Politecnico di Torino

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Politecnico di Torino
Universitat Politècnica de Catalunya

Master's Degree in Computer Engineering
A.Y. 2022/2023
October 2023 Graduation Session

Evaluation of the Parallel Features of Rust for Space Systems

Supervisors:
Alessandro Savino
Leonidas Kosmidis

Candidate:
Alberto Perugini

Abstract

The rise in complexity of the algorithms run on space systems, largely attributable to higher resolution instruments which generate a large amount of the data to be processed, as well as to the need for increased autonomy which relies on Neural Network inference systems in future missions, demand the adoption of more powerful on-board hardware, such as multicores.

At the same time, the correctness and reliability of critical on-board software is of paramount importance for the success of the missions. However, developing such complex software in low-level languages can have a negative impact on these aspects.

For this reason, this thesis evaluates the role that the Rust programming language can have in this change, given its memory safety and built in support for parallelism, which allows to better utilise more powerful hardware, in particular multicore cpus, without compromising programmability and safety of the code.

To this end, the GPU4S benchmarking suite, part of the open source OBPMark benchmarking suite of the European Space Agency (ESA), is ported to Rust, with sequential and parallel implementations. The applications are ported both in a hosted as well as in bare metal environment for a RISC-V based platform for the space domain, developed within the METASAT Horizon Europe project. The performance of the ported benchmarks is compared to the existing sequential and parallel implementations in low-level languages to evaluate the trade-offs of the different solutions, and it is evaluated on several multicore platforms which are candidates for future on-board processing systems. A particular focus is put on parallel versions of the benchmarks, where Rust offers solid native support, as well as library support for fast parallelization similar to OpenMP. Finally, in terms of correctness, the Rust implementations are free of recently detected defects in the low-level implementations of the GPU4S benchmarks.

Acknowledgements

Thanks to my family and my friends for supporting my during my studies. Without them I would not have gotten to where I am today.

This work has been performed at the Computer Architecture / Operating Systems (CAOS) group of the Barcelona Supercomputing Center (BSC). It has been supported by the European Space Agency (ESA) through the "Formal Methods for GPU Software Development and Verification" project with Contract No. ESA STAR AO 2-1856/22/NL/GLC/ov, as well as by the European Commission, through the METASAT Horizon Europe Project with grant agreement number 101082622.

Contents

Acronyms

API Application Programming Interface.

CLINT Core Local Interrupt.

CPU Central Processing Unit.

CSR Control and Status Register.

DRAM Dynamic Random Access Memory.

ELF Executable and Linkable Format.

ESA European Space Agency.

FFT Fast Fourier Transform.

FIR Finite Impulse Response (Filter).

FPGA Field Programmable Gate Array.

GCC GNU C Compiler.

GPU4S GPUs for Space.

GPU Graphics Processing Unit.

HPC High Performance Computing.

IPI Inter-Process Interrupt.

ISA Instruction Set Architecture.

ISR Interrupt Service Routine.

LPDDR Low Power Double Data Rate (DRAM).

LRN Local Response Normalization.

MM Matrix Multiplication.

OBPMark On-Board Processing Benchmarks.

QEMU Quick EMUlator.

RAII Resource Acquisition Is Initialization.

RELU REctified Linear Unit.

RFC Request For Comment.

RTEMS Real-Time Executive for Multiprocessor Systems.

UART Universal Asynchronous Receiver/Transmitter.

List of Listings

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The development of more powerful application for safety critical missions, together with the growth in the number of cores in commercial processors, requires the software to adapt to use more efficiently the newer platforms. C/C++ are have been the only languages used in the field, but they are older languages that lack in more modern features and in particular in safety. Memory safety becomes even more relevant in parallel application, as it is easier to make mistakes in this context. More modern and safer programming languages such as Java and Go, offer better programmability in particular in parallel code, however they are not usable in safety critical systems due to the fact that the garbage collector can cause big latency spikes, and does not allow for *Worst Case Execution Time* calculations.

In this niche domain, the Rust programming language has a real chance of being the best option. It is a memory safe language without garbage collection, which gets rid of latency spikes [45]. Instead of garbage collection, it uses the RAII (Resource Acquisition Is Initialization) paradigm [29] to insure that memory is freed. It offers modern tools and features, and has native parallelism support. Performance is also one of the biggest focuses of Rust, which is important to compete with C and C++.

In this thesis we analyse if the potential that Rust has in theory is carried out in practice, by porting the GPU4S benchmarks [33] [15]. These benchmarks, while they will be useful for platform performance evaluation, in the context of this thesis are used instead to show a performance and programmability comparison between Rust and C, both in sequential and parallel code.

The code developed is released as open source at [25].

1.2 Thesis organization

This thesis is organised in 7 chapters: Chapter 2 and 3 serve as an introduction to the GPU4S benchmarks and the Rust programming language respectively. Chapter 4 presents the hosted (i.e. running on an operating system) version of the benchmarks, both in their sequential and parallel versions. Chapter 5 introduces the benchmark code designed to run on bare metal, in particular on a RISC-V platform. Chapter 6 discusses the platforms on which the code was evaluated, as well as the performance comparisons and finally Chapter 7 presents the conclusions and future work.

Chapter 2

The GPU4S benchmarks

In this Chapter we introduce the GPU4S benchmarks suite [15] [33], also known as OBPMark Kernels, part of the open source OBPMark benchmarking suite [40][41] of the European Space Agency (ESA) hosted at [4], which are the applications we ported to Rust in this thesis. Next we describe their purpose, design and delve into the benchmarks functionality and relevance.

2.1 Introduction and motivation

As the amount of data to be processed on board of spacecraft increases and the type of algorithms to be run expand to allow autonomous operation, the need for performance in the embedded system employed is growing significantly. The GPU4S Bench suite was developed to improve the performance testing capabilities of such systems, with a particular focus on Embedded Graphics Processing Units (GPUs) for satellite on-board processing and other safety critical systems, where existing benchmarks were particularly inadequate.

2.2 The benchmark suite

The complete list of the benchmarks available in the suite is:

- CIFAR 10
- Convolution
- Correlation
- Fast fourier transform
- Fast fourier transform window

- Finite impulse response filter
- LRN
- Matrix multiplication
- Max pooling
- Memory bandwidth
- Relu
- Softmax
- Wavelet transform

The benchmarks were chosen as a representative subset of typical algorithms run in space relevant applications, with a look also at applications that will become important in the future, such as Computer Vision and Neural Networks.

Each of the benchmarks is designed to use different datatypes, in particular single and double precision floating point and integer numbers, and are developed using different programming models:

- A sequential cpu version written in C/C++.
- A parallel cpu version that uses OpenMP for parallelization.
- Two portable GPU versions, using OpenCL and HIP.
- A CUDA version, meant to be run on NVIDIA GPUs
- Porting to other programming models has been performed as part of related Bachelor's and Master Theses, such as Ada SPARK for CPU and GPU [1][2] and SYCL [24][23][22]. Their merging in the mainline GPU4S Bench repository is pending on the following release, and will also include the sequential and parallel implementations of Rust from this thesis [25].

One important functionality of the benchmarks is the ability to validate their result using a reference implementation in C, that is provided in the suite itself. This is important when porting the benchmark to be able to validate the code, as in our case. Below we describe some of the benchmarks to highlight their relevance and functionality.

2.2.1 Fast Fourier transform

The Fast Fourier Transform (FFT) is a numerical approximation for the Fourier Transform, and aims at calculating the contribution of different frequencies to a signal. It is manly used in telecommunications applications.

2.2.2 Convolution

The matrix convolution uses a kernel to transform each point in a matrix to be a linear combination of itself and its neighbours. It is widely used in image processing and machine learning, as it decreases computational needs by only considering the neighbours of a pixel rather than the whole image.

2.2.3 Matrix multiplication

Matrix multiplication is one of the most used numerical methods, useful for all kinds of applications, from solving linear systems to machine learning. It is one of the most optimized algorithms, especially when in GPUs, thanks to its regularity.

2.2.4 Max pooling

The max pooling function is widely used in Neural Networks, in particular along with convolutional layers. It reduces the dimensionality of the input by returning a matrix where each element is the maximum in its subsection of the input matrix. The size of the subsection side is usually called **stride**.

2.2.5 Softmax

Softmax is also typically used in Neural Networks. Similar to a simple max function, where the max element becomes a one and all the others zeroes, in this case an exponential function is used to distribute weights in a "softer" way. Each output element is calculated using the formula:

$$x_i{}' = \frac{e^{x_i}}{\sum_{j=0}^{K} e^{x_j}}$$

2.2.6 CIFAR 10

The CIFAR 10 benchmark simulates the inference of an image on a Neural Network trained on the CIFAR 10 dataset, which is a collection of images divided among ten different classes. It uses functions from other benchmarks (e.g. Matrix Multiplication, Convolution, Softmax etc.) to simulate either one or multiple inferences, using random weights and inputs.

2.3 Benchmark's functionality

The benchmarks offer both compile time and run time configuration.

At compile time the user can select the data type to be used and select the implementation to be run amongst those mentioned before; the Makefile deals with compiling the correct files and exporting the relevant flags so that the C compiler can select the appropriate configuration. This approach reduces the size of the executable by only linking the relevant code.

At run time the user can select options through command line arguments. Here we give a list of the behaviour common to all benchmarks:

- Specify the size of the matrix (or matrices).
- Verify the output against the reference CPU implementation.
- Print the time it took to run the benchmark function, in different formats.
- Possibility to read the input from file. The default behaviour is to generate random matrices in memory, which makes these benchmarks independent from operating system support such as filesystem access. This enables their use in bare metal environments, as we did for the RISC-V platform.
- Possibility to export the result to file.
- Select which GPU to use if several ones are available. This option is only available for the GPU versions of the benchmarks.

2.4 Porting methodology

When porting the benchmark, we want to keep the same functionality as the original benchmarks, while when possible increasing ease of use. To this end, the code is divided in two packages: one for the actual benchmarks to be run on the platform being tested, which allows for the same functionalities as the original code, making it easy to compare results between the different implementations; one that serves as a library for the benchmarks, so that we can offer higher generality, to support potential further developments. For example the library code, that contains the implementation algorithms, will not be specialized for square matrices, but rather will accept any shape of matrix; the benchmark code will then make sure to pass square matrices, so that the results are comparable.

To avoid increasing complexity too much, however, some functions remain similar in behaviour to their C counterparts: as an example, the convolution function is implemented to work only with a specific kind of padding, and to have as output a matrix with the same size of the one in input, which is not quite general but it is sufficient for benchmarking purposes.

Chapter 3

Rust

The Rust programming language is designed for safety and parallel programming, which makes it interesting for safety critical applications where there is a need for high performance systems. This chapter provides an overview of its syntax, features, data types and some libraries that we will use in the thesis work. The contents of the chapter are heavily inspired by the explanations in *The Rust Book* [42], which is an introductory guide that covers much of Rust's features.

3.1 Important components

Being a modern programming language, Rust offers features that are uncommon in legacy programming languages such as C/C++. In this section we discuss some utilities that Rust offers to make our life easier.

Rustup

The first tool we introduce is **rustup**, which is a utility that helps us getting started, by making the installation process very easy, and it provides an command line interface for installing functionalities and updating Rust. To get started then we can download the installer from `https://rustup.rs`, which will install the toolchain for the current platform, as well as **cargo**, the Rust packet manager, which we will discuss in detail later.

Once the installation is done, we can for the most part forget about rustup, though it will come in handy if we want to install the toolchain for another target, which is useful for cross-compiling for another platform, if we want to use a some features that are not yet available in the stable release, or when we want to update the version of Rust we are using.

`rustc`

rustc is the Rust compiler. As we will see, we almost never want to call rustc directly, as we can use `cargo` for most of our needs. In the course of the thesis we indeed never used it directly, though we passed flags to it through cargo.

Cargo

Cargo is usually referred to as Rust's packet manager, because it helps dealing with dependencies by pulling the specified version of the packages (called **crate**s in Rust) from `crates.io`, which is the Rust package registry. This type of tool is typical in more modern programming languages such as Python.

Cargo however covers a lot more roles than just a packet manager, as it also deals with much of the compiling and running interface of a Rust program. It offers a simpler compiling interface than rustc as it is aware of the packet structure, since it is defined in the `Cargo.toml` file which is included in the base directory of all Rust projects.

```
[package]
name = "package name"
version = "0.1.0"
edition = "2021"

[dependencies]
clap = { version = "4.1.8", features = ["derive"] }
rayon = "1.7.0"

[features]
default = []

feature1 = []
```

Listing 3.1: Example of a `Cargo.toml` file.

Another important feature of Cargo is the `features` compiler flags. This can be defined in the `Cargo.toml` file and allow for having different versions of a crate depending on the feature flags that are passed, which also allows for conditional linking and compiling.

3.2 Primitive data types

Rust has more data types than most other programming languages, in particular when it comes to numeric types, which provides more information to the compiler, making catching some errors easier. For example, if we try to run the code in Listing 3.2 we get an error telling us that `the literal ‘1000‘ does not fit into the type ‘u8‘ whose range is ‘0..=255‘`.

```
let mut num: u8 = 0;
...
num = 1000;
```

Listing 3.2: Example of illegal assignment in Rust.

Obviously not all these types of errors can be known at compile time, however when building in debug mode the overflow will get caught at run time. On the other hand, in release mode this is not the case; this is for performance considerations, since adding the check after each operation can lead to significantly slower code, but, differently from C[13], the overflow will not lead to undefined behaviour, whereas in C it is is the case due to compiler optimizations [21]. The default behaviour in both languages is for the result to *wrap around*, meaning that if `a` and `b` are of type `int-n`, and the maximum and minimum values representable in type `int-n` are respectively `MAX` and `MIN`, the result of $a + b$ will be $((a + b)$ **mod** `MAX`$) +$ `MIN`. This is assured to be the always the case in Rust, but not in C.

The fact that overflow results in the program stopping execution if the program is compiled in debug mode, also serves as a clear indication that implicit wrapping is not acceptable behaviour in Rust: instead the type `Wrapping` is the correct way to deal with intentional overflow, without any run time cost since it just serves to tell the compiler to not introduce the overflow checks it would otherwise put when compiling for debug purposes.

It should be noted that while the numeric type can be specified, requiring the programmer to decide a specific type for any variable would be quite onerous, but thankfully Rust has a powerful type inference engine that allows us to not specify the type of variables in most cases.

A list of Rust's primitive data types follows:

- **Signed integer**: `i8`, `i16`, `i32`, `i64` and `i128`.
- **Unsigned integer**: `u8`, `u16`, `u32`, `u64` and `u128`.
- **Integer type aliases**: `usize` and `isize`, their number of bits depend on the specific architecture and are usually used for addressing arrays and similar structures.
- **Floating point**: `f32` and `f64` for respectively single and double precision.
- **Character**: `char`, it is not a single byte, but rather a UTF-32 4 bytes value
- **String**: `&str`, the string type in Rust is complex so we will explore it further in another section.
- **Boolean**: `bool` with values `true` and `false`.
- **Array**: `[T; N]`, where `T` is the type of the array content and `N` is the number of elements.

- **Slice**: `[T]`, a contiguous view onto some array like structure, the number of elements can be unknown at compile time.
- **Tuples**: `(T, U, ...)` a sequence of elements of potentially different types

3.3 Enums, Structs and user defined types

There are two different kinds of user defined types in Rust, `Enums`, `Structs`.

Structs

`Structs`, just as in `C`, group named properties related to an istance of some sort. They are also the equivalent of `Classes` in object oriented languages such as Java, with 2 main differences: when defining a `struct`, only the data fields are specified, while methods are defined in separate `impl` blocks; a single struct can have zero, one, or multiple `impl` blocks. Another big difference compared to OOP languages like `Java`, is that there is no inheritance; this will be explained further when talking about `Traits`.

```
pub struct BasicStructure {
    field1: i32,
    field2: UserDefinedType
}

impl BasicStructure {
    pub fn new(field1: i32, field2: UserDefinedType) -> BasicStructure {
        BasicStructure {
            field1,
            field2,
        }
    }
}
```

Listing 3.3: Example of a `struct` and its `impl` block.

Enums

Rust's `enums`, while similar in purpose to `C` `enums`, are significantly more powerful and cover the functionality of `unions` as well. In Listing 3.4, we see an example of a simple `enum`, like the ones that we can find in other programming languages.

Another feature that enums have is to store along with the "type", like in the previous example, an associated value, making it similar to the C union type. This comes in handy particularly since Rust does not allow null pointers.

```
enum Resolution {
    FHD,
    QHD,
    UHD,
}
```

Listing 3.4: Example of an `enum`.

The `Option` enum

The `Option` enum is used for data that can or cannot be present, to substitute the concept of a null pointer. It is defined in the standard library and it is in its prelude, meaning that there is no need to bring it into scope. The definition is something like the one provided in Listing 3.5. This is a very important feature of Rust, which gets rid of null pointers, a design which was referred to as the *Billion Dollar Mistake* by its creator, Tony Hoare [11].

```
enum Option<T> {
    None,
    Some(T),
}
```

Listing 3.5: The `Option` enum.

