindings that provide those primitive GPU building blocks, this implied that we had to rewrite everything ourselves. This endeavor proved very time-consuming and challenging to do efficiently by hand. We encountered multiple issues with kernel code generation producing invalid PTX, which meant we had to switch to CUDA C++ implementations instead of Rust-CUDA ones.

In summary, the Rust-CUDA project serves as a robust foundation for crafting “basic” GPU code using Rust. However, it is not currently equipped to handle more complex hardware-accelerated programming tasks, as it lacks many useful abstractions and bindings to libraries that hasten the development of optimized GPU kernels for scientific computing applications.
We were not able to achieve a working Rust-CUDA port of the RCB algorithm at the time of writing. However, we are investigating a different approach that should simplify the GPU implementation while also better exploiting the architecture of the hardware accelerator. This work will be our main focus for the remainder of the internship.

4 Conclusion

This internship aimed to explore Rust's viability for GPGPU programming in scientific computing use cases. On the one hand, HPC is shifting towards an ever-growing reliance on hardware accelerators, especially GPUs, and it is crucial to write robust code that efficiently exploits these new heterogeneous architectures. On the other hand, the Rust programming language focuses on performance, safety and concurrency, three aspects that align well with HPC's needs. Rust's type, memory, and thread safety features make it a particularly appealing language for this task.

Without any prior comprehensive work on Rust's potential for GPU programming, we set out to establish an exhaustive state of the art for the available options in this domain. We investigated various possibilities, first starting with the language's native support. Then, we explored compute shaders libraries that brought hardware accelerator programming capabilities to Rust. We concluded that the most relevant strategies for developing GPGPU kernels involved using bindings to the OpenCL API and the Rust-CUDA project, designed to bring first-class support for NVIDIA's CUDA framework within the Rust ecosystem.
During this internship, we also contributed to the Rust-CUDA project by updating the code generation pipeline to support the newly released CUDA 12 Toolkit. We also added support for the most recent NVIDIA architectures to optimize the generated PTX targeting them.

The next step of this internship was to develop a scientific approach to evaluate the performance of Rust-based GPU programming. To this end, we developed an open-source tool, HARP, which automates the performance benchmarking of basic kernels often encountered in scientific applications. HARP was carefully designed to provide accurate and reliable results and remain portable across a wide variety of systems, from small laptops to bleeding-edge supercomputers.

Finally, we pushed the boundaries of Rust-based GPU programming by attempting to port parts of an industrial-grade scientific library for mesh partitioning. This proved to be remarkably challenging as we reached the limits of the Rust-CUDA project. Numerous constraints constitute critical opportunities for improving Rust's support in GPU programming.

The main challenge for Rust's adoption remains the low amount of contributions in projects that focus on HPC and scientific computing. This is reinforced by the absence of libraries that provide the basic blocks for writing more complex and well-optimized algorithms on hardware accelerators (e.g., NVIDIA CUB and NVIDIA Thrust in the CUDA ecosystem).

However, while the prospect of developing an entire application leveraging GPUs in Rust may currently seem challenging, the language exhibits several promising features that could turn into serious assets in the future. Rust already largely outclasses C and C++ for orchestrating the environment surrounding GPU execution. This includes the management of devices, streams, memory allocations and transfers, kernel launches, and more. Rust's distinctive memory management approach significantly alleviates the challenges posed by error handling and resource deallocation in C/C++. These often result in a plethora of elusive memory bugs, which can be arduous to trace and rectify. The OpenCL API bindings and the `cust` crate within the Rust-CUDA project already constitute a significantly better alternative than their default C/C++ counterparts.

Furthermore, as we demonstrated through several benchmarks presented in this report, Rust is capable of rivaling and even exceeding the performance of C++. Although kernels written using Rust-CUDA still lack some of the usual abstractions traditionally offered by Rust, they are more compact than their CUDA C++ equivalents. Given that Rust-CUDA is still in its early stages of development, it is easy to envision a future where Rust is the most accessible, efficient, and robust choice for GPU programming.