Before using the value inside an `Option`, we need to extract it. While there are many ways to do this, they fall under three main categories:

1. We know from the structure of the code that there is some value inside the `Option`, but the compiler is not able to infer it. This can happen for example when we have dealt with the `None` case in some other place. In this case we can use the method `.expect("Why we expect this to be set")`, or, perhaps more appropriately the macro `unreachable!("This should never happen because ...")`. If this turns out to not be the case, we will get a panic where the message is the one passed, so that we can check why the logic was incorrect.

2. We don't know how to deal with the `None` case, so we panic in case we encounter it, which is done again with the `.expect()` method, where the message should be especially clear, so that the user can make sure that this case does not happen.

3. We want to deal with both cases in a way that the compiler statically knows that both cases are dealt with. This is often done using a `match` construct, which we will discuss in a later section.

A method we did not discuss, is the `.unwrap()` method, which is the same as the `.expect()` method, where a reason is not passed. It is usually employed in prototyping for quick tests, but should be replaced in the final code with one of the cases described before.

3.3.1 The `match` construct

The `match` construct is one of the most important control flow constructs in Rust, and allows for powerful operation using enums. The basic behaviour of a `match` is similar to a C `switch` construct, like shown in Listing 3.6, where the value of `s` after the code execution is `"the image is quad hd"`.

```
let res = Resolution::QHD;
let s = match res {
    FHD => "the image is full hd",
    QHD => "the image is quad hd",
    UHD => "the image is ultra hd"
}
```

Listing 3.6: Basic `match` example.

The `match` construct is particularly useful when dealing with enums such as `Option` and `Result`, to have a structured way to deal with errors and empty `Option`s, like shown in Listing 3.7. If we forget to deal with the `None` case for example, the compiler will tell us that the `match` is not exhaustive, making sure that each possibility has a defined action.

```
let res = function_that_returns_option();
match res {
    Some(value) => {
        // do something with `value`,
        // which is the content of the Option
    },
    None => {
        // return an error, ask for a new value, etc.
    }
}
```

Listing 3.7: `match` example with `Option`

To define a default action in case no other branch matches the case, we can use the branch `_ => // some action` as the last item of the `match` (note that if a values matches more than one branch it will take the top one).

3.4 Ownership and references

The concepts described in this section, are some, if not the most important to the Rust programming language, as they are responsible of the memory safety of Rust programs, and are probably going to be some the hardest to grasp with for programmers that are used to other programming languages. The rules that we will describe throughout, are enforced by a compiler module called **borrow checker**; when we refer to the compiler in this section, we will mean

specifically the borrow checker.

Let's start simple with two variable declarations (Listing 3.8): one where the data is on the stack, the other with data allocated on the heap. After the initialization, we will say that `number` and `string` are *owners* of their respective data stored in memory, regardless of which area of memory (this *data* from here onward will be referred to as a value).

```
{
    let number = 25;
    let string = String::from("This is a dynamic string");
}
```

Listing 3.8: Simple variable declarations.

A value must have a single owner at all times while it is valid, so assigning a value to another variable, will **move** the ownership to the new variable and invalidate the first. If we try to access a variable after its value has been moved, the compilation process will fail. Listing 3.9 shows the specifics of this process.

```
{
    let string = String::from("This is a dynamic string");
    let new_str = string; // here the move happens
    print!("{}", string); // this will result in a compile error
}
```

Listing 3.9: Valid and invalid variables.

The move behaviour is the default in Rust. This is because it is the *safer* behaviour: if with a move we get no compilation error, it means that the original variable is never used from there forward; if, on the other hand, we get an error, we will need to think what our desired outcome was: do we want to have another copy of the value (in which case we use the `.clone()` method), or do we want another variable to *point* at the same value (which we will describe later how to do). This pattern is similar to the `let` vs `let mut` one, where you need to opt-in to the less *safe* option, rather than having it being the default.

There is an exception to the previously described case, and this is if the value in question implements the trait `Copy` (we will discuss traits in more detail later on). In that case, when we assign the content of a variable to another, rather than moving it, the value is copied to another memory cell (actually most likely only if it is modified, to optimize the code) and the each variable is the owner of their own value. The `Copy` trait is implemented for types where copying is not an onerous operations, and intuitively we expect this to be the case, for example for numbers.

The reasons for the limitation introduced by Rust are two:

- Since each value has only one owner, it cannot happen that the value is modified by, for

example, a function call, without the explicit passing of the variable.

- Once the owner of the value goes out of scope, the value can be automatically dropped, since we know for sure no one else has access to it (more details about this when we will discuss references). This is the way Rust manages memory, to make sure that on (unintentional) memory leak ever happens. This also means that the programmer never has to call the `.drop()` method explicitly to deal with freeing memory (the reason to call the method is often to intentionally invalidate a value), and gets rid of *double free* errors as well.

What we described until now will, rightly, feels too constrained: most programs require us to have access to values at the same time from different context. This is where **references** come into play. References are similar to pointers in other languages, and are sometimes called pointers themselves, but there are some important differences.

There are two types of references: immutable references and mutable references. Listing 3.10 shows how we can create both. A value can have any number of immutable references, or a single mutable reference.

```
let string = String::from("This is a string");
let ref_str = &string; // immutable reference
let mut num = 10;
let num_ref = &mut num; // mutable reference
```

Listing 3.10: References example.

While possible to create a reference like so, the more common case is to create a reference to pass to a function or a method. The function itself will define what should be passed to it; there are three possible cases for a function's argument:

- It needs to only be read, in which case an immutable reference is sufficient.
- It will be modified by the function body, which means the caller needs to pass a mutable reference.
- The value will be consumed, which means the value is moved inside the function, so the caller does not have access to it after the return.

```
fn concat(accumulator: &mut String, value: &String) {
    *accumulator += value;
}
```

Listing 3.11: References as arguments.

Listing 3.11 shows a function that takes both a mutable and an immutable reference. The `accumulator` needs to be mutable because its content will change with the function call, while

`value` is only read, so it can be immutable. Note that if we removed the & from the type of `value`, it would be consumed, so it will not be possible to access its value after the function has been called. While perhaps a bit confusing at first, this behaviour can be quite useful. A typical example of this is when we pass some value to a constructor method: after the call, the value should be owned by the struct, and not directly accessible in any other context (this is a case where even safe languages like Java create potential bugs: when you pass a reference to the constructor, it is your responsibility to then use setter methods to modify it, but it is not enforced by the compiler).

Another advantage to the Rust way of doing things, is that both the function definition and the function call, make evident without the need for documentation, that the function will have some side-effect, it will be evident on specifically which argument the side-effects will take effect. This is also a reason why it is important to require a mutable reference exclusively if the value will actually be modified; the good news is that the linter will give us a warning if this is not respected.

3.4.1 Lifetimes

One question that could arise from the previous discussion is: what happens if the owner of a value goes out of scope, causing the drop of the value, while a reference to it is still in scope? The answer is the compilation process will fail, telling us that a reference *outlives* the value. However sometimes it is not obvious how long a value should live.

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 3.12: Example of ambiguous lifetime [42].

An example of this is code Listing 3.12. How long will the value referenced by the `a` live, if we write the code `a = longest(x, y)`? As long as `x` or `y`. In this case, the only safe options is to have the reference be valid as long as both `x` and `y` are both still valid, as `a` could be a reference to either of their values. To let the compiler know this, we can use **lifetime annotations**, as shown in Listing 3.13.

We will not go into details on lifetimes, as most of the times the compiler is able to infer them. While it is true that it can be complex to write correct lifetime annotations, two points should be noted:

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 3.13: Lifetime annotations[42].

1. Most of the times, it is possible to rewrite the code so that the lifetime annotations are not necessary, for example by having `a` becoming the owner of the value. The fact that Listing 3.12 used `&str` is not a coincidence: string references are one of the main causes of lifetimes problems, and that's the reason why they are usually substituted with dynamically allocated strings in such situations.

2. While dealing with lifetimes is laborious, it is often not a limitation of the compiler that is unable to infer the correct lifetime (however sometime it is), but rather an actual logical question. The example provided is one case where a very smart compiler could tell the correct lifetime, however this is the case because it is a toy example, and a slightly more complex logic would make it a very hard, or impossible problem.

3.5 Standard library

This section describes some of the types and functionalities we can find in the Rust's standard library. When we refer to the standard library we really mean three packages: `core`, `alloc` and `std`. In hosted enviroments this difference is of little impact, as everything that is defined in `core` and `alloc` is re-exported by `std`, so we will not make distinctions in this section (see next section for more detail).

3.5.1 Collections and Iterators

Rust defines a number of collections in its standard library [37]:

- Sequences: `Vec`, `VecDeque` and `LinkedList`.
- Maps (or dictionaries): `HashMap` and `BTreeMap`.
- Sets: `HashSet` and `BTreeSet`.
- Other: `BinaryHeap`.

When it comes to dealing with collections, Rust's preferred approach is a functional one, through the use of **iterators**. Each structure in the `std::collection` crate implements three methods that allow the programmer to obtain an iterator:

- `.iter()`: This method returns an iterator of immutable references of the elements inside the collection. This means that the elements can be read but not modified, and so more than one function can have access to the same underlying collection.
- `.iter_mut()`: Similar to the previous one, this allows modifications of the contents of the collection, and as such it requires an exclusive reference to the structure.
- `.into_iter()`: This method consumes the underlying collection, meaning that after the operation is completed, any reference to the collection will be invalid.

Iterators provide many useful methods typical of functional languages, such as `map`, `for_each`, `enumerate` and `reduce`, which are defined in the `Iterator` trait[38].

3.5.2 String types

There are two main types of strings in Rust: string slices `&str` and dynamic strings `String`.

`&str` is simply a slice over a `Vec<u8>` or `[u8; N]` (i.e. a view over some contiguous cell in memory) with guaranties that the content is valid UTF-8 and some more complex APIs as to make the access to its content easier. The slice itself can be over statically or dynamically allocated memory. As all slices, they don't own the underlying data, so they are somewhat harder to use in certain context, in particular inside a `struct`, which is why they are usually replaced by `String` in these cases.

`String` is a dynamically allocated string. Similarly to `&str`, the data is just `Vec<u8>`, and it has the guarantee of its content being valid UTF-8 and the additional APIs, but it owns its content.

Being encoded using UTF-8, makes something simple as iterating over characters not that straight forward. For this reason all string types define two methods to iterate over them:

1. `.bytes()`: returns an iterator that yields one byte at a time, making the user responsible for interpreting the data; It is used mostly for I/O or copying operations.
2. `.chars()`: returns an iterator that yields unicode *code points*. These usually correspond to what we call characters, at least in western languages, but not always. The number of bytes yielded depends on the size of the single unicode code point.

While the `.chars()` method will be sufficient in most cases, if we need to deal with more complex characters we could want to iterate over *graphemes*, which can be made up of one or

more unicode code points. This is not directly supported in the standard library, but an be achieved using external crates.

3.5.3 Atomics

Atomic operations are very important in parallel programming. They are a way to make sure that during an operation no other thread (or interrupt) will cause the operation to have an undefined result. In Rust, due to the complexities in using them directly, the programmer is encouraged to use easier ways to deal with parallelism, namely `Mutex`es and `Channels`. In this thesis, however, we did not use `Mutex`es and `Channels` as where they are available, namely in hosted environments, they are not necessary for the parallelization, while in the bare metal version of the benchmarks, where they would be useful, they are not available, as there is no Operating System. For this reason, we have use Atomics instead.

There are many Atomic types in Rust, which include all primitive integer types, both signed and unsigned, booleans and an atomic pointer type. On different architectures the underlying implementation could rely on atomics, but it is not guaranteed to.

Atomic operations require an *ordering* to be provided, which specifies if an operation has to happen before (or after) others, or if it can do either. The specifics of this are quite complex, so much so that Rust does not define its own memory model when it comes to them, but rather uses the C++ specification [39]. This complexity is one reason why atomics are bug prone, which is why we will use always the strictest ordering, `SeqCst`, which can reduce performance but makes the code more likely to be correct.

3.6 Embedded environments and `no_std`

Not all platforms support the whole standard library. In particular, embedded platforms usually do not, especially when there is no operating system. For this reason these environments are usually referred to as `no_std`. We still have access to the `core` package, which contains many useful APIs. Even in `no_std` environments it is possible to have access to dynamically allocated structures, which are in the `alloc` crate, however we need to specify an allocator to use them.

As we will see later in the thesis, in our bare metal versions of the benchmarks we do not use any allocator, as we deal with statically allocated data, so we will not go into detail on how this can be achieved.

3.7 Traits

Rust has some Object Oriented features, but a notably missing one is **inheritance**. In an extreme of the "composition over inheritance" principle[9], only composition is available in Rust, and it is achieved using **traits**.

When we say composition, what we mean is that traits define behaviour, and a struct can implement a specific trait to show that it supports the given behaviour. Any number of traits can be defined by a single struct. An important point is that a trait can define methods with or without a default implementation. This is not a problem in Rust (while it is for example in Java, where a class cannot inherit from multiple abstract classes to avoid multiple definitions of the same method), because in Rust when calling a method from a trait, the given trait needs to be in scope; if the same method is defined in two different traits and both are in scope, then we need to specify which one we are trying to call, otherwise the code will not compile.

Traits are used to define behaviour, so for example the standard library contains a trait called `Deref`, which allows to use the `*` operator to dereference something like if it was a reference. This characteristic makes them useful in generics contexts.

A generic structure or function allows to reduce code duplication by extracting an operation that is common to different types. This is the same concept as templating in C++. This approach, however, is very implicit and can cause difficult to debug errors.

A list of a few of the most commonly used traits defined in the standard library follows next:

- `Display`: Used to derive a textual representation of the type.
- `Debug`: Similar to `Display` but used for debugging purposes, can be automatically generated.
- `Copy`: Makes the data of the type be copied on assignment, should only be implemented if the copy operation is easy (i.e. no need to follow references).
- `Clone`: Allows explicit copy of the data of the type, requires the programmer to explicitly call the `clone()` method.
- `Deref`: Allows to use the `*` operator on the type.
- `IntoIter`: Specifies the method to create a consuming iterator from the type.

There are a lot more, but this is to help understand what kind of functionalities are defined in traits.

3.7.1 Trait bounds

To allow for *zero cost abstraction*, just as in C++, generic data structures and functions are **monomorphised** at compile time. This means that an instance of them is created for each type that is used when creating the structure or calling the function. Note that if the compiler is not able to infer the type of every call, compilation will fail and we will need to use `Trait objects` rather than generics.

A trait bound in Rust, specifies that a given type can be used in the generic context only if it implements the given trait. For example if we want a generic function that returns the sum of two generic values, we need a trait bound to specify that the addition operation is defined on the generic type as shown in Listing 3.14 (the trait bound specified `std::ops::Add<Output = T>` means that variables of type `T` can be added to each other, and that the result will itself be of type `T`).

```
1 fn add<T: std::ops::Add<Output = T>>(a: T, b: T) -> T {
2     a + b
3 }
```

Listing 3.14: Generic function for addition

Even without specifying the trait bound, at compile time the type will be known, so the compiler will know if the addition operation is defined for the type on which the function is called. So why does Rust require trait bounds? This is a mechanism that reduces the likelihood of bugs, and clearly specifies where the problem lies at compilation, so as to facilitate the solution.

As an example, let us consider our `add` function from before. Should we be allowed to pass two values of type `String` to the function? Likely that is not what the code is intended to do, however the sum operator on `Strings` is implemented and as such we could call the function. If, on the other hand, we define our own trait and implement it only on the types that we believe should be passed to the function, we know for sure that the behaviour is well defined. When we try and run the function on some arguments that do not implement the trait, the compiler will clearly point out that, for example, the `String` type does not implement the given trait, and the programmer will decide whether that is an incorrect call to the function, or if the trait should be implemented on `String` as well.

This highlights a further nuance in trait bounds: even when a trait exists that defines some useful behavior, it might be preferable to wrap it in a trait we define, so that we have control over which types implement it, making the behavior even more explicitly defined. This depends if the function in question will work on a few well defined types, or if it is intended to implement some generic functionality that should be permitted on a wide and unknown variety of types.

3.8 Useful crates

Rust's philosophy when deciding what to include in the standard library, is that only basic and universal behaviour should be present. As an example of this design decision, `channels`, meaning a tool for communication between threads, were included in the standard library. This inclusion has been defined as *opinionated* in an official Rust blog post, as they are a model for concurrent programming rather than a fundamental parallel programming feature, also going as far as saying "concurrency is entirely a library affair" [43]. This means that a lot of useful functionalities are not included, and we need to use external crates to have access to them. This also means that external dependencies in Rust should not necessarily be seen as something that if possible we should get rid of, as in many cases they are considered to be the correct way to deal with the problem at hand. In this section we present two widely used crates were this is the case, respectively for command line interfaces and concurrent programming.

3.8.1 `Clap`

The **clap**[3] crate (Command Line Argument Parsing) is the most commonly used crate to parse command line arguments in Rust. It offers advanced features and makes writing a well documented and easy to use command line interface particularly easy.