4.1 Perspectives and future work

The final weeks of this internship will focus on finishing to implement the GPU version of the RCB algorithm discussed in Section 3.4.2. We will also get the chance to conduct more benchmarks on an NVIDIA A100 GPU, which we hope to present during the thesis defense.

Beyond the scope of this internship, Rust has several challenges to address if it aims to become a proper first-class language for GPU programming. Its primary goal should be to improve the native language support, at least make it a usable alternative to Rust-CUDA. The Rust project leadership could also consider integrating the Rust-CUDA project into the upstream rustc compiler so as to benefit from the advanced work that has already been done and improve upon it. Likewise, the focus should be on enhancing the existing abstractions, particularly for manipulating slices, as well as fixing some of the overly verbose syntax required to write kernels.

Moreover, the serious lack of compute-focused libraries for GPU programming should be tackled as soon as possible to make Rust a more viable language for writing kernels.
There are two possible approaches we can already propose:

1. Write bindings for existing C/C++ libraries that already fit this purpose, e.g., NVIDIA CUB and NVIDIA Thrust.
2. Implement an idiomatic wrapper over Rust-based kernels that provide the same functionality as Rust's native standard library, notably for handling iterators. This should allow for writing kernels that look and feel very much like usual CPU code while also benefiting from highly optimized GPU primitives that are abstracted away thanks to "syntax sugar."

Another point that could be explored is Rust's support and performance on AMD hardware accelerators. This internship has been predominantly focused on NVIDIA GPUs, as it was the only hardware we had available. Future work could investigate the performance of OpenCL Rust on AMD, especially compared to ROCm/HIP C++. Likewise, we should examine the viability of a Rust-ROCm project, similar to what we have with Rust-CUDA, and even consider a direct integration into the Rust compiler.

5. Bibliography

[1] "NVIDIA h100 tensor core GPU architecture overview." https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper (accessed: Aug. 28, 2023).

[2] "Inside volta: the world's most advanced data center GPU." https://developer.nvidia.com/blog/inside-volta/ (accessed: Sep. 1, 2023).

[3] "Contents — cuda-toolkit-release-notes 12.2 documentation." https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/contents.html (accessed: Aug. 31, 2023).

[4] "HIP/docs/reference/terms.md at develop · ROCm-developer-tools/HIP." https://github.com/ROCm-Developer-Tools/HIP/blob/develop/docs/reference/terms.md (accessed: Aug. 21, 2023).

[5] "OpenCL - the open standard for parallel programming of heterogeneous systems." https://www.khronos.org// (accessed: Aug. 31, 2023).

[6] "SYCL - c++ single-source heterogeneous programming for acceleration offload." https://www.khronos.org/sycl/ (accessed: Aug. 22, 2023).

[7] "Home - OpenMP." https://www.openmp.org/ (accessed: Sep. 1, 2023).

[8] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: enabling manycore performance portability through polymorphic memory access patterns ," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, 2014, doi: https://doi.org/10.1016/j.jpdc.2014.07.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[9] "June 2023 \textbar TOP500." https://www.top500.org/lists/top500/2023/06/ (accessed: Sep. 1, 2023).

[10] "What is ownership? - the rust programming language." https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (accessed: Aug. 24, 2023).

[11] "Rayon," rayon-rs, 2023. Accessed: Sep. 1, 2023. [Online]. Available: https://github.com/rayon-rs/rayon

[12] "Target tier policy - the rustc book." https://doc.rust-lang.org/rustc/target-tier-policy.html#tier-2-target-policy (accessed: Aug. 27, 2023).

[13] "Nvptx64-nvidia-cuda - the rustc book." https://doc.rust-lang.org/rustc/platform-support/nvptx64-nvidia-cuda.html (accessed: Aug. 27, 2023).

[14] "NVPTX target specification by denzp · pull request #57937 · rust-lang/rust." https://github.com/rust-lang/rust/pull/57937 (accessed: Aug. 27, 2023).

[15] "🐉 rust-gpu," Embark, 2023. Accessed: Aug. 31, 2023. [Online]. Available: https://github.com/EmbarkStudios/rust-gpu

[16] "WebGPU." https://www.w3.org/TR/2023/WD-webgpu-20230726/ (accessed: Aug. 31, 2023).