There are two ways to create the argument parsing using the clap: using the `Command` struct or using *procedural macros*[27]. We will concentrate on the latter as the more elegant and easy to use of the two.

The first step in creating the interface, is to define a structure with a field for each argument we are trying to capture. The struct needs to be annotated with `#[derive(Parser)]` so that at compile time the parsing method can be created.

Each field inside the structure is then annotated using an `#[arg()]` attribute, as well as a documentation comment, that will be used in the generation of the output to the help command. Inside the arg attribute we can specify some settings about each argument, such as which versions are allowed (as in short e.g. `-h` and long e.g. `--help`), if it should have a default value when it is not specified, if it accepts more than one value and so on. To specify that an argument is not required, we can wrap the expected type in an `Option`, so that if it is not provided the value will be `None`.

After having specified all the required arguments and their relative settings, we get automatically from clap a well formatted help command and helpful error messages if the user does not provide a required argument or it provides a not valid type, as clap also deals with the casting into the required types.

Another useful feature is the possibility of an argument type to be an `enum`, so that only

some specific values are allowed.

```rust
#[derive(Parser, Debug)]
pub struct Args {
    /// Size of the matrix
    #[arg(short, long)]
    pub size: usize,

    /// Prints the output in human readable format to stdout
    #[arg(short, long, default_value_t = false)]
    pub output: bool,

    /// Prints the kernel execution time to stdout
    #[arg(short, long, default_value_t = false)]
    pub timing: bool,
}
```

Listing 3.15: Arguments structure example.

```
Usage: test [OPTIONS] --size <SIZE>

Options:
  -s, --size <SIZE>  Size of the matrix
  -o, --output       Prints the output in human readable format to stdout
  -t, --timing       Prints the kernel execution time to stdout
  -h, --help         Print help
```

Listing 3.16: Help output from Listing 3.15.

3.8.2 Rayon

The **rayon** [30] crate gives Rust powerful data parallelism APIs. Similarly to OpenMP, it allows for minimal code modification to achieve good parallel results, even though it does not offer the same level of control.

The crate introduces the `ParallelIterator` trait, that defines the same, or in some cases slighly different, methods as the `Iterator` trait in the standard library, which means that some code that manipulates an iterator can be used, usually without any modifications, on its parallel version.

To create a parallel iterator, `Rayon` offers three methods, similar to the standard library ones:

- **`par_iter`**: the equivalent of `iter` method, returns a parallel iterator over references to the elements in the collection.
- **`par_iter_mut`**: the equivalent of `iter_mut` method, returns a parallel iterator over mutable references to the elements in the collection.

- **into_par_iter**: the equivalent of `into_iter` method, returns a parallel iterator over the elements in the collection and consumes the collection.

To parallelize some code using `Rayon` then, all it takes is to substitute the standard library method to create the iterator with its `Rayon` equivalent, and the other operations usually require no other modification, as shown in Listing 3.17.

```
use rayon::prelude::*;
fn main() {
    let vec = vec![1, 2, 3, 4, 5];
    let sequential_res: i32 = vec.iter().map(|el| el*el).sum();
    let parallel_res: i32 = vec.par_iter().map(|el| el*el).sum();
    assert_eq!(sequential_res, parallel_res);
}
```

Listing 3.17: Parallelizing with `Rayon`.

The two main methods that are an exception to the previous case, are `reduce` and `fold`: these use a slightly different syntax, as well in the reduce case it does not guarantee the same order of operations, which means the result could be different; if this is not intended, we can use the fold method instead, with an obvious performance deficit.

Chapter 4

Hosted Benchmarks

This chapters presents the implementation of the hosted version of the benchmarks, which means that they run on top of an OS. First we define the data structures, traits and method implementations of the code to be benchmarked, defined in the `obpmark_library` crate, before delving in the code of the benchmarks executables.

4.1 Data Structures

The benchmarks from the GPU4S Bench project operate on vectors and matrices, in most cases 2D matrices. As a consequence, the main data structure required is a matrix data structure. While the C version of the benchmarks employs a dynamically allocated one-dimensional buffer for this purpose, we decide to develop two distinct versions of the data structures, to compare performance and ease of use of the two approaches:

- `Matrix1d`, that stores the data in a 1D Vector.
- `Matrix2d`, that stores the data in a 2D Vector.

The difference in the memory disposition of the two version depends on the platform and allocator on which the code will run. As the structure will be allocated at the start of the executable, we suspect that the memory will look very similar amongst the different versions. This could make the 2D vector solution preferable, as it will be less bug prone than the 1D version, by allowing an easier use of iterators. We will look further into this when analysing the results of the benchmarks. Listing 4.1 shows the `struct` code.

Both structures are generic, so that they can contain any type that implements the trait `Number`, which we will discuss in detail in the following section.

```rust
pub struct Matrix1d<T: Number> {
    data: Vec<T>,
    rows: usize,
    cols: usize,
}

pub struct Matrix2d<T: Number> {
    data: Vec<Vec<T>>,
    rows: usize,
    cols: usize,
}
```

Listing 4.1: Matrix struct in its 1D and 2D version

When analysing the benchmark executables we will see how the `Matrix` methods will be called from the benchmarks; to allow the benchmark code to not be specialized for the different matrix types, all operations on them are defined in traits. The `BaseMatrix` trait defines basic operations on matrices:

- Initialization of a new matrix from a 2D vector representation.
- Retrieval of a 2D vector representation of the data.
- Initialization of a new matrix populated with zeroes.
- Initialization of a new matrix with random contents based on a seed.
- Input and output operations on files.
- Conversion to a 1D vector data representation, which is needed for verification.
- Reshaping the matrix by adjusting its row and column counts.

4.2 The Number trait

The original C code, provides the benchmarks with different data types that are selected at compile time. We decided that the library should not need to be compiled to a specific numeric type, but rather should be generic over the content of the matrix. This allows the user of the library to easily have matrices that contain different types in the same context, and makes the library useful also outside of the benchmarks code.

There are four types that we will need to support for the various benchmarks (not all the types are supported for all benchmarks, we will later show how to deal with this):

- The integer type `i32`.

```rust
pub trait BaseMatrix<T: Number> {
    fn new(data: Vec<Vec<T>>, rows: usize, cols: usize)
        -> Self;

    fn get_data(&self) -> Vec<Vec<T>>;

    fn zeroes(rows: usize, cols: usize) -> Self;

    fn from_random_seed(
        seed: u64, rows: usize, cols: usize, min: T, max: T
        ) -> Self;

    fn from_file(path: &Path, rows: usize, cols: usize)
        -> Result<Self, FileError>;

    fn to_file(&self, path: &Path)
        -> Result<(), std::io::Error>;

    fn reshape(&mut self, new_rows: usize, new_cols: usize)
        -> Result<(), Error>;

    fn to_c_format(self) -> Vec<T>;
}
```

Listing 4.2: The base matrix trait (some code excluded)

- The single precision floating point type `f32`.
- The double precision floating point type `f64`.
- The half precision floating point type `f16`; this is not available in the standard library, so we used the crate `half` [10]. The inclusion of the f16 version of the benchmarks is for consistency with the C version, however at the moment the support for intrinsics is very limited making the code very slow. This is an area in which Rust is rapidly moving forward and we should expect the situation to be quite different in a short time after the publication of this thesis. During the time of writing of this thesis the hardware support on x86 has changed already. For this reason we will not discuss further this feature, as it is subject to rapid change.

4.2.1 The `num_traits` crate

The `num_traits`[19] crate is used to help with definitions of common numerical behaviour in Rust programs. While external to the standard library, it enjoys widespread adoption, and many external numeric types implement the crate's traits.

The num_traits crate offers many useful traits; here we describe the ones we used in defining our own traits:

- `NumRef`: Encompasses basic numeric operations like addition, subtraction, multiplication,

and division, both for a type and its reference.

- `NumAssignRef`: Adds assignment-based operations to `NumRef` (e.g. `+=`, `*=`).
- `AsPrimitive<f64>`: Allows casting to f64,serving as a superset of all types to be implemented.
- `PrimInt`: Contains common operations on integer types.
- `Float`: Contains common operations on floating point types.

4.2.2 Limitations of num_traits

While the traits discussed are useful to define common behaviour, not all the operations that someone might want to do on a number are covered by the num_traits crate. The crate funty [6] was inspiration for some of the missing traits, in particular for its `Fundamental` one, that enforces basic behaviour such as `Copy`, `Display` and `Debug`.

```
pub trait Fundamental:
    'static
    + Sized
    + Send
    + Sync
    + Unpin
    + Clone
    + Copy
    + Default
    + std::str::FromStr
    + PartialEq<Self>
    + PartialOrd<Self>
    + std::fmt::Debug
    + std::fmt::Display
{
}
```

Listing 4.3: Fundamental trait from `funty` crate

Until now we described behaviour that we could anticipate we would need even before writing a single benchmark function, also because they are bundled in existing traits. The additional behaviour we describe from here on, was mostly added while developing some specific benchmark. The typical process would be: 1. Do some operation you know you can do on a numeric type, 2. Have the compiler complain that it is not possible for the given bounds, 3. Add a new bound to the existing trait. This can be somewhat frustrating, especially when this is not straight forward, as in some of the cases we will describe later.

The ability to obtain a `Number` from an iterator of `T: Number` and `&T: Number` also needs to be specified as a trait bound. At this point two missing behaviours still remain:

- Obtain a `Number` from a sum operation on an iterator of elements of type `T` where `T: Number` or `&T: Number`. This is done with the following trait bounds:
  `std::iter::Sum<Self>`
  `for<'a> std::iter::Sum<&'a Self>`.

- `Serialize`: A way to serialize the numbers to their bytes representation and back; fundamental numeric types all have this behaviour available but it is not behind a trait, as such we need to implement the trait to call the method defined in the standard library. Since the method name is the same for all types, we can achieve this through a macro rule (Listing 4.4).

- `RngRange`: A way to generate numbers in a given range. In this case a trait does exist (`SampleUniform` from the `rand` crate), with some complications for the `f16` type which we will not delve into.

```
macro_rules! impl_serialize {
    ($type: ty) => {
        impl Serialize for $type {
            type Bytes = [u8; core::mem::size_of::<$type>()];
            ...
            fn to_ne_bytes(self) -> Self::Bytes {
                self.to_ne_bytes()
            }
            fn from_ne_bytes(bytes: Self::Bytes) -> Self {
                <$type>::from_ne_bytes(bytes)
            }
        }
    };
}
```

Listing 4.4: Fundamental trait from `funty` crate

4.2.3 More specific bounds

The `Number` trait is designed to work with all the types supported by the benchmarks. However not all benchmarks support all the different types: as an example the Softmax benchmark only works on floating point types. Another situation that sometimes happens is that the implementation is very different amongst different types. To allow for this, two more specific traits are defined, to differentiate between integer and floating point types. Listing 4.5 shows how this was easily achieved thanks to the `num_traits` crate.

```
pub trait Float: Number + num_traits::Float {}
pub trait Integer: Number + num_traits::PrimInt {}
```

Listing 4.5: Integer and Float traits.

4.3 Sequential traits

In this section, we will go through the steps required to implement the library code of a benchmark function, and we will analyze a few examples of benchmark traits.

As discussed previously, the matrix methods are defined inside traits, so that the same function can be called on different implementations of the matrix structure from the benchmark code. Each benchmark trait consists of two member functions:

- The main benchmark function, that is called from the executable and is timed to assess the performance of the hardware.
- A function that operates on a subsection of the matrix. This often is an extraction of an internal loop of the algorithm.

This code separation does it a little bit less readable, but it allows to reduce code duplication when other implementations, in particular the parallel versions, are coded.

```
pub trait MatMul<T> {
    fn multiply_row(&self,
        other: &Self, result_row: &mut [T], row_idx: usize);
    fn multiply(&self, other: &Self, result: &mut Self)
        -> Result<(), Error>;
}
```

Listing 4.6: Matrix multiplication trait

For example, Listing 4.6 shows the trait for the matrix multiplication operation. In this case the function `multiply_row` calculates one row of the output given the two input matrices and the index of the row to calculate. The `multiply` function, on the other hand, is the one that will be benchmarked, that deals with calling `multiply_row` for each row of the result matrix.

Listing 4.7 shows the implementation of the `multiply_row` function for the 1D version of the matrix structure. The code is really straight forward, being the inner two loops of a matrix multiplication function, with the value of the outer loop index being passed as the argument `row_idx`.

The `multiply` function then is very easy, since all it has to do is implement the outer loop and call the `multiply_row` function with the correct arguments. We will discuss this further when analysing the parallel implementations, where the design of the functions will become more relevant.

The matrix multiplication example was chosen because representative of most of the benchmarks available, where the inner function operates on a row of the result and the calculation of each row of the output is completely independent of the others. In benchmarks such as the finite

```rust
fn multiply_row(&self,
    other: &Self, result_row: &mut [T], row_idx: usize) {
    let i = row_idx;
    for j in 0..other.cols {
        let mut sum = T::zero();
        for k in 0..self.cols {
            sum += self.data[i * self.cols + k] *
                other.data[k * other.cols + j];
        }
        result_row[j] = sum;
    }
}
```

Listing 4.7: `multiply_row` implementation.

```rust
fn multiply(&self, other: &Self, result: &mut Self)
    -> Result<(), Error> {
    ...
    result
        .data
        .chunks_exact_mut(self.cols)
        .enumerate()
        .for_each(|(i, result_row)|
            self.multiply_row(other, result_row, i)
        );
    Ok(())
}
```

Listing 4.8: `multiply` implementation for `Matrix1d`.

impulse response filter, where the input and output are one-dimensional, the inner loop operates on a single element of the output rather than a row, as we can see from the trait definition in Listing 4.9.

```rust
pub trait FirFilter<T> {
    fn fir_filter_element(&self, kernel: &Self, element_idx: usize)
        -> T;
    fn fir_filter(&self, kernel: &Self, result: &mut Self)
        -> Result<(), Error>;
}
```

Listing 4.9: Finite impulse response filter trait.

Another case where the design is somewhat different is where, instead of having only operations that can happen in any order, we have the need to enforce some kind of order amongst operations: this is the case for example in the softmax and the wavelet transform benchmarks, and it will become more relevant in the parallel versions of the benchmarks.

4.4 Parallel versions

The parallel version of the benchmarks comes in two flavours: one that uses only the standard library, and one that uses the Rayon crate described in the Rust chapter. This way we can assess if the added complexity of dealing with data parallelism ourselves is worth it in some metric, such as performance, code maintenance, etc.

4.4.1 Using the standard library

The easiest way to parallelize the code, is to iterate over the result matrix, and spawn a thread for each row to do the computation. This would look something like the code in Listing 4.10. The advantages of this approach is the readability and simplicity of the code, which becomes even more evident when comparing it with the one in Listing 4.8; aside from the thread scope and calling the `spawn()` method, the code is identical to the sequential version.

We did not comment the `chunks_exact_mut()` method in the sequential version, because the reason for it being there is the design of the row calculation function, which is suited to parallel code. In the matrix multiplication example in Listing 4.7, the function modifies the result by receiving mutable slices of its rows, rather than through a mutable reference to the whole result matrix. This is not strictly necessary in the sequential version, as the calls do not overlap with each other. However it is crucial in the parallel versions, because each thread needs a mutable reference to part of the matrix simultaneously: `chunks_exact_mut()` does exactly this while making sure that there is no overlap amongst the different chunks, which makes sure that to avoid any data race between threads.

```
thread::scope(|s| {
        result
                .data
                .chunks_exact_mut(result.cols)
                .for_each(|chunk| {
                        s.spawn(move || {
                                // Do the row calculation here
                    }
                });
        });
});
```

Listing 4.10: Template for naive parallel implementation

The disadvantage of the solution presented is that the number of threads spawned is not constant, rather it grows with the size of the matrix, which significantly degrades the performance of this solution if the number of rows is large.

To improve on this, what we need to do is spawn a proper number of threads, which is usually

equal to the number of cores, or hardware threads, available on the platform, and assign multiple rows to each thread. This makes the code only slightly more complicated than the previous case, but improves significantly the performance. As shown in Listing 4.11, each thread has an internal loop (`chunk.chunks_exact_mut(result.cols)`, so that it can call the row function on each row of the chunk. The use of the method `chunks_mut` instead of the exact version in the outside loop, allows us to run the code regardless of the number of rows of the matrix, as it does not need to be a multiple of the number of threads.

```
let rows_per_thread = (self.rows - 1) / n_threads + 1;

thread::scope(|s| {
    result
        .data
        .chunks_mut(result.cols * rows_per_thread)
        .for_each(|chunk| {
            let start_row = chunk_idx * rows_per_thread;
            s.spawn(move || {
                chunk.
                    chunks_exact_mut(result.cols).
                    for_each(|row| {
                        // Do the row calculation here
                    }
            }
                });
        });
});
```

Listing 4.11: Template for parallel implementation

4.4.2 The Softmax Benchmark

The softmax benchmark, as mentioned before, has a slightly different design than the one presented with the matrix multiplication case. This is because, before we can calculate the element of the final matrix, we need to now the sum of the exponentials of the whole matrix. For this reason the code has two parallel sections, with a synchronization in the middle so that the sum is available to the second parallel section. As shown in Listing 4.12, this only requires us to have two threads scopes rather than one, so that the threads are joined and the sum is calculated before the start of the second scope.

While the code presented is long, especially when compared to the rayon version as we will see, if we consider only the code specific to the benchmark, without the part to spin the threads presented in the previous section, which is the same in all benchmarks, we can see that the code is indeed very straight forward.

```rust
fn parallel_softmax(&self, result: &mut Self, n_threads: usize)
    -> Result<(), Error> {
    ...
    let rows_per_thread = (self.rows - 1) / n_threads + 1;
    let mut total_sum = T::zero();
    thread::scope(|s| {
        result
            .data
            .chunks_mut(result.cols * rows_per_thread)
            .enumerate()
            .map(|(chunk_idx, chunk)| {
                let start_row = chunk_idx * rows_per_thread;
                s.spawn(move || {
                    let mut sum = T::zero();
                    chunk.chunks_mut(self.cols).enumerate()
                        .for_each(|(i, result_row)| {
                            sum += self.softmax_row(result_row, start_row + i);
                        });
                    sum
                })
            })
            .for_each(|handle| total_sum += handle.join().unwrap());
    });
    thread::scope(|s| {
        result
            .data
            .chunks_exact_mut(result.cols * rows_per_thread)
            .for_each(|chunk| {
                s.spawn(|| {
                    chunk.iter_mut().for_each(|el| {
                        *el /= total_sum;
                    });
                });
            })
    });
    Ok(())
}
```

Listing 4.12: Parallel implementation of Softmax benchmark

4.4.3 Using Rayon

As we described in the data parallelism section of the Rust chapter, Rayon is a great tool for dealing with the kind of problems we are working on. This becomes evident when comparing the rayon code with the sequential code, for example for the matrix multiplication function (Listing 4.13), which we included in its sequential version in Listing 4.8.

The only modification necessary to the code is using Rayon's parallel version of the `chunks_exact_mut()` method from standard library.

The softmax benchmark presented when discussing the standard library versions, is a good example both to show the compactness that Rayon can achieve compared to the standard library code, and to showcase some case where the parallel code is not just adding a `par_` in front of the

```rust
fn rayon_multiply(&self, other: &Self, result: &mut Self)
    -> Result<(), Error> {
    ...
    result
        .data
        .par_chunks_exact_mut(other.cols)
        .enumerate()
        .for_each(|(i, chunk)| {
            self.multiply_row(other, chunk, i);
        });

    Ok(())
}
```

Listing 4.13: Parallel matrix multiplication using Rayon.

standard library iterator. The Rayon version of the code is available in Listing 4.14. The second loop is the same as in the sequential version, while the first needs to use the `reduce()` method for calculating the sum. The syntax of reduce is the typical functional syntax of the method, and allows for the reduction operation itself to be performed in parallel as well, which will not be a major consideration on our target platforms, since they will have relatively few cores (4-8), but could be relevant if we had a very large number of threads. It should be noted that the order in which the reductions will happen is not specified, which means the result could be not exactly the same due to the non-associativity of floating point operations [44], which is why we should use a tolerance when validating the results.

```rust
fn rayon_softmax(&self, result: &mut Self) ->
        Result<(), Error> {
    ...
    let sum = result
        .data
        .par_chunks_mut(self.cols)
        .enumerate()
        .map(|(i, row)| self.softmax_row(row, i))
        .reduce(|| T::zero(), |partial_sum, next_sum| partial_sum + next_sum);

    result.data.par_iter_mut().for_each(|el| {
        *el /= sum;
    });
    Ok(())
}
```

Listing 4.14: Parallel softmax using Rayon.

4.4.4 Parallel traits

Just like in the sequential case, the parallel versions of the benchmarks are defined inside traits. The parallel traits are defined inside two modules: `rayon_traits` for the Rayon versions and

`parallel_traits` for the standard library versions. In this case the traits only require one member, that being the function to be benchmarked, and the naming scheme used is the following:

- The traits have the same name as their sequential counter-part, preceded by `Rayon` or `Parallel` depending on the version.
- The method defined inside the trait has the same name as the sequential version preceded by `rayon_` or `parallel_` depending on the version.

An example of parallel traits for the rectified linear unit benchmark is shown in Listing 4.15.

```
pub trait RayonRelu {
    fn rayon_relu(&self, result: &mut Self) -> Result<(), Error>;
}
pub trait ParallelRelu {
    fn parallel_relu(&self, result: &mut Self, n_threads: usize) -> Result<(), Error>;
}
```

Listing 4.15: Parallel traits.

4.5 The Benchmark Code

This section deals with the `benchmarks` crate, that contains the binaries' code and utilities used by them. Each benchmark in the project has its own executable that deals with argument parsing, initialization, timing, input/output and validation of the result. Besides the executable, in the benchmarks crate, there is a `lib.rs` file that defines some common functionalities and helper functions.

4.5.1 Type aliasing

The benchmark_utils module defined in the crate deals with some useful code for the benchmarks. Amongst its functionalities, it deals with defining some types based on arguments passed at compile time, in particular a `Number` type that is going to be the content of the matrices, and a type `Matrix`, which is the specific matrix implementation to be used. This is done using `#[cfg(feature = ...)]` directives, that work similarly to `#ifdef` directives in C.

Listing 4.16 shows the code needed to define the correct type for the `Number` alias. In order to pass the flag using cargo, all it takes is to pass to cargo the flag `--features "feature_name"`, and cargo will then pass the flag to rustc. There is no way yet [26] to have mutually exclusive features, however, if more than one datatype is set we will get a compile error for trying to set a type alias already defined. With the last cfg command, if no type is specified at compilation, we default to `f32`.

```rust
#[cfg(feature = "float")]
pub type Number = f32;
#[cfg(feature = "double")]
pub type Number = f64;
#[cfg(feature = "int")]
pub type Number = i32;
#[cfg(feature = "half")]
pub type Number = half::f16;
#[cfg(not(any(
    feature = "float",
    feature = "double",
    feature = "int",
    feature = "half"
)))]
pub type Number = f32;
```

Listing 4.16: Type aliasing based on compile flag.

4.5.2 Argument parsing

While the previous section dealt with compile time arguments, this deals with runtime command line arguments. This is done using the **clap** crate, which was introduced in Chapter 3.

By analysing the arguments in the original benchmark code we collected some that either are common to all benchmarks, or would make sense also in benchmarks where they are absent. These are defined in the `CommonArgs` struct (Listing 4.17) in the benchmark_utils module. This is then used by the benchmarks binaries, that are able to add arguments specific to them if needed. Some of the most important arguments allow the user to specify the size of the matrix, whether the result should be verified, whether we want the timing result to be printed and which implementation should be used, i.e. sequential, rayon or standard parallel.

4.5.3 Validating the results

The final common functionalities defined for the benchmarks is a way to validate the result. To do this, we need to call the C reference implementations from the Rust code, and compare the result to make sure that the Rust code is equivalent to the C one. The validation code is defined in its own crate (`reference_algorithms`), so that it will be callable from the bare metal version of the benchmarks as well, which we will discuss in a later chapter.

The crate contains a C file, which has the actual C code to be called, and a Rust file that defines the foreign function interface (FFI) for each of them. An FFI is a bit of Rust code, that defines how to call a foreign function, in this case a C function. Calling a foreign function is unsafe, as it is up to the programmer to provide a sound interface to call the code.

The first thing we need, to be able to call the C code, is to know which type the functions will use; for this reason we need a feature flag that will tell us which type the functions should

```rust
pub struct CommonArgs {
    /// Size of the matrix (or matrices)
    #[arg(short, long)]
    pub size: usize,
    /// Export the result in hex format to file <export>
    #[arg(short, long)]
    pub export: Option<String>,
    /// Verifies the result against 2d reference implementation
    #[arg(short, long)]
    pub verify: Option<Option<String>>,
    /// Prints the output in human readable format to stdout
    #[arg(short, long, default_value_t = false)]
    pub output: bool,
    /// Prints the kernel execution time to stdout
    #[arg(short, long, default_value_t = false)]
    pub timing: bool,
    /// Uses "mat_A.in" ["mat_B.in"] for input data, can take 1 or 2 files
    #[arg(short, long, num_args = 1..=2)]
    pub input: Option<Vec<String>>,
    /// Print the input matrix (or matrices) to stdout
    #[arg(long, default_value_t = false)]
    pub print_input: bool,
    /// Number of threads to use
    #[arg(short, long)]
    pub nthreads: Option<usize>,
    /// Random seed to use
    #[arg(long, default_value_t = 3894283)]
    pub seed: u64,
    /// Parallel implementation to use
    #[arg(value_enum, long, default_value_t = Implementation::Sequential)]
    pub implementation: Implementation,
}

```

Listing 4.17: `CommonArgs` struct.

be compiled for. We will see how the C code is specialized when analysing the build script; the Rust code, on the other hand, uses a similar mechanism to what we did when defining the type alias `Number` in the benchmark_utils module, this time using the types defined in the `core::ffi` module, as we can see in Listing 4.18.

Once `Ctype` has been defined, we need to define the argument types inside the function signature. If we look for example at the interface for the matrix multiplication function in Listing 4.19 we can see that pointers will be marked as `*const` or `*mut`, depending on if their content will be modified or not, while in this case `unsigned int` becomes a `usize`, since it is a size parameter. The `extern "C"` is needed so that the arguments will be passed using C conventions.

```rust
#[cfg(feature = "float")]
type CType = core::ffi::c_float;
#[cfg(feature = "double")]
type CType = core::ffi::c_double;
#[cfg(feature = "int")]
type CType = core::ffi::c_int;
#[cfg(feature = "half")]
type CType = core::ffi::c_float;
#[cfg(not(any(
    feature = "float",
    feature = "double",
    feature = "int",
    feature = "half"
)))]
type CType = core::ffi::c_float;
```

Listing 4.18: Defining `Ctype` based on feature value.

```rust
extern "C" {
    pub fn matrix_multiplication(
        a: *const CType,
        b: *const CType,
        c: *mut CType,
        n: usize,
        m: usize,
        k: usize,
    );
}
```

Listing 4.19: Defining the interface for matrix multiplication.

The build script

By putting a file named `build.rs` in the top directory of the crate, we can specify some actions that we want to run when calling `cargo build`. The reason why we want to use a build script in this case, is so that we can have the C source code inside the project, that can be compiled using the required flags when building. To do this we use the `cc` crate, which is a build dependency, meaning it will not be linked to the resulting library, but rather is used by cargo to build the library.

As we can see in Listing 4.20, the crate allows us to specify which files should be compiled, using which flags: in particular we define the flag `C_type` depending on the value of the feature flag previously discussed, so that the `#ifdef` used by the reference C code can define their own type aliases, and the optimization level for faster validation.

The build script has also some other functionalities, which we will discuss in the Chapter 5.

```
cc::Build::new()
    .file("src/cpu_functions.c")
    .define(C_TYPE, None)
    .opt_level(3)
    .compile();
```

Listing 4.20: Using `cc` to customize compilation.

4.5.4 Putting it all together

Now that all the useful code has been defined, we can walk through the structure of a benchmark binary. All the benchmarks have the same structure, so for the sake of this section, the code listings will be from the max pooling benchmark.

First, we need to define the benchmark specific arguments. This is done both in benchmarks that have specific arguments, and in those that do not, so that we can specify the command name to be printed when `--help` is passed to the binary. Listing 4.21 shows the definition, while Listing 4.22 reports the first few lines of the output of the help command.

```
#[derive(Parser, Debug)]
#[command(about = "Max pooling benchmark")]
struct Args {
    /// Common arguments
    #[clap(flatten)]
    common: CommonArgs,

    /// Stride
    #[clap(long)]
    stride: usize,
}
```

Listing 4.21: Max pooling `Args` struct.

```
Max pooling benchmark

Usage: max_pooling [OPTIONS] --size <SIZE> --stride <STRIDE>

Options:
  -s, --size <SIZE>
          Size of the matrix (or matrices)
  -e, --export <EXPORT>
          Export the result in hex format to file <export>
  -v, --verify [<VERIFY>]
          Verifies the result against 2d reference implementation
...
```

Listing 4.22: Max pooling help output.

In the main body, the arguments are parsed and validated if necessary, the input matrices are initialized, filled with random numbers or from a file depending on the value of the `--input`

flag, and printed to stdout if the `--print-input` flag is set. Now the proper benchmark code starts: after the current time is stored in a variable, a `match` statement is used to select which implementation of the benchmark should be executed (Listing 4.23).

```
match (args.common.nthreads, args.common.implementation) {
    (None, Implementation::Rayon) => {
        A.rayon_max_pooling(&mut B, args.stride, args.stride)
            .unwrap();
    }
    (Some(_), Implementation::Rayon) => {
        panic!("Cannot specify number of threads for Rayon implementation")
    }
    (Some(n), Implementation::Sequential) if n != 1 => {
        panic!("Invalid parameter combination: sequential with nthreads != 1")
    }
    (_, Implementation::Sequential) =>
        A.max_pooling(&mut B, args.stride, args.stride).unwrap(),
    (Some(n), Implementation::StdParallel) => A
        .parallel_max_pooling(&mut B, args.stride, args.stride, n)
        .unwrap(),
    (None, Implementation::StdParallel) => {
        A.parallel_max_pooling(&mut B, args.stride, args.stride, 8)
            .unwrap();
    }
}
```

Listing 4.23: Selecting the max pooling implementation to benchmark.

We then store the end time, so that the execution time can be computed, and depending on the values of flags `timing`, `output`, `export` and `validation`, the information is printed and saved.

Chapter 5

Bare Metal

In this chapter, we discuss the development of the bare metal version of the benchmarks. By not relying on host platform features, they can run without an operating system so their overhead is small and results are a lot more reproducible, since there is no noise from the operating system or external interrupts and other processes. This last point in particular is an issue in parallel code on lower power platform, where the difference in execution time amongst different iterations can change significantly, making benchmarking harder.

5.1 Platform selection

The first task in creating the bare metal benchmarks, is selecting a platform we want to test them on. Since we want to have multi threaded versions of the benchmarks, we need some kind of synchronization mechanism, and without an operating system, this means that the platform needs to have support for `Atomics`. We also want the platform to be a 64 bit one, so that we can support the double precision floating point versions of the benchmarks. The candidate platforms that were considered were the following one:

- Arm, in particular AArch64 (also known as Arm64v8), which is widely used in lots of applications. However, the high licensing fees make it less attractive for custom chips for niche markets such as space.
- RISC-V, a royalty-free Instruction Set Architecture (ISA) which is still in early days, but it is rapidly gaining ground in the space sector thanks to its open nature and availability of tools.
- SPARC, one of the most commonly used architecture in space missions; its age and the fact that only one vendor (FrontGrade Gaisler) still develops processors and maintains

compilers for this ISA have led the company to consider RISC-V for its future processor products.

In the end we landed on the RISC-V architecture, for a few reasons. First of all the ease of start up code for enabling multicore support in bare metal, unlike the case of the Arm ISA, which is more complex and not as standardised and varies from board to board. Moreover, we did not have available an Arm-based board with the ability to boot in bare metal. On the other hand, we had access to the RISC-V based platform targeting high performance space applications developed in the METASAT Horizon Europe project on an FPGA, which we describe in Chapter 6, over which we had full control and which was a perfect candidate for Rust support, as we mentioned that there is an increased interest for the use of safe languages in this domain. Finally, the METASAT platform is compatible with the generic RISC-V model of the QEMU emulator, which allowed us to start developing the code in a virtual environment, until having access to the FPGA version for the performance evaluation.

There are a few priviledge levels to execute code on RISC-V. In particular the SBI (Supervisor Binary Interface) [31] offers an higher level of abstraction. We decided to develop our code for the lowest level, that is the Machine mode, which is universally available to all RISC-V platforms, in order to be portable also to other RISC-V platforms, even micro-controllers. We believe that more investigation would be interesting in this area, especially if a Rust port of the SBI where more mature than what is available at the moment.

5.1.1 The RV64GC ISA

The RV64GC instruction set architecture which is supported by the NOEL-V space processor from Frontgrade Gaisler, which is the basis of the METASAT platform was chosen. The G stands for general expansion, and is a short hand for the IMAFD expansions:

- **I**: Integer instruction set, it is the base instruction set shared by all RISC-V architectures.
- **M**: Extension that introduces multiplication and division instructions.
- **A**: Extension for Atomic instructions support, particularly important for us.
- **F**, **D**: Extensions for respectively single and double precision floating point operations.
- **C**: Allows for use of compressed instructions to reduce the size of the source code.

All the extensions described are strictly necessary for running the benchmarks, except for the **C**, which is included since this architecture is supported by Rust tools, making our life easier when building the code.