[17] "Vulkano," vulkano-rs, 2023. Accessed: Aug. 28, 2023. [Online]. Available: https://github.com/vulkano-rs/vulkano

[18] "KhronosGroup/glslang: khronos-reference front end for GLSL/ESSL, partial front end for HLSL, and a SPIR-v generator. https://github.com/KhronosGroup/glslang (accessed: Aug. 28, 2023).

[19] "SPIR - the industry open standard intermediate language for parallel compute and graphics." https://www.khronos.org// (accessed: Aug. 31, 2023).

[20] "ArrayFire: overview." https://arrayfire.org/docs/index.htm#gsc.tab=0 (accessed: Aug. 28, 2023).

[21] “Ocl,” Cogciprocate, 2023. Accessed: Aug. 28, 2023. [Online]. Available: https://github.com/cogciprocate/ocl

[22] “PTX and SASS assembly debugging.” https://docs.nvidia.com/gameworks/content/developertools/desktop/ptx_sass_assembly_debugging.htm (accessed: Aug. 29, 2023).

[23] “Rust-GPU/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: Aug. 31, 2023).

[24] “NVVM IR spec.” https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html#revision-history (accessed: Aug. 28, 2023).

[25] “BLAS (basic linear algebra subprograms).” https://netlib.org/blas/ (accessed: Sep. 1, 2023).

[26] C. Nugteren, “CLBlast: the tuned OpenCL BLAS library,” 2023. Accessed: Aug. 31, 2023. [Online]. Available: https://github.com/CNugteren/CLBlast

[27] “Chapter 39. parallel prefix sum (scan) with CUDA.” https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda (accessed: Aug. 31, 2023).

[28] “Faster parallel reductions on kepler.” https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/ (accessed: Aug. 31, 2023).

[29] M. Harris, “Optimizing parallel reduction in CUDA.”

[30] “Ieee standard for floating-point arithmetic,” *IEEE Std 754-2019 (Revision IEEE 754-2008)*, vol. 0, no. , pp. 1–84, 2019, doi: 10.1109/IEEESTD.2019.8766229.

[31] “MAQAO.” https://www.maqao.org/ (accessed: Sep. 1, 2023).

[32] “Coupe,” LIHPC-Computational-Geometry, 2023. Accessed: Aug. 31, 2023. [Online]. Available: https://github.com/LIHPC-Computational-Geometry/coupe

[33] Berger, and Bokhari, “A partitioning strategy for nonuniform problems on multiprocessors,” *IEEE Trans. Comput.*, no. 5, pp. 570–580, May 1987, doi: 10.1109/TC.1987.1676942.

[34] B. Bramas, “A novel hybrid quicksort algorithm vectorized using AVX-512 on intel skylake,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 8, no. 10, 2017, doi: 10.14569/IJACSA.2017.081044. Accessed: Aug. 31, 2023. [Online]. Available: http://arxiv.org/abs/1704.08579

6 Appendix

6.1 Glossary

Items in the glossary are ordered alphabetically.

ABI: Application Binary Interface
AI: Artificial Intelligence
API: Application Programming Interface
ASIC: Application-Specific Integrated Circuit
AXPY: $\alpha x + y$, vector addition
BLAS: Basic Linear Algebra Software
CU: Compute Unit
CUDA: Compute Unified Device Architecture
(G)DDR: (Graphics) Double Data Rate (memory)
DNN: Deep Neural Network
FMA: Fused Multiply-Add
FP: Floating Point
FPGA: Field Programmable Gate Array
FFT: Fast Fourier Transform
GEMM: General Matrix Multiplication
(GP)GPU: (General Purpose) Graphics Programming Unit
HBM: High-Bandwidth Memory
HIP: Heterogeneous-Compute Interface for Portability
HPC: High Performance Computing
I/O: Input/Output
IR: Intermediate Representation
ISA: Instruction Set Architecture
JIT: Just-In-Time (compilation)
MMA: Matrix Multiply-Accumulate
OpenCL: Open Computing Language
OpenMP: Open Multi-Processing
OS: Operating System
PCIe: Peripheral Component Interconnect Express
PTX: Parallel-Thread eXecution
RAII: Resource Acquisition Is Initialization
(V)RAM: (Video) Random Access Memory
RCB: Recursive Coordinate Bisection
RDMA: Remote Direct Memory Access
RIB: Recursive Inertial Bisection
ROCm: Radeon Open Compute
SASS: Streaming ASSembler
SIMD: Single Instruction, Multiple Data
SIMT: Single Instruction, Multiple Thread
SM: Streaming Multiprocessor
TF: Tensor Float

6.2 Listings

```
error[E0373]: closure may outlive the current function, but it borrows `result`,
which is owned by
the current function
  --> src/thread_safety.rs:18:41
   |
18 |         threads.push(std::thread::spawn(|| {
   |                                         ^^ may outlive borrowed value `result`
19 |             for i in start..end {
20 |                 result += array[i];
   |                 ------ `result` is borrowed here
   |
note: function requires argument type to outlive `'static`
  --> src/thread_safety.rs:18:22
   |
18 |           threads.push(std::thread::spawn(|| {
   |  ______________________^
19 | |             for i in start..end {
20 | |                 result += array[i];
21 | |             }
22 | |         }));
   | |_________^
help: to force the closure to take ownership of `result` (and any other referenced
variables), use
the `move` keyword
   |
18 |         threads.push(std::thread::spawn(move || {
   |                                          ++++

error[E0499]: cannot borrow `result` as mutable more than once at a time
  --> src/thread_safety.rs:18:41
   |
18 |           threads.push(std::thread::spawn(|| {
   |                        -                   ^^ `result` was mutably borrowed here
in the previous iteration of the loop
   |  ______________________|
   | |
19 | |             for i in start..end {
20 | |                 result += array[i];
   | |                 ------ borrows occur due to use of `result` in closure
21 | |             }
22 | |         }));
   | |_________- argument requires that `result` is borrowed for `'static`
```

Listing 7: Rust's compiler error message for a race condition bug in

Listing 8: Minimal OpenCL C code that builds and run an OpenCL DAXPY kernel on a GPU

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#ifdef MAC
    #include <OpenCL/cl.h>
#else
    #include <CL/cl.h>
#endif

#define PROGRAM_FILE "add_numbers.cl"
#define KERNEL_FUNC "add_numbers"
#define ARRAY_SIZE 64

cl_device_id create_device() {
    cl_platform_id platform;
    cl_device_id dev;
    int err;

    err = clGetPlatformIDs(1, &platform, NULL);
    if (err < 0) {
        perror("Couldn't identify a platform");
        exit(1);
    }

    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &dev, NULL);
    if (err == CL_DEVICE_NOT_FOUND) {
        err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &dev, NULL);
    }
    if (err < 0) {
        perror("Couldn't access any devices");
        exit(1);
    }

    return dev;
}

cl_program build_program(cl_context ctx, cl_device_id dev, char const* filename) {
    cl_program program;
    FILE* program_handle;
    char *program_buffer, *program_log;
    size_t program_size, log_size;
    int err;

    program_handle = fopen(filename, "r");
    if (program_handle == NULL) {
        perror("Couldn't find the program file");
        exit(1);
    }
    fseek(program_handle, 0, SEEK_END);
    program_size = ftell(program_handle);
    rewind(program_handle);
    program_buffer = (char*)malloc(program_size + 1);
    program_buffer[program_size] = '\0';
    fread(program_buffer, sizeof(char), program_size, program_handle);
    fclose(program_handle);
```

```
    program =
                clCreateProgramWithSource(ctx,  1,  (char  const**)(&program_buffer),
&program_size, &err);
    if (err < 0) {
        perror("Couldn't create the program");
        exit(1);
    }
    free(program_buffer);