Throughout the chapter we will talk about **harts**, which are RISC-V terminology for hardware threads. Each of the threads executes the same code unless control flow diverges, making them the perfect choice for Symmetric Multi-Programming (SMP).

5.2 Booting into Rust

Differently from hosted environments, the entry point of a `no_std` [34] is not main, but it rather needs to be defined by the user. For this reason, we will need to write some low level code that interfaces with the hardware to boot our program. In this section we describe the process to boot into the VIRT QEMU machine, while later we will look into what needs to be modified to support other platforms as well. The information on the VIRT machine addresses and components is available on the QEMU github repository [28]. The contents of this section were inspired by the excellent Stephen Marz blog on writing a RISC-V OS in Rust [18]. While the final product is very different in scope, the blog was instrumental to getting Rust to run on the RISC-V emulator.

It should be noted that an easier way to do what is described in this section is to use a runtime crate, which also offers some additional functionalities such as initializing certain areas of memory before getting to main. For RISC-V this is possible through the crate `riscv_rt` [32]. Although we have a demo running on the emulated hardware, which required significantly less code and setup, we had trouble getting it to work on the METASAT FPGA platform, so we opted for the more *manual* way, which we present in this section.

5.2.1 The linker script

The linker script serves to define where the different parts of the program should be stored in memory, as well as providing the addresses of different parts of memory. We start by defining the target architecture, the base address of RAM and its length (Listing 5.1). We can we also specify the address of the entry point, using the `_start` symbol, however this is only for documentation purposes, since it is our responsibility to make sure that the start of the code is at the correct address. The `(wx)` in the ram area definition specifies that it is a read/write area that can be executed.

At this point we define three sections inside ram: `text`, `data` and `bss`. These hold respectively the code, initialized data and uninitialized data (which is set to zero) at start up. First, we define the three sections using the program headers command `PHDRS`. The `SECTIONS` section specifies the positions in which the various part of the code and data will be placed. In particular the `text` section starts with the `.text.init`, which we will define in our code to start the program, after which any other `.text` section from any file is placed (this is specified with the `*(.text.init) *(.text .text.*)` syntax). After the end of the section definition, `>ram AT>ram :text` specifies

```
OUTPUT_ARCH( "riscv" )
ENTRY( _start )
MEMORY
{
  ram  (wx) : ORIGIN = 0x80000000, LENGTH = 128M
}

PHDRS
{
  text PT_LOAD;
  data PT_LOAD;
  bss PT_LOAD;
}
```

Listing 5.1: Linker script - platform specific.

that it should be placed in RAM, in the `text` section defined previously in the program headers. The same is done for read only data, initialized and uninitialized data.

In Listing 5.2 we can also see some `PROVIDE` statements: this allows us to export symbols, that can then be used inside our program. This is useful for example for the `bss` section (block starting symbol), since we will need to set its content to be zeroes.

```
SECTIONS
{
  .text : {
    PROVIDE(_text_start = .);
    *(.text.init) *(.text .text.*)
    PROVIDE(_text_end = .);
  } >ram AT>ram :text

  .rodata : {
    PROVIDE(_rodata_start = .);
    *(.rodata .rodata.*)
    PROVIDE(_rodata_end = .);
  } >ram AT>ram :text

  .data : {
    . = ALIGN(4096);
    PROVIDE(_data_start = .);
    *(.sdata .sdata.*) *(.data .data.*)
    PROVIDE(_data_end = .);
  } >ram AT>ram :data

  .bss : {
    PROVIDE(_bss_start = .);
    *(.sbss .sbss.*) *(.bss .bss.*)
    PROVIDE(_bss_end = .);
  } >ram AT>ram :bss
}
```

Listing 5.2: Linker script - sections definitions.

Since we will not use dynamically allocated memory, we do not need to define a heap section,

while we do export some symbols to deal with stack allocation, as we will see later.

While the script is designed to work with the QEMU machine, really in most cases the only modification necessary to get it to work on another processor is to change the address and length of the RAM, as everything else is generally the same on different platforms.

5.2.2 From assembly to Rust

Now that our `.text.init` section is set to be stored at the beginning of RAM, we need to write the code that will be executed as the processor starts. Before executing Rust code, we need to do some setup work in assembly:

1. Store zeroes in the bss section.
2. Enable the interrupts we are interested in, in particular software interrupts.
3. Enable the Floating Point Unit.
4. Divide the stack amongst the different harts.

Some of these actions are dependent on whether the code we want to run is sequential or parallel. In particular in the sequential code, we expect one single hart to boot; if this is not the case, the other harts are forced in a `wfi` (wait for interrupt, a RISC-V instruction) loop. The implementation of `wfi` is platform dependent: it should not be busy waiting, but on platforms where the instruction is implemented as a `nop`, it will indeed result in busy waiting.

Once the initialization is completed, the hart (or harts) are sent to the Rust code by storing in the `mepc` register (machine exception program counter) the address of the `main` function, so that when we call `mret` the program counter gets set to the correct value.

To allow for the sequential and parallel versions, we introduce the features `sequential` and `parallel`, so that the user can specify which version they want to run. Then, depending on the feature selected, a different assembly file is linked, thanks to the code in Listing 5.3.

```
use core::arch::global_asm;

#[cfg(feature = "sequential")]
global_asm!(include_str!("asm/boot_single_hart.s"));
#[cfg(any(
    feature = "parallel",
    not(any(feature = "sequential", feature = "parallel"))
))]
global_asm!(include_str!("asm/boot.s"));
```

Listing 5.3: Selecting the assembly code.

The `global_asm!` macro used makes it possible to use a whole file as inline assembly. By default (i.e. if no feature is specified), we link the multi-hart version `boot.s`, if the feature flag `sequential` is specified, on the other hand, we link `boot_single_hart.s`.

There is another assembly file in the project that we did not yet discuss: `mem.s`. It's purpose is to store the symbols defined by the linker in a read-only data section, so that they can be accessed by the assembly and Rust code.

5.2.3 The panic handler

Before we can go on with writing Rust code, given that we are in a `no_std` environment, we need to define the behaviour when a panic happens. This has become easy to do in Rust, thanks to the attribute `#[panic_handler]`, that allows us to specify a function that will be called when a panic happens. While in the future we might want to implement some more complex behaviour, at the moment we simply print to the console the cause of the panic (we will discuss the console later), which is automatically passed to the panic handler, and call the `ebreak` instruction, which should stop the program execution.

```
#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    println!("{}", info);
    abort();
}
#[no_mangle]
extern "C" fn abort() -> ! {
    core::arch::asm!("ebreak");
}
```

Listing 5.4: Panic handler.

5.3 Interfacing with the hardware

Now we have all the necessary to start writing Rust code to run on our platforms, however, since we are missing the standard library, we miss some of the functionalities that we will need to run the benchmarks. As we mentioned already, we will not use dynamic allocation, so that is something we do not need to deal with, but we will still need a way to write to a terminal, and a way to measure time.

5.3.1 Writing to terminal

To write (and read) data to a terminal, we need to interface with a hardware component called UART, which stands for Universal Asynchronous Receiver/Transmitter. The QEMU VIRT

machine uses the 16550A UART [36] and its registers are memory mapped, which means that for setting it up and logging to the console, all we need to know is to write to the correct memory address.

For this reason, we create a `Uart` module that deals with initializing it and the input/output. In reality we do not use the input functionalities of the UART, since it is not useful for us, but adding the functionality would be a matter of a few lines of code.

```
pub struct Uart {
    base_address: usize,
}
```

Listing 5.5: `Uart` structure.

First we define the `Uart` structure, that only stores the base address of the UART (Listing 5.5), after which we define a `new` and an `init` method. While the first is trivial, the latter deals with writing the relevant information in the registers for setting up the functionalities we need.

```
pub fn init(&mut self) {
    let ptr = self.base_address as *mut u8;
    unsafe {
        ptr.add(3).write_volatile((1 << 0) | (1 << 1));
        ptr.add(2).write_volatile(1 << 0);
        let divisor: u16 = 592;
        let divisor_least: u8 = (divisor & 0xff).try_into().unwrap();
        let divisor_most: u8 = (divisor >> 8).try_into().unwrap();
        let lcr = ptr.add(3).read_volatile();
        ptr.add(3).write_volatile(lcr | 1 << 7);
        ptr.add(0).write_volatile(divisor_least);
        ptr.add(1).write_volatile(divisor_most);
        ptr.add(3).write_volatile(lcr);
        write!(self, "UART initialized\n").unwrap();
    }
}
```

Listing 5.6: Initializing the UART.

Listing 5.6 shows the operations necessary for the initialization:

1. Set the word length.
2. Enable the FIFO, so that we can write more than one byte at a time.
3. Set the bits for the divisor: this is the value of the ratio between the processors clock and the BAUD rate of the UART.
4. Once the initialization is terminated, we print `"UART initialized"` to the console.

The calculation of the divisor is not very important in this case since this module deals with the emulated UART, however it is necessary for real hardware, so we include it anyways.

As we can see in Listing 5.6, all the operations inside the `init` method are wrapped inside an `unsafe` block: this is because the initialization requires the use of raw pointers to write to the necessary registers. While in this case the use of raw pointers is definitely error prone, since we need to do pointer arithmetic, however being essentially I/O operations there is no owner consideration to be had, which means the code is intrinsically sound.

Another interesting point is the use of the `read_volatile` and `write_volatile` methods: these, similarly to the volatile keyword in C, make sure that the operations will not be removed by the compiler when optimizing the code. This can happen when the compiler notices that we write to a part of memory that we never read again, which is often the case when using memory mapped I/O.

To write to the console we want to be able to use the macro `write!`, as shown in the code just described. To do this, we need our `Uart` struct to implement the trait `Write`. This is useful because it allows us to use the `print` and `format` like syntax, where we pass a format string and a list of arguments, and the compiler will deal with calling the `write_str` function with the correct arguments.

```
impl Write for Uart {
    fn write_str(&mut self, out: &str) -> Result<(), Error> {
        for c in out.bytes() {
            self.put(c);
        }
        Ok(())
    }
}
```

Listing 5.7: Write implementation.

The `put` function called inside `write_str` just writes the byte it receives to the memory mapped UART register at the base address, which is then put into the FIFO and eventually written to the output.

If we were in a single threaded situation, as long as interrupt service routines did not use the output, we could simply use the `write!` macro passing the instance of the UART without much considerations. However, if we want more than one thread to be able to write to the console without their output mixing, we need to have a mechanism for exclusive access to stdout. To achieve this, we define another module called `console`.

The `console` module contains the `Console` structure, that stores the UART, together with an atomic boolean value (see Chapter 3 for more information on Atomics in Rust), so that it can be modified by different threads without causing race conditions.

Since we need a single instance of the console, we use a Singleton pattern, where a user instead of instantiating a new console, uses the `get` method to get an existing one. The first

```rust
pub struct Console {
    uart: Option<uart::Uart>,
    locked: AtomicBool,
}
```

Listing 5.8: `Console` structure.

time the `get` method is called, the `uart` has not yet been stored in the structure, so it is `None`. When this happens, using the lock to make sure it happens only once, the `uart` is initialized and stored in the static variable `CONSOLE`, which is then returned to the caller. Each subsequent time, the `uart` will not be none, so the console will be directly returned.

To illustrate the most basic pattern for sharing resources across threads in a `no_std` environment, we use what is called a `static mut` variable, which is inherently unsafe, since we are sharing a mutable reference without a chance for the borrow checker to assert any property in the sharing. We will see when discussing the parallel algorithms the *correct* way to share a mutable reference, which is quite a bit more complex and still requires unsafe code, but limits its scope to the structure methods.

As in the UART case, we implement the `Write` trait on the `Console` struct: in this case all we need to do is make sure that only one thread will try to write to the UART at once. To do so we use the `locked` atomic stored in the struct to keep track of if some other thread is already using the console. The `compare_exchange` method is used to set `locked` to true if it is not already in an atomic way. The two ordering arguments passed to the method are used respectively in case of success and of failure. As discussed when introducing Atomics, we use the `Ordering::SeqCst` and since this code has no performance consideration it is the ones we use, even though the pattern is a typical one, where we believe an `Acquire`/`Release` ordering would be sufficient.

```rust
fn write_str(&mut self, out: &str) -> Result<(), Error> {
    while self
        .locked
        .compare_exchange(false, true, Ordering::SeqCst, Ordering::SeqCst)
        .is_err()
    {}
    let res = write!(self.uart.as_mut().unwrap(), "{}", out);
    self.locked.store(false, Ordering::SeqCst);
    res
}
```

Listing 5.9: `Console`'s `Write` implementation.

After we exit the while loop, we are sure that only one thread is in the exclusive section of the code, so we can write to stdout without having the threads writing over each other.

One important note here is that we are expecting ISRs not to write to the console, at the

very least if the error is not fatal. If this is not the case, the previous pattern can incur in a deadlock, with the thread being interrupted not releasing the lock and the ISR trying to acquire it. Given that in the actual code we will have a single thread writing to the console, and no real ISR code, this will not be a problem in our case, but it should be kept in mind if we needed to make the code more general.

Since a call to `write!` can fail, we save the result to `res`, so that we can make sure that lock is released before we propagate the error to the caller.

Using `Atomics` gets rid of much of the help that Rust gives us, but this is not avoidable given that we have no access to better patterns from the standard library. It is our responsibility to use the compiler features to create better interfaces: we will see some examples of this when discussing the matrix structures.

Now that we have a console ready, we want to define the macros `print!` and `println!` to offer the typical way Rust code writes to the console. Listing 5.10 shows their code, which is pretty straight forward: in the case of `print!` all that is needed is to call the `write!` macro on the console structure, forwarding the arguments received; as for `println!` we simply pass to `print!` the format string (the first argument) concatenated with a newline and the rest of the arguments. In this case though the macro needs three arms to deal with the cases where no argument is passed, or only the format string is passed.

```
#[macro_export]
macro_rules! print {
    ($($args:tt)+) => ({
        use core::fmt::Write;
        let _ = write!(unsafe {crate::console::Console::get()}, $($args)+);
        });
}

#[macro_export]
macro_rules! println
{
    () => ({
        print!("\r\n")
    });
    ($fmt:expr) => ({
        print!(concat!($fmt, "\r\n"))
    });
    ($fmt:expr, $($args:tt)+) => ({
        print!(concat!($fmt, "\r\n"), $($args)+)
    });
}
```

Listing 5.10: `print` and `println` macros.

5.3.2 Timing the execution

The only other functionality we need to interface with hardware is for timing the benchmarks execution. This is quite easy on the QEMU VIRT machine: a timer is available as a memory mapped device in the Core Local Interrupt module (CLINT), which is more commonly used for Inter-Process Interrupts (IPIs). To obtain the current timer value, we read from address 0xBFF8 of the CLINT set of addresses, which in the VIRT machine starts at 0x2000000. The value we get is in 10^{-7} seconds, so we use the `from_nanos` method and pass `100 * value` to get the `Duration` struct, which will allow us to easily calculate the elapsed time and display the time.

```
use core::time::Duration;
pub fn time() -> Duration {
    let mtime = 0x200_BFF8 as *const u64;
    Duration::from_nanos(unsafe { mtime.read_volatile() } * 100)
}
```

Listing 5.11: `time` function.

5.4 Benchmark data

As in the hosted benchmarks case, our main data structure is the matrix. In this case, to minimize the unnecessary unsafe code, the input matrices are constant, so that they can easily be shared amongst the threads.

Being static structures, their size needs to be known at compile time. To allow for this while at the same time having the ability to have different sized matrices in the same program, we need to use something called **const generics**.

Whereas generics allow to use the same struct with different underlying types, const generics do the same for different values of constants. This means that at compile time, a different structure will be created for each value of the constants that are passed to the structure. A current limitation of const generics in Rust is that we are not allowed to do arithmetics on them, which, as we will see, makes it necessary to have more generic parameters than it would be strictly necessary, making the code quite a bit more verbose.

The basic matrix data structure is very easy and similar to the one used in the hosted environments, with besides the fact that it used a statically allocated array to store the data instead of a `Vec`.

Here we can see the const generics for the side lenght and the size (i.e. the total number of elements). In this case we only support square matrices so it is always the case that `SIZE = SIDE * SIDE`, however as previously mentioned we need the code to specify both. The other

```rust
pub struct Matrix<
    'a,
    const SIDE: usize,
    const SIZE: usize,
    const SECTION_SIZE: usize,
    const N_SECTIONS: usize,
> {
    data: [Number; SIZE],
    rows: usize,
    cols: usize,
    _phantomdata: &'a (), // letting 'a be a lifetime parameter
}
```

Listing 5.12: `Matrix` structure.

two generic parameters and the `_phantomdata` field are used for sharing the matrices, as we will see in the next section.

The matrix has two `const` methods, i.e. methods that return always the same value, to allow for initialization of matrices from slices and full of zeroes.

5.5 Sharing resources amongst threads

Now that we have an environment in which to run the benchmarks, the complexity of working without a standard library becomes the most apparent.

The main problem in this case is that the threads need to have mutable access to the same memory area: in the hosted version of the benchmarks, a thread distributed references to parts of the result matrix where only one thread worked. This is feasible in our case, however we do not have any support from the compiler in checking that our code is correct. This is because, even if one thread was to subdivide the matrix in the correct number of sections, the addresses of the various parts would need to be a known address to all the threads, and the area would need to be mutable to allow the main thread to write to it.

This is a known limitation of embedded Rust [5], and it affects not only multithreaded applications, which are still very much experimental, but also single threaded applications where we need an ISR to have access to some data that can be modified by the main program.

As we discussed when describing the console interface, the easiest way to deal with shared mutable references is to use a `static mut` reference. This however, pollutes the codebase with `unsafe` calls, which increases significantly the code to be validated for memory safety.

A better option is to use an `UnsafeCell`: as the name suggest, operations on it are inherently unsafe, but we will see how it can help with grouping together the unsafe code, which allows easier inspection.

It should be noted that multi threaded code is unsafe also in the standard library. Even with access to the operating system's primitives, it is not possible for the borrow checker to validate it. Looking at any API, for example for `Arc`, will reveal `unsafe` blocks. As such it is not surprising that our code needs unsafe calls. The main difference is that when using the standard library, the user has no need to write unsafe code, and hence to validate it, because the work has been done on the backend. On the other hand without the standard library we are either writing our own unsafe code, or trusting third party implementations. Since our case does not require complex synchronization schemes, we opted for the first option, also because there is not really any multithreaded implementation of critical sections code. This means that the chances of the code having bugs is somewhat high, even though we did not detect any in our testing.

5.5.1 Modifying `Matrix`

As we saw when introducing the `Matrix` data structure, we have two const generics parameters whose purpose was not clear at the time: `SECTION_SIZE` and `N_SECTIONS`. These exist because we want to be able to return references to different sections of the output matrix, so that different threads will be able to modify it at once without writing the same data, which would cause undefined behaviour.

For this reason we create a new structure, `MatrixSection`, which is very similar to `Matrix` but with two differences: the data it contains is not static, and it contains an additional field, `section_number`, which identifies its position inside the original matrix.

```
pub struct MatrixSection<
    'a,
    const SIZE: usize,
    const MATRIX_SIDE: usize,
    const MATRIX_SIZE: usize,
    const N_SECTIONS: usize,
> {
    section_data: &'a mut [Number; SIZE],
    rows: usize,
    cols: usize,
    section_number: usize,
}
```

Listing 5.13: `MatrixSection` structure.

This is the structure on which the algorithms are defined, as we will see later, but for now the only method it has is a `new` method, which simply takes what is passed to it and stores it inside the structure.

We now need a way to create the `MatrixSection`s from the whole `Matrix`. For this reason we create the method `sections_mut`, which we want to operate similarly to the `chunks_mut` from the standard library. It uses the generic parameters `SECTION_SIZE` and `N_SECTIONS`, so that its

output type is known at compile time, and as such can be statically allocated. Since we only allow the `SECTION_SIZE` to be a whole number of rows, and a perfect divisor of `MATRIX_SIZE`, we can infer one from the other, however as already mentioned, Rust does not (yet) allow us to do arithmetic operations with const generics, meaning that we require both parameters.

```
pub fn sections_mut(
    &mut self,
) -> [Option<MatrixSection<'_, SECTION_SIZE, SIDE, SIZE, N_SECTIONS>>; N_SECTIONS] {
    let mut sections: [Option<MatrixSection<'_, SECTION_SIZE, SIDE, SIZE, N_SECTIONS>>;
        N_SECTIONS] = Default::default();
    self.data
        .chunks_exact_mut(SECTION_SIZE)
        .enumerate()
        .for_each(|(i, section)| {
            sections[i] = Some(MatrixSection::new(
                section.try_into().unwrap(),
                self.rows,
                self.cols,
                i,
            ))
        });
    sections
}
```

Listing 5.14: Getting `MatrixSection`s from a `Matrix`.

This, although it may not look like it, is one of the most complex parts of the bare metal implementation discussed so far, and it highlights some of the nuisances of working with generics and statically allocated structures in Rust. It's purpose is simply to return `N_SECTIONS` `MatrixSection`s from the original matrix.

The first problem, which is not visible in the method code because it is specified in the impl block, is that we need the following trait bound: `where [Option<MatrixSection<'a, SECTION_SIZE, SIDE, SIZE, N_SECTIONS>>; N_SECTIONS]: Default`; it specifies that a default array of `Option` type. This is needed for the declaration of `sections` in the beginning of the method (Listing 5.14). You might be wondering why we do not use the simpler syntax `[None; N_SECTIONS]` to create an array of `None Options`. The reason is that the syntax does not work, because the option of `MatrixSection` does not implement `Copy`, which is required for the given syntax. While understandable, `None` should clearly be a special case, since we are not copying any data in doing the initialization, but just specifying that no data is stored in the `Option`. There is an RFC [12], which would introduce the ability to do something similar with the inline `const` syntax, however it is not yet (as of the time of this writing) available on stable Rust.

It should also be noted, that the way we used to circumvent the limitation, actually only works up to 32 elements in the array. In our case this is no big deal, especially considering that in the future this should be easily replaced with easier code, however it means that the code

works only up to 32 harts, which realistically is a lot more that we would need and possible to be found in embedded systems.

But wait, why return `Option`s in the first place? Is it possible for the operation to fail, and even if that were to happen, does it make sense to continue the execution? Really, the operation cannot fail given that the all the parameters are constants, and if the logic to make sure that the constants are compatible is incorrect, this is an unrecoverable error and we should panic (this could happen because of the use of `try_into()`.

The reason why we return an array of options has more to do with the fact that we do not have an allocator, and we cannot have uninitialized areas of memory in safe Rust. The lack of an allocator means that we cannot use the `collect` method to collect the elements generated with the iterator. So we need to create the array `sections` beforehand. However this would mean working on uninitialized memory: sometimes the compiler is able to infer that we initialize the memory exactly once, but in this case it would be complex, since the initialization happens inside an iterator. We know this is the case because we know the relations between `MATRIX_SIZE`, `SECTION_SIZE` and `N_SECTIONS`, but it is far from trivial for the compiler to infer it (even though possible, this is probably linked to why we cannot do arithmetic on const generics).

The use of `Option`s then, allows us to initialize the memory using the default value (i.e. `None`), and then modifing it to be a reference to the `MatrixSection`, which is why we need `sections` to be mutable. The body of the method is then pretty straight forward, we chunk the matrix data in `SECTION_SIZE` sections and create a new `MatrixSection` for each of them, with the appropriate section id.

5.5.2 Interior Mutability

We have gotten to the crux of this chapter, and possibly of parallel bare metal Rust in general. As we discussed in the introduction to this section, dealing with interior mutability with shared data structures is the main complexity of the problem we are trying to tackle. We do not have any of the tools that standard Rust offers us, as they all rely on an allocator and some sort of way to communicate amongst threads. It is then our responsibility to create an unsafe but hopefully sound mechanism to deal with this problem. The reason why we feel relatively comfortable in tackling this problem is that the mechanism we are trying to implement do not require complex synchronization, but rather just a way to share exclusive references, while trying to use the compiler to our advantage in analysing the code.

This is the main reason why we do not use a `static mut` matrix, but create a rather complex structure to deal with our problem. In fact, if we were to use `static mut`, we could easily use the existing `Matrix` structure and have multiple threads have concurrent access to it. This code would be as safe as an equivalent C code, and we would not get any help from the Rust compiler.

```rust
pub struct SharedMatrix<
    'a,
    const SIDE: usize,
    const SIZE: usize,
    const SECTION_SIZE: usize,
    const N_SECTIONS: usize,
> {
    matrix: UnsafeCell<Matrix<'a, SIDE, SIZE, SECTION_SIZE, N_SECTIONS>>,
    sections: UnsafeCell<
        Option<[Option<MatrixSection<'a, SECTION_SIZE, SIDE, SIZE, N_SECTIONS>>;
        N_SECTIONS]>,
    >,
    initializing: AtomicBool,
    initialized: AtomicBool,
    section_available: [AtomicBool; N_SECTIONS],
    computation_completed: AtomicUsize,
}
```

Listing 5.15: `SharedMatrix` structure.

Instead we create a new structure called `SharedMatrix`. After considering quite a few designs, we landed on the one in Listing 5.15. The matrix data is stored in the `matrix` field, which is an `UnsafeCell` of a `Matrix`. It then contains the references to the various sections, which can be uninitialized, so they are stored inside an `Option`, once again itself stored in an `UnsafeCell`, as well as some atomics for dealing with synchronization.

`UnsafeCell` is a smart pointer, that allows us to modify the content while only having a shared reference to it. For this reason, it is inherently unsafe. We will see later how we can do operations on it, but this is necessary to have a static `SharedMatrix`, which is as such accessible to all threads, while the content is not constant.

The structure is designed to allow its user to not deal with synchronization, as the structure itself will do the (busy) waiting when some operation is not yet allowed on it. The public operations on it are only four, one `const fn` which is statically called, and the other three to be called at run time:

- **`new()`**: returns a shared matrix with the `Matrix` passed to it. `sections` is `None` (i.e. uninitialized), `initializing` and `initialized` are both false, `computation_completed` is 0, while `sections_available` is an array of true values, we will see later why.
- **`initialize()`**: this function deals with the initialization of the `SharedMatrix` using a data race. This means that any number of threads (at least one) can call the method without worrying if some other thread already called it, and if that is the case `initializing` will be true, making the function spin until the initialization is completed and then return, since some other thread will deal with the initialization. At the end of the initialization, the thread which was the one to get into the section first will set `initialized` to true, while leaving initializing to true so that no other thread will enter the section. While we

could return instantly without spinning if `initializing` is true, we do not do this for easier timing, and there is no real penalty for this, as it would just mean spinning when trying to compute instead of inside the initialization function.

- **`compute()`**: this method takes the operation (a closure) to be done on a `MatrixSection` and the index of the one to operate on, and executes the operation. The method can do busy waiting if the initialization has not yet completed, or if there is some previous operation on the same section still in progress. It should not be possible for this function to be called if the initialization has not yet completed, however it is better to double check, as atomic operations are prone to bugs and a small change in the APIs could break the code otherwise.

- **`wait_completed()`**: this method is only used for the timing of the operation; all synchronous operations on the matrix do busy waiting while the computation is still in progress, but this method allows the hart to know the computation is completed without the need to call an operation on the matrix.

```
pub fn initialize(&self) {
    if self
        .initializing
        .compare_exchange(
            false,
            true,
            core::sync::atomic::Ordering::SeqCst,
            core::sync::atomic::Ordering::SeqCst,
        )
        .is_err()
    {
        while self.initialized.load(core::sync::atomic::Ordering::SeqCst) == false {}
        return;
    }
    unsafe {
        assert!((*self.sections.get()).is_none(), "API internal error");
        (*self.sections.get()) = Some((*self.matrix.get()).sections_mut());
    }
    self.initialized
        .store(true, core::sync::atomic::Ordering::SeqCst);
}
```

Listing 5.16: Initialization code.

The `compute` method acts as a wrapper of the operation that we want to do on a given `MatrixSection`, as it surrounds the call with two calls to private `SharedMatrix` methods: `get_section()` is used to obtain the section at the index passed, while `notify_completed()` serves to keep track of the completed operations. It is still responsibility of the caller to make sure that a call to compute is done exactly once for each section. If a section is not computed, the program will hang forever when we are trying to do a synchronous operation on it (displaying

it or calling `wait_completed`), while if we call `compute` twice on the same section, the program will panic.

At the moment only one parallel operation is allowed on the `SharedMatrix`: this is clearly a limitation, but the algorithms we implemented until now are possible like so. Still, the design is capable of dealing with more than one operation easily, and we will discuss how if we implement any of the more complex algorithms (e.g. correlation).

To better understand how the design of `SharedMatrix` allows us to write simple code, we can look at an example of some code that uses the structure.

```
extern "C" fn main(hart_id: usize) {
    C.initialize();
    let t = crate::time();

    C.compute(
        |section| {
            // some function on matrix section
            // that we want to benchmark
        },
        hart_id,
    );

    if hart_id == 0 {
        C.wait_completed();
        println!("Time: {:?}", crate::time() - t);
        println!("Result: {}", C);
    }
}
```

Listing 5.17: `SharedMatrix` example.

In Listing 5.17, all the threads (harts in RISC-V) have the same entry point, being the main function. They all call the `initialize` function on `C`, our `SharedMatrix` variable, that is a static variable, which is why the declaration is not shown. They then all start a timer, even though only the one from hart 0 will be used. All the threads then do a computation, each on its section, which is achieved by passing the closure that acts on the section and the id of the section to operate on. At this point, the secondary threads have finished their job, while the main one calls `wait_completed` to make sure that all the computations are terminated, after which it prints the time elapsed between the start and the end of the computation.

We will see in more detail how we can use the structure when describing the benchmarks implementations. It should be noted that, without the timing the code would be even more straight forward, as the timing requires synchronous operations that would not be necessary otherwise.

5.6 The benchmark functions

As with the hosted versions of the benchmarks, we define a trait for each benchmark algorithm. The resulting code is very similar to the hosted version, with two main differences: the function is called on the result matrix rather than the input matrix, and it operates on a section, of type `MatrixSection`, instead of the whole matrix.

This differences are due to the fact, that instead of having the benchmark function spawn the threads, they already exist and each of them calls the benchmark function.

```
pub trait Convolution<
    const MATRIX_SIDE: usize,
    const MATRIX_SIZE: usize,
    const SECTION_SIZE: usize,
    const N_SECTIONS: usize,
    const KERNEL_SIDE: usize,
    const KERNEL_SIZE: usize,
>
{
    fn convolute(
        &mut self,
        a: &Matrix<MATRIX_SIDE, MATRIX_SIZE, SECTION_SIZE, N_SECTIONS>,
        kernel: &Matrix<KERNEL_SIDE, KERNEL_SIZE, 0, 0>,
    );
}
```

Listing 5.18: `Convolution` trait definition.

If we analyze the convolution trait for example, we see that it expects two matrices, `a` and `kernel`, which are the input of the algorithm, as well as a mutable reference to `self`, which should be of type `MatrixSection`. In the future we would want this to return a `Result`, but at the moment we did not want to add any more complexity.

If we now look at the implementation, it is essentially just a wrap of the function `convolute_row` from the hosted benchmarks, where the body of the function is called for each row in the matrix section. That said, it is impossible given the current structure of the code to reuse the function, and we do not consider it worth it to deal with the added complexity given the relatively small chunk of code that would not be duplicated, at the cost of significantly more code to be able to call said functions.

Note that we did not specify different code for the sequential and parallel versions of the benchmarks: this is because the code we present works in both cases, as a `MatrixSection` can be the whole underlying matrix. This allows the code to be much more compact, as the code presented is all the computation code necessary for both versions of the benchmark.

```rust
fn convolute(
    &mut self,
    a: &Matrix<MATRIX_SIDE, MATRIX_SIZE, SECTION_SIZE, N_SECTIONS>,
    kernel: &Matrix<KERNEL_SIDE, KERNEL_SIZE, 0, 0>,
) {
    let kernel_y_radius = (kernel.rows - 1) / 2;
    let kernel_x_radius = (kernel.cols - 1) / 2;
    self.section_data
        .iter_mut()
        .enumerate()
        .for_each(|(i, elem)| {
            let row = (self.section_number * self.rows + i) / self.cols;
            let col = i % self.cols;
            for k in 0..kernel.rows {
                for l in 0..kernel.cols {
                    let y = (row + k) as isize - kernel_y_radius as isize;
                    let x = (col + l) as isize - kernel_x_radius as isize;
                    if (y >= 0 && y < self.rows as isize) &&
                        (x >= 0 && x < self.cols as isize)
                    {
                        *elem += a.data[y as usize * self.cols + x as usize]
                            * kernel.data[k * kernel.cols + l];
                    }
                }
            }
    });
}
```

Listing 5.19: `Convolution` implementation.

5.7 Verification code

We want to be able to validate the result of the algorithms, in particular of the parallel versions, against the reference C code. As we already did this in the hosted versions, all we need to do is to import the crate we defined and call the C functions from our code. This should be very straight forward, however it does pose an added complexity due to the fact that we are cross compiling the code without the help of `rustc` in dealing with the target when using the `cc` crate.

For dealing with this, we need to add two feature flags to the `reference_versions` crate: `std` to specify if the C standard library is available on the target, which we will enable by default, and `riscv-hard-float`, which specifies the floating point ABI gcc should use.

For what concerns the standard library, we do not have much of a choice other than removing functionality when it is not available: in particular this matters when the reference implementations use `math.h`. What we do is simply removing the implementations that need the functions defined in the header, so not allowing for verification in those cases. The alternative would be to allow the user to provide an implementation of the needed libraries, but this is beyond the scope of this thesis. The algorithms ported to the bare metal version do not need this, so the verification is available for them.