    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    if (err < 0) {
      clGetProgramBuildInfo(program, dev, CL_PROGRAM_BUILD_LOG, 0, NULL, &log_size);
        program_log = (char*)(malloc(log_size + 1));
        program_log[log_size] = '\0';
          clGetProgramBuildInfo(program, dev, CL_PROGRAM_BUILD_LOG, log_size + 1,
program_log, NULL);
        printf("%s\n", program_log);
        free(program_log);
        exit(1);
    }

    return program;
}

int main() {
    cl_device_id device;
    cl_context context;
    cl_program program;
    cl_kernel kernel;
    cl_command_queue queue;
    cl_int i, j, err;
    size_t local_size, global_size;

    float data[ARRAY_SIZE];
    float sum[2], total, actual_sum;
    cl_mem input_buffer, sum_buffer;
    cl_int num_groups;

    for (i = 0; i < ARRAY_SIZE; i++) {
        data[i] = 1.0f * i;
    }

    device = create_device();
    context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
    if (err < 0) {
        perror("Couldn't create a context");
        exit(1);
    }

    program = build_program(context, device, PROGRAM_FILE);

    global_size = 8; // WHY ONLY 8?
    local_size = 4;
    num_groups = global_size / local_size;
   input_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                 ARRAY_SIZE * sizeof(float), data, &err);
    if (err < 0) {
        perror("Couldn't create a buffer");
        exit(1);
    };
```

```
    sum_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                                num_groups * sizeof(float), sum, &err);
    if (err < 0) {
        perror("Couldn't create a buffer");
        exit(1);
    };

    queue = clCreateCommandQueue(context, device, 0, &err);
    if (err < 0) {
        perror("Couldn't create a command queue");
        exit(1);
    };

    kernel = clCreateKernel(program, KERNEL_FUNC, &err);
    if (err < 0) {
        perror("Couldn't create a kernel");
        exit(1);
    };

    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_buffer);
    err |= clSetKernelArg(kernel, 1, local_size * sizeof(float), NULL);
    err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &sum_buffer);
    if (err < 0) {
        perror("Couldn't create a kernel argument");
        exit(1);
    }

    err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size, &local_size,
0, NULL, NULL);
    if (err < 0) {
        perror("Couldn't enqueue the kernel");
        exit(1);
    }

    err = clEnqueueReadBuffer(queue, sum_buffer, CL_TRUE, 0, sizeof(sum), sum, 0,
NULL, NULL);
    if (err < 0) {
        perror("Couldn't read the buffer");
        exit(1);
    }

    clReleaseKernel(kernel);
    clReleaseMemObject(sum_buffer);
    clReleaseMemObject(input_buffer);
    clReleaseCommandQueue(queue);
    clReleaseProgram(program);
    clReleaseContext(context);
    return 0;
}
```

Listing 9: Minimal OpenCL Rust code for building and launching an OpenCL DAXPY kernel on a GPU, using the `ocl` crate

```
extern crate ocl;
use ocl::ProQue;

const ARRAY_SIZE: usize = 64;
static SRC: &str = r#"
__kernel void fma(__global float const* input, float scalar, __global float* output)
{
    int idx = get_global_id(0);
    output[idx] += scalar * output[idx];
}
"#;

fn main() -> ocl::Result<()> {
    let pro_que = ProQue::builder().src(SRC).dims(ARRAY_SIZE).build()?;
    let input = pro_que.create_buffer::<f32>()?;
    let d_output = pro_que.create_buffer::<f32>()?;

    let kernel = pro_que
        .kernel_builder("fma")
        .arg(&input)
        .arg(10.0f32)
        .arg(&d_output)
        .build()?;

    unsafe {
        kernel.enq()?;
    }

    let mut h_output = vec![0.0f32; d_output.len()];
    d_output.read(&mut h_output).enq()?;

    Ok(())
}
```

6.3 Figures

Figure 13: Complete HARP output for the SAXPY kernel benchmark

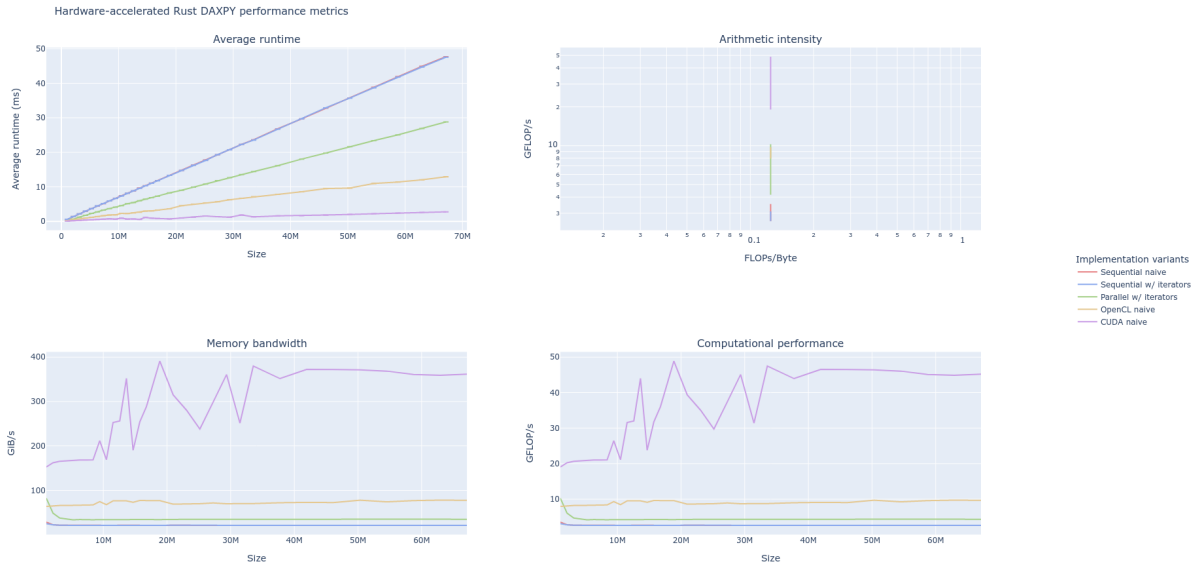

Figure 14: Complete HARP output for the DAXPY kernel benchmark

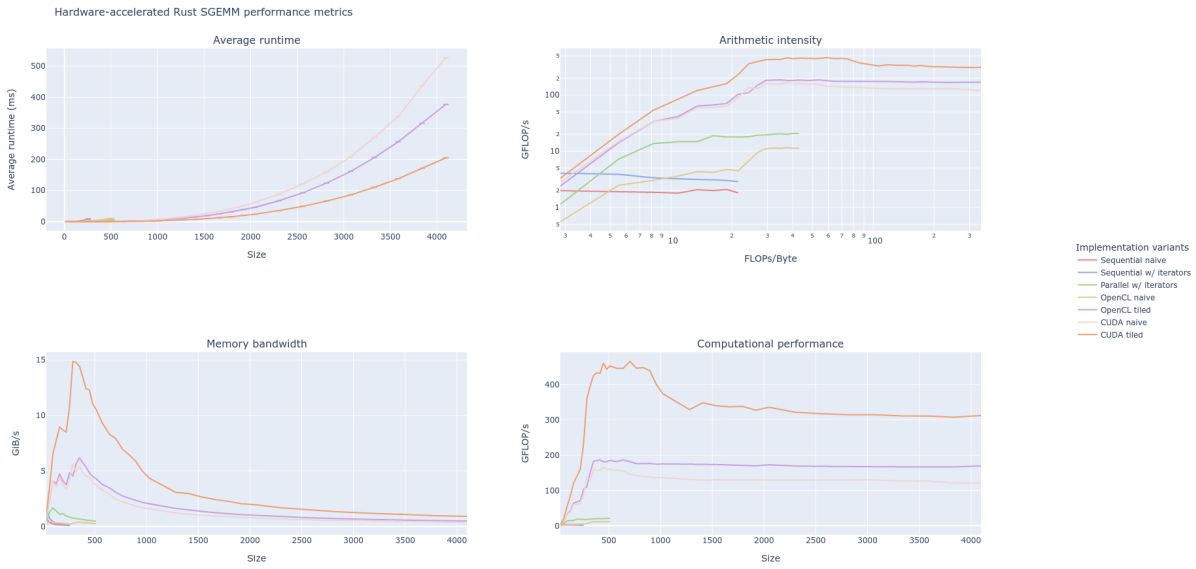