The floating point ABI on the other hand is needed by gcc for compiling and linking the code. When `riscv-hard-float` is enabled, we add the flag `-mabi=lp64d` when compiling, which is specific for the RISC-V version of gcc. This is the only ABI we support, as it is what we need for both QEMU and the METASAT platform: it specifies hardware support for both single and double precision floating point types.

```
extern crate cc;
...
fn main() {
    let mut base_config = cc::Build::new();
    let compiler_config = if cfg!(feature = "std") {
        ...
    } else if cfg!(feature = "riscv_hard_float") {
        base_config
            .file("src/cpu_functions.c")
            .define(C_TYPE, None)
            .opt_level(3)
            .warnings(false)
            .flag("-mabi=lp64d")
    } else {
        unreachable!("No valid configuration found")
    };
    compiler_config.compile("reference_algorithms");
    ...
}
```

Listing 5.20: `build.rs` for reference implementations.

This configurations are made relatively easy by using `cc` crate in a build script (Listing 5.20). Finally, when specifying `reference_algorithms` as a dependency in the bare metal version of the code, we need to specify that default feature `std` should not be enables, while `riscv-hard-float` should, as well as re-exporting the feature flag that defines the numeric type to be used.

5.8 The benchmarks

In this section we discuss how we can use what we described until now to run the benchmarks and time their execution. We present both the sequential and parallel versions of the benchmarks, that, as we will see, are very similar and we will see how the APIs of `SharedMatrix` allow us to write simple code, without much consideration of the synchronization between the threads. In particular we use the matrix multiplication benchmark as our case study.

One thing we have not quite figured out yet is how to deal with the matrices initialization, so in the example presented the code includes the matrices content: this is fine for our purposes but it limits the ability for benchmarking the hardware and makes the size of the binaries much larger than it needs to be. We will discuss in Chapter 7 how this problem can be solved and the limitations of our proposed solution.

5.8.1 Sequential versions

After having defined the benchmark function, which we described earlier in this chapter, the sequential code is quite straight forward.

```
...
const SIDE: usize = 4;
const SIZE: usize = SIDE * SIDE;
const A: Matrix<SIDE, SIZE, SIZE, 1> = Matrix::from_slice(A_DATA);
const B: Matrix<SIDE, SIZE, SIZE, 1> = Matrix::from_slice(B_DATA);

#[no_mangle]
extern "C" fn main(hart_id: usize) {
    assert_eq!(hart_id, 0);
    let mut C: Matrix<SIDE, SIZE, SIZE, 1> = Matrix::zeroes();
    let t = crate::time();
    for section in C.sections_mut() {
        section.expect("We expect this to be set").multiply(&A, &B);
    }
    println!("Time: {:?}", crate::time() - t);
    println!("Result: {:?}", C);
}
```

Listing 5.21: Sequential benchmark code.

The two input matrices `A` and `B` are defined as constants, since their content does not need to change. After initializing the ouput matrix to all zeroes, we start the timer and execute the benchmark: we can see that we use a for loop on the sections, even though we initialized the matrix with only one section; this is because the code works also in case more than one section is defined, so we use the more general algorithm. Inside the `for` loop we have an `expect` on the section: this is because `section` has type `Option`, but as mentioned when describing the `sections_mut` function, it should never return `None`, and if it does this is an API error that the user cannot recover, so panicing is the correct behaviour. After the calculation is completed, the execution time and the result are printed to the console.

```
UART initialized
Time: 4.6368ms
Result: Matrix { data: [56, 62, 68, 74, ...], rows: 4, cols: 4, _phantomdata: () }
```

Listing 5.22: Sequential benchmark output.

5.8.2 Parallel versions

Going from the sequential to the parallel code is mostly a matter of changing the type of the result matrix to `SharedMatrix` and using its functions to wrap the parallel operations. Listing 5.23 shows the code where we removed the parts in common with the sequential version. We can see that the main difference is that the calculation on the section is passed to the `compute`

method of `SharedMatrix`, and after the calculation, the main hart calls `wait_completed` before stopping the timer to make sure that we are calculating the correct execution time.

```
...
const SECTION_SIZE: usize = SIZE / N_HARTS;
const N_SECTIONS: usize = N_HARTS;
...
static C: SharedMatrix<SIDE, SIZE, SECTION_SIZE, N_SECTIONS> =
    SharedMatrix::new(Matrix::zeroes());
#[no_mangle]
extern "C" fn main(hart_id: usize) {
    C.initialize();
    let t = crate::time();
    C.compute(
        |section| {
            section.multiply(&A, &B);
        },
        hart_id,
    );
    if hart_id == 0 {
        C.wait_completed();
        ...
    }
}
```

Listing 5.23: Parallel benchmark code.

5.9 Adding Support for the METASAT platform

When we discussed the code required to boot into Rust on the bare metal RISC-V environment we focused on the QEMU VIRT machine. Since we want to be able to benchmark the METASAT platform as well, we need to add support to it in our code. Since the target architecture is the same, this process is not too complex, though we need to deal with the specifics of the hardware on the different platform. This means creating some different code to interface with the hardware, which requires the same three steps described for QEMU, which we will discuss in the following sections.

5.9.1 Modifying the linker script

To help the linker in the generation of the ELF file, we need to create a new linker script for the platform, which we call `metasat.lds`. The only modification necessary is actually in the RAM area definition, which needs to have a different starting address and size to comply with the hardware. The rest of the script does not need any modification, as the current memory organization works fine in both of our supported platforms.

We need though to specify which script should be linked depending on the build command.

```
MEMORY
{
    ram : ORIGIN = 0x00000000, LENGTH = 2048M
}
```

Listing 5.24: RAM area definition in `metasat.lds`.

To simplify the process we create two alias for the build commands for the two platforms, that pass the appropriate file to rustc using the `-Clink-arg` flag. They also pass a feature flag to cargo, which will be useful in the following sections.

```
[alias]
build-virt = "build --target riscv64gc-unknown-none-elf
    --config build.rustflags=['-Clink-arg=-Tbare_metal/src/lds/virt.lds']
    --features virt"
build-metasat = "build --target riscv64gc-unknown-none-elf
    --config build.rustflags=['-Clink-arg=-Tbare_metal/src/lds/metasat.lds']
    --features metasat"
```

Listing 5.25: Build command aliases.

5.9.2 UART module

Our target platforms has more than one UART, however we will use one, that is called APBUART [7]. Its memory mapped registers are available in Table 5.1.

Register	**Address offset**	**Purpose**
Data register	0x0	Read/Write input output characters.
Status register	0x4	Stores information about the UART operation.
Control register	0x8	Stores control information, that is used to specify the behaviour of the UART.
Scaler register	0xC	Defines ratio between the processor's clock and the UART BAUD rate.
FIFO debug register	0x10	Used for debugging purposes.

Table 5.1: APBUART memory mapped registers.

The control register is useful to do the UART setup, however we had trouble when trying to write to it in the FPGA, so instead of writing to it in our Rust code, we expect the register to have been setup properly already (the main thing to do is to enable FIFO operations). The same is true for the scaler register. In our case, all it takes to do the initialization in the hardware is to use the command `forward enable UART2` (which is the UART we are using), after which the contents of the two registers are set to the correct values.

```rust
pub struct Uart {
    base_address: usize,
    status_address: usize,
    _control_address: usize,
}
```

Listing 5.26: APBUART structure.

The two registers we do need to interact with at run time are the data register and the status register. To the data register we write the characters that we want to write, that are then shifted in the FIFO so that they can be written to the output. However, if we do not pay attention to the status of the UART we can overfill the queue, resulting in part of our output going lost. This is why before writing a new character to the data register, we need to check that the FIFO is not full, and this is easily done by checking the value of the **Transmitter FIFO full** flag in the status register. This means that the code of the `put` function is sligthly more complicated, and importantly it can hang while it waits for the UART to empty its queue. In our application this is not a big deal, but if we had to deal with a lot of input/output operations we would need to keep in mind the time that is wasted waiting for a slow peripheral.

```rust
pub fn put(&mut self, c: u8) {
    let ptr = self.base_address as *mut u8;
    unsafe {
        // wait while transmitter queue full
        while (self.status_address as *mut u32).read_volatile() & 0x200 != 0 {}
        ptr.write_volatile(c);
    }
}
```

Listing 5.27: APBUART `put` method.

To select the correct UART module at compile time, we use a feature flag, which we mentioned when talking about the building command aliases. This way, depending on the value of the feature flag, a different path is used to load the UART module, making it easy to change between the two. This is the same way we select which benchmark should be run, and is an easy way to deal with having different files for different compilation configurations.

5.9.3 Measuring time

At this point, the time to develop the project is close to the end, so to allow us to measure the time to execute the algorithm we use the simplest, if not very portable, solution: we know that on our target platform the clock cycle is fixed at 100 MHz, and the mcycle CSR tells us how many clock cycles the processor has gone through, so using the value of the register we can get a *current time*, which we then use to time the execution. As mentioned this solution is not very portable, but it works just fine for timing the benchmarks.

Chapter 6

Results

6.1 Experimental Setup

The evaluation of the code performance is carried out on both ARM and x86 hosted platforms, while preliminary evaluation of the bare metal code is carried out on an FPGA of the METASAT platform. All the evaluated platforms are considered good candidates for upcoming high performance aerospace and avionics systems [35]. This section describes the different platforms characteristics. The platforms selection has been made to add to the evaluations of the existing implementations, however since our code does not make use of GPUs, we only use the multicore CPU on each of the selected platforms.

6.1.1 NVIDIA Jetson AGX Xavier

The NVIDIA Jetson AGX Xavier [20] is an embedded platform from NVIDIA which has 8 CPU cores as well as a GPU. The board has different power-modes (Table 6.1), that affect both the CPU and GPU. In particular we used power-mode 1 and power-mode 2, which maximises the multicore performance of the platform which keeps the board consumption under 15W which has been identified a thermal limit of on-board processing platforms in the GPU4S ESA-funded project [15]. The selected power modes have respectively 2 and 4 CPU cores active and they are same used in similar multicore performance evaluations [35]. The version of the platform we are using has 32 GB of LPDDR4 shared between the CPU and GPU.

The software on the platform is:

- Linux Kernel version: 4.9.140 / L4T 32.3.1
- Ubuntu version: Ubuntu 18.04 LTS aarch64

	Mode		
Property	MAXN	10W	15W
Power budget	n/a	10W	15W
Mode ID	0	1	2
Online CPU	8	2	4
CPU maximal frequency (MHz)	2265.6	1200	1200
GPU TPC	4	2	4
GPU maximal frequency (MHz)	1377	520	670

Table 6.1: NVIDIA Jetson AGX Xavier Power Modes.

- GCC version: 7.5.0 (Ubuntu/Linaro 7.5.0-3ubuntu1 18.04)

6.1.2 AMD Ryzen V1605B

The AMD Ryzen V1605B is part of the Ryzen Embedded V1000 family, which offers high performance platforms with a CPU and a Vega GPU in an SoC.

The V1605B has 4 cores, each with 2 hardware threads each, however in our case we did not see improvements by using 8 threads, so we report our results using 4 threads.

The software used on the platform is:

- Linux kernel: 5.4.0-42-generic
- Ubuntu version: Ubuntu 18.04.5 LTS x86_64
- GCC version: 7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04)

6.1.3 METASAT Hardware Platform

The bare metal platform on which we carry on our testing has been developed in the context of the METASAT Horizon Europe project [16] and it is prototyped on the Xilinx VCU118 FPGA from AMD. It has 4 64-bit RISC-V cores, in particular the NOEL-V space processors from FrontGrade Gaisler [8] . Each CPU has 16KB of L1 cache, while the L2 cache is unified across the cores. The design also includes a RISC-V GPU, however we do not use it for our testing. A more detailed description of the METASAT hardware platform, which will be released as open source at the end of the project can be found in [17].

The platform is of particular interest as it is designed for the applications that the benchmarks are focused on, which is why we include the preliminary results in the document even though we do not have extensive testing on it as in the rest of the platforms.

6.2 Performance Results

In this section we present the performance results of the Rust code compared to the C implementations. We report the result on the AMD Ryzen V1605B and the NVIDIA Jetson AGX Xavier, so that we have two different architectures. All the benchmarks in this section, unless differently specified, use a 4096 size and a single precision floating point type, which means that for benchmarks that operate on vectors rather than matrices (FFT, FIR FFT windowed) the number of elements is very small. The decision on which size and datatype to use is for consistency with the standard sizes defined for the GPU4S Bench/OBPMark Kernels Benchmarking Suite [41][4], which mainly targets existing space processors which have significantly lower performance and memory. Moreover, the same sizes have been used in multicore evaluation of the same platforms using the same benchmarks presented in [35] under the RTEMS SMP space qualified operating system in sequential C and OpenMP, which are used for comparison with our sequential and parallel Rust implementations.

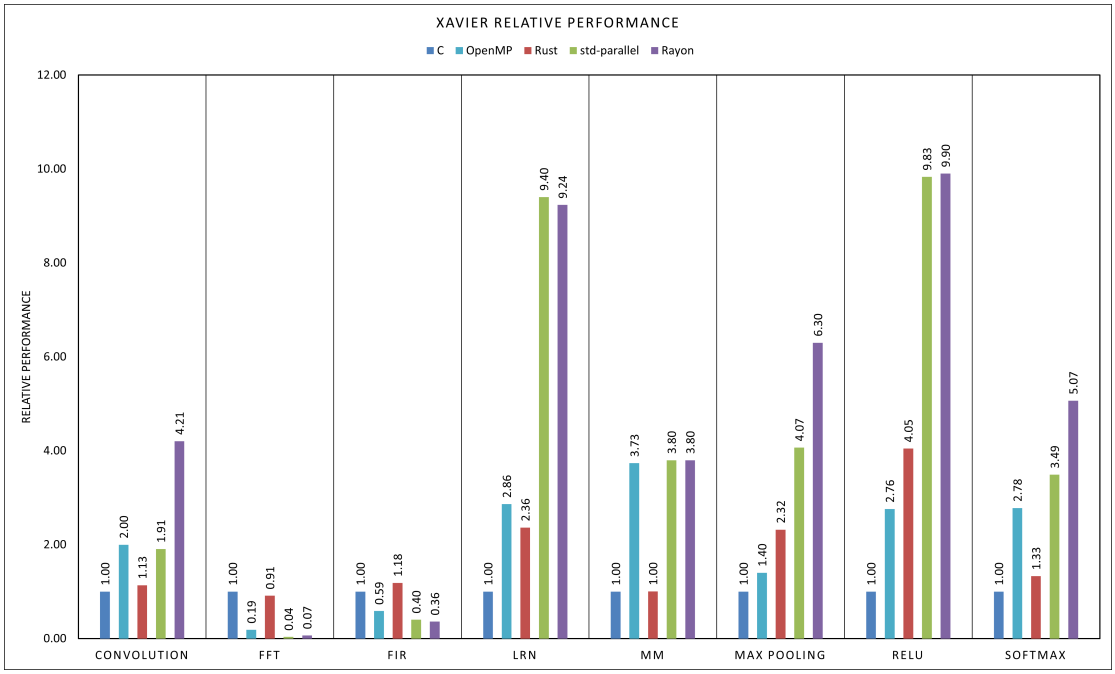

Figure 6.1: Performance comparison on the Xavier in power-mode 2 of the implementations of the algorithms. The results are normalised to the sequential version, which is shown as 1x.

The result on the the Xavier in power-mode 2 (so using 4 cores in the parallel benchmarks) are shown in the graphs in Figure 6.1. If we compare the performance of the sequential versions in C and Rust, we see a few different cases:

- In FFT and Matrix Multiplication the performance of the two implementations is very

close, in the case of Matrix Multiplication almost identical. The small differences are probably attributable to slightly different optimization between LLVM, which is used by Rust and GCC, which is used by RTEMS or different memory arrangements between the different allocators.

- In Convolution, LRN, Max Pooling, Relu and Softmax the performance of the Rust sequential versions is from 10% all the way to 130% better than that of the C version. This clearly cannot be just slight differences in the compiled code, but requires the code to use significantly different calculations. A few hints have led us to believe that the difference has to be due to the introduction of more vector operations in the Rust compiled code. In particular, while we were not able to identify the specific functionalities of the instructions in the disassembled code, we saw an increase in the number of such operations in the code, as well as in some cases we see the verification to have to be with a tolerance to pass, which suggests that the floating point operations are carried out in some different order compared to the C version.
  While in the case of Convolution and Softmax the performance difference is not that large, we are convinced there is a real difference in performance due to the difference in the results for the 2 different parallel implementations. We will discuss this in more detail when analysing the parallel versions.

- In FIR (and perhaps FFT) the performance difference measured is hard to quantify with confidence given the very short execution time of the benchmarks, which is in the 0.5 ms range.

If we now look at the parallel implementations, we see some interesting differences in performance. First, let's look at the short execution time benchmarks, FIR and FFT: the parallel versions in this case are slower than the sequential version, and this is true both for the Rust and the OpenMP code. This is likely due to the overhead of spinning up the different threads which is not worth the small improvement in execution time. In FFT in particular, we are not actually able to parallelize the code due to complex task dependencies which are not supported in Rayon. Similarly, the OpenMP version of the code uses homogeneous parallelism (i.e. using parallel for) instead of the OpenMP tasking model, because it provides easier certification for multicore contention in aerospace systems [35].