Figure 15: Complete HARP output for the SGEMM kernel benchmark

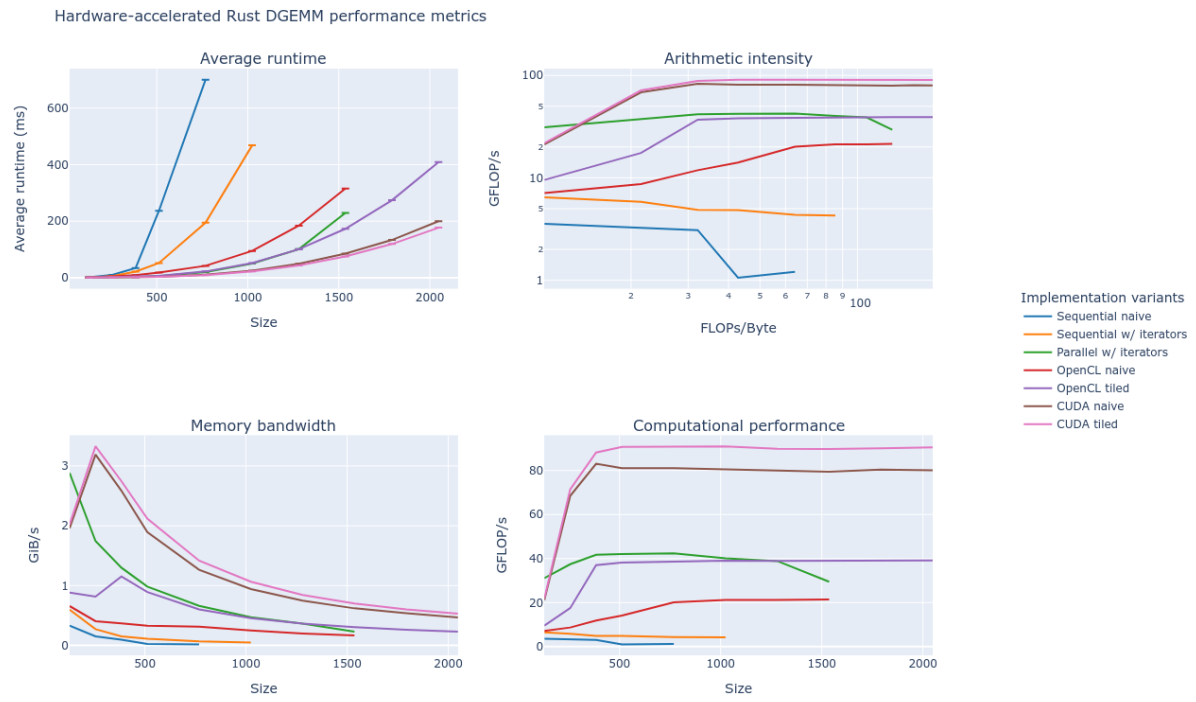

Figure 16: Complete HARP output for the DGEMM kernel benchmark

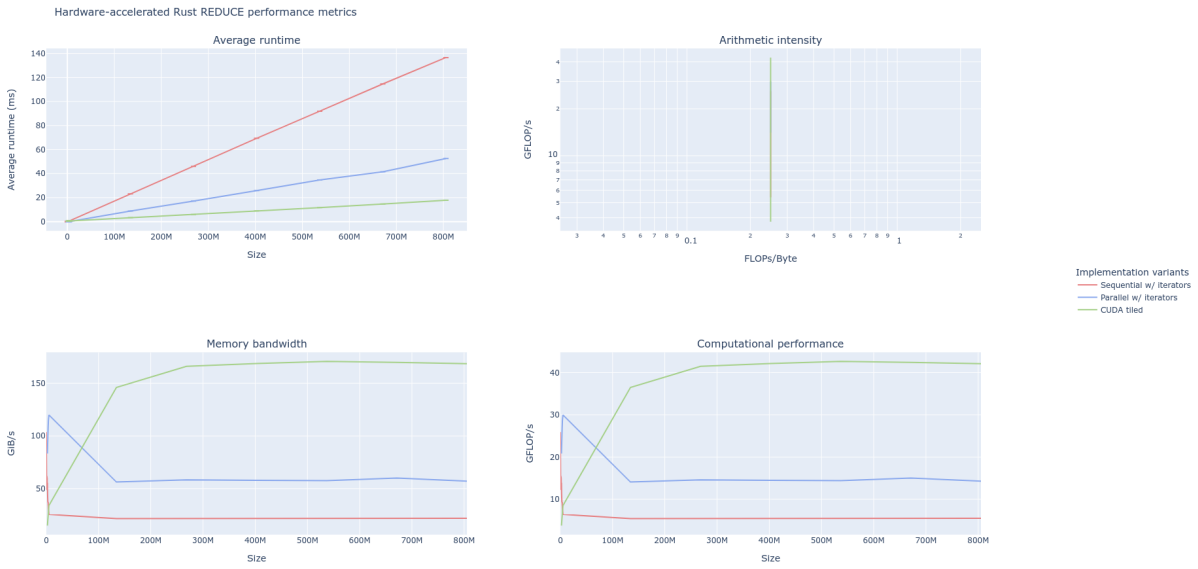

Figure 17: Complete HARP output for the reduction kernel benchmark

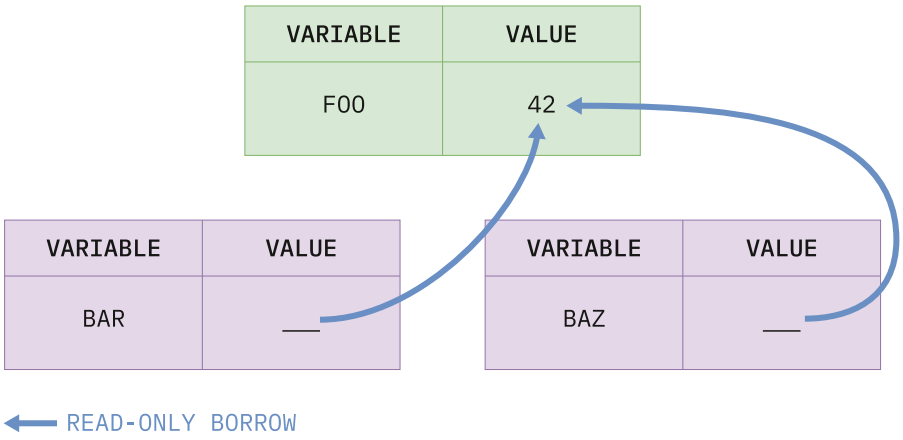

Figure 18: Read-only borrowing visualization

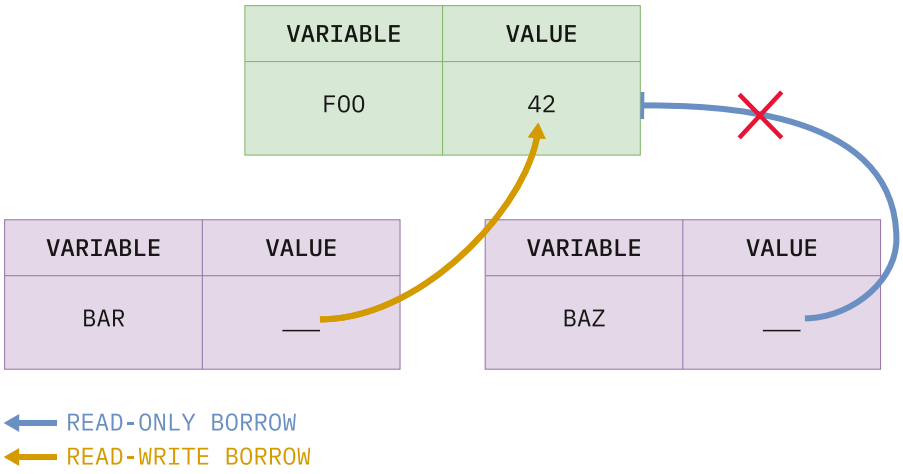

Figure 19: Read-write borrowing visualization