This means that for the parallel code we use the windowed version of the FFT benchmark as in [35], which is much easier to parallelise with homogeneous parallelism, and offers a way to get to an approximated result. This however shows promise for larger vectors, where we can see an actual speedup. The reason for the choice of the sizes is to use the same values from the testing that has been done on the C versions. It should also be noted that also the windowed algorithm cannot compete with the library implementation in FFTW, which is highly optimised, making it perhaps not a good candidate for parallelization.

For Matrix Multiplication, we can see that, as in the sequential case, both Rust parallel implementations have extremely similar performance to the OpenMP version, which makes Matrix Multiplication the most consistent benchmarks between the C and Rust implementations, with a very good 3.8x speedup on 4 cores.

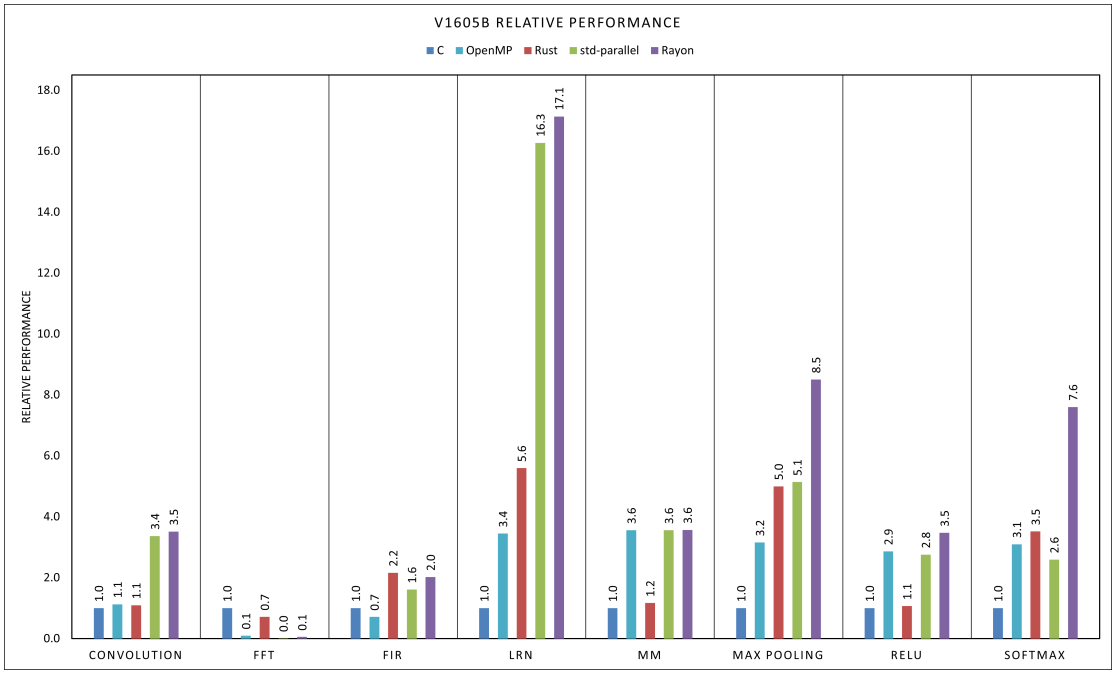

Figure 6.2: Performance comparison on the AMD V1605B of the implementations of the algorithms. The results are normalised to the sequential version, which is shown as 1x.

Looking at the AMD platform results we can see that in some cases the results are quite similar to the Xavier platform, while in some other we have very different relative performance. Since we are not particularly interested in the performance difference amongst the platforms we do not report the execution times. However it should be noted that the V1605B has much higher performance, with the benchmark taking usually anywhere from half to 1/5 to complete execution compared to the Xavier. The behaviour of Matrix multiplication is still very consistent amongst the different implementations, with speedups that approach the linear case both for OpenMP and the parallel Rust implementations. Similar to the Xavier, LRN, Max Pooling and Softmax show much higher sequential performance compared to the C version, in this case performing even better than the OpenMP version, once again thanks to a higher use of vector instructions. Relu, on the other hand, is quite close to the C versions in this case, both in the sequential and parallel execution, with a slight upper hand of the Rayon code. FFT is significantly slower in the parallel versions on the AMD platform too, for the same reasons discussed above, while FIR manages to have better sequential performance compared to the C code, but the parallel code is still slower than the sequential version due to the small sizes on the input.

6.2.1 Taking a closer look to the parallel implementations

As mentioned in the previous section, the two different parallel implementations of the Rust code can have quite different performance depending on the benchmark.

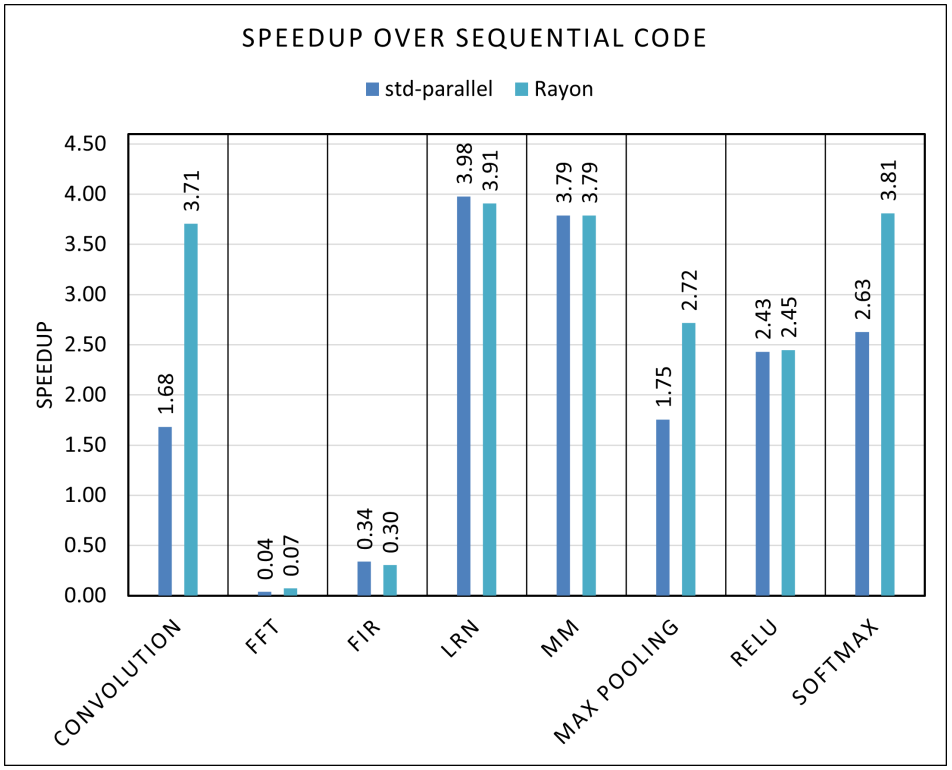

Figure 6.3: Comparison of the speedup of the parallel Rust implementations on the NVIDIA Xavier.

In Figure 6.3 we see the speedup over the sequential code of the rayon and std-parallel implementations, which seems to favor significantly Rayon, in particular where the vector instructions made the sequential code faster. Our guess to the cause of this behaviour, is that Rayon is does a better job for keeping the vector operations, as the library can decide, knowing the hardware features available, if it makes sense to make something parallel and to which thread assign each part of the matrix, while in the std-parallel the programmer takes this decision, which can result in sub optimal performance. This is true in particular in the Convolution and Softmax benchmarks, where the std-parallel code has similar performance to the OpenMP one, even though the Rust sequential version is faster. In LRN and Relu this is not the case, with both the std-parallel and Rayon version showing very impressive speed-ups compared to the sequential C version.

In the AMD results shown in Figure 6.4, we can see somewhat similar results, but here the std-parallel version of Softmax is slower than the sequential code and in Max Pooling it has a very similar performance, while on the ARM platform we could still see a speedup over the sequential code. As mentioned before, this is likely because the manual subdivision of the

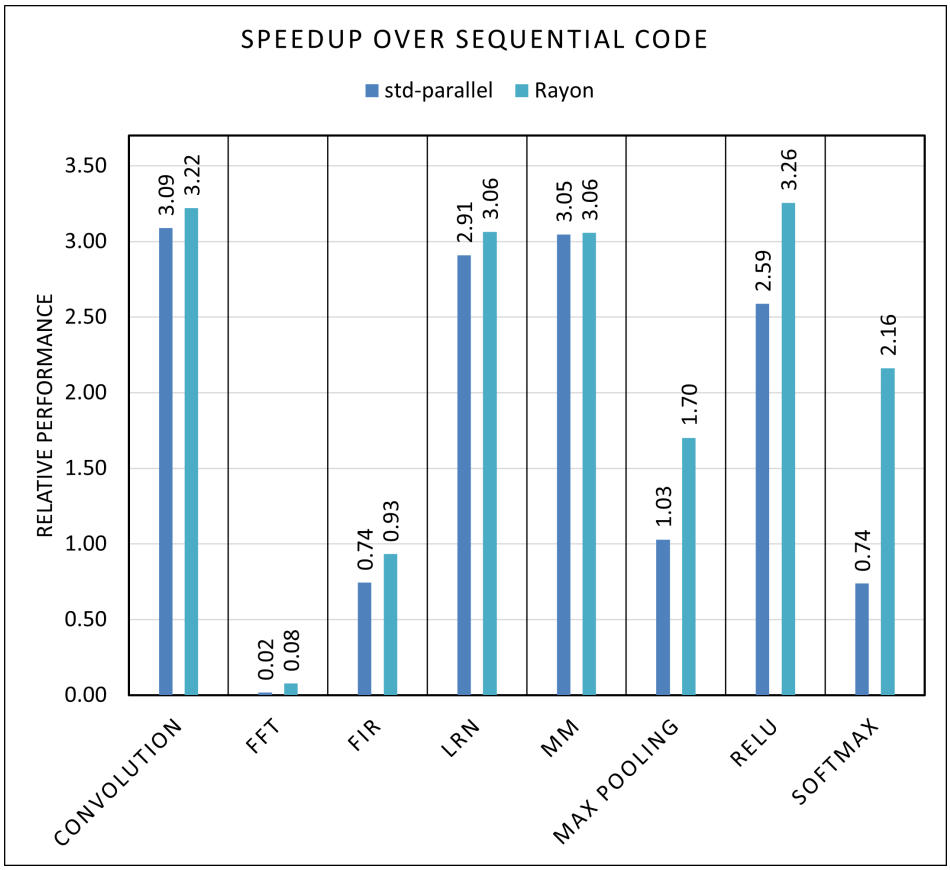

Figure 6.4: Comparison of the speedup of the parallel Rust implementations on the V1605B.

input and output matrices to make the parallelization possible interferes with the ability of the compiler to introduce vector instructions, while the more complex Rayon runtime is able to still utilize them. On the other hand, both Rayon and std-parallel manage to perform very well in the LRN benchmark, on both architectures.

Another difference with the performance on the Xavier is that on the V1605B the speedup does not go much higher than three, while in the Xavier case we had some cases, like LRN and Matrix Multiplication, where the performance was close to the theoretical maximum.

6.2.2 2D matrices

Until now we mostly considered the 1D version of the Matrix structure to have a better comparison with the C code, however as we mentioned, we developed also a 2D version which improves on programmability and decreases bugs when manually dealing with identifying rows.

Figure 6.5 shows the speedup of some 2D Matrix algorithms over their 1D counterparts, and as we can see in general there is a performance deficit by having the rows allocated independently from each other. It seems to be the case that this deficit is not the same for all benchmarks,

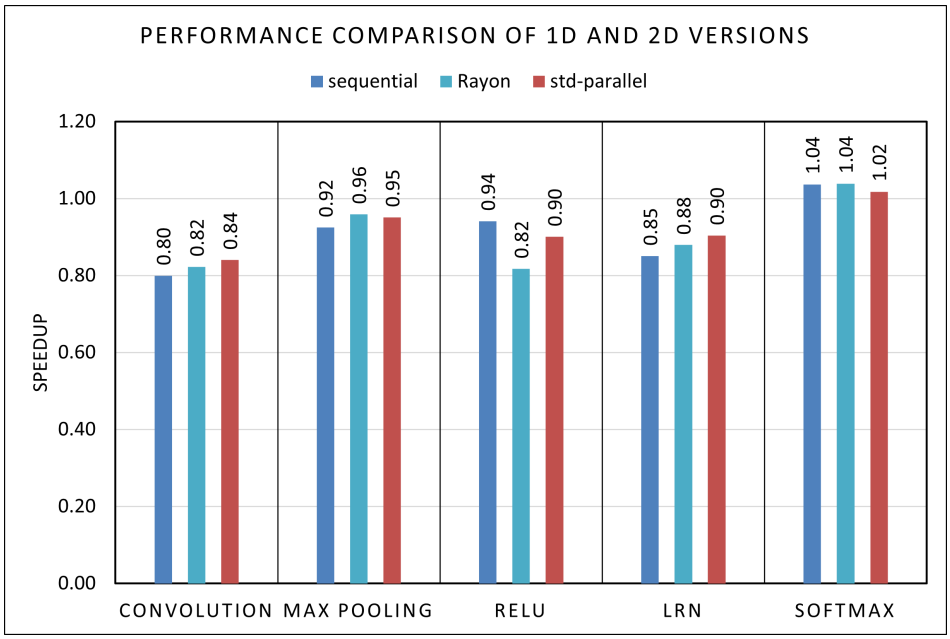

Figure 6.5: Speedup of the 2D matrix over the 1D version on the AMD V1605B.

with Softmax actually showing an improvement, though perhaps not statistically significant. In our results the difference is usually pretty small, making the 2D matrix probably preferable in situations where we don't care about very small performance differences. It should be noted though that the difference in the real world *could* be higher, if the structures are allocated during the execution rather than at the start of the program, or with more programs running at the same time, as the OS could place the different rows far away from each other, increasing the cache misses.

6.2.3 Preliminary METASAT Platform results

Due to limited time available with the FPGA, the bare metal results available on the METASAT platform are limited, but very promising. Note that QEMU could not be used for performance evaluation, since it only models the functionality of a RISC-V platform, but not emulates its cache memory hierarchy or the execution time latency of its instructions.

Table 6.2 compares the sequential and parallel versions of the bare metal Rust implementation on the METASAT platform, where we notice a 3× speedup over the single core execution, which is similar to the speedup of the same benchmark on the AMD V1605B on top of Linux.

	sequential	**parallel (4 cores)**	**speedup**
Matrix Multiplication 256x256	9435 ms	3141 ms	3.00x

Table 6.2: METASAT Matrix Multiplication Results with Rust.

We also compared the parallel results of the Rust code to the OpenMP code running under RTEMS [35], which is a low-overhead real-time operating system. The results show that the Rust code is slightly slower but has similar performance to the C code.

	Rust parallel	**OpenMP**	**OpenMP speedup**
Convolution 1024x1024	1568 ms	1318 ms	1.19x
Convolution 2048x2048	6612 ms	5200 ms	1.27x

Table 6.3: Comparison between parallel Rust code and OpenMP code on the METASAT platform.

In the future, we plan to perform a more detailed performance evaluation and comparison between C and OpenMP under RTEMS SMP.

Despite the actual performance obtained with Rust, we have to highlight that currently, the only way to execute parallel code in the METASAT platform or any other embedded platform is through RTEMS SMP and OpenMP. In particular, it is not possible to run multicore C code or OpenMP on a bare metal environment. However, with our thesis contribution, it is possible to run parallel code in Rust on a bare metal RISC-V environment.

Chapter 7

Conclusions and Future Work

The results presented in the previous chapter show that the performance of sequential Rust is similar to C in our space-relevant applications. The same holds for the parallel versions, with Rayon showing some of the most promising results both in terms of ease of use and performance. These findings together with the memory safety and easier development of Rust make it a promising technology in the space and other safety critical domains, as well as in embedded systems in general.

Another take away comes from the 2D version of the algorithms, that perform only slightly worse but help a lot with programmability. Throught the development we caught bugs on the 1D version that stemmed from inadequate testing, as the benchmarks only use square matrices, due to the use of the incorrect dimension in iterators; these bugs did not happen in the 2D versions as the there is no need to manually divide the structure in rows.

Similarly, it is worth noting that during the multicore evaluation performed in our group for the [35] publication, a couple of software defects (out of bounds accesses and incomplete initialisation) were found in the C and OpenMP implementations of the FIR GPU4S benchark, which manifested with a crash only on the GR740 space processor under the RTEMS SMP real-time operating system. These latent defects were masked in all other hardware platforms and operating systems combinations. After investigating and correcting these defects in the GPU4S Bench official repository, we checked whether these defects were also present in our Rust port, as well as in the Ada SPARK versions performed in [1]. Interestingly, these defects were not present in the GPU4S Bench ports in these two safe languages [14], since both languages prevent uninitialised memory and out-of-bound accesses.

Rust has a reputation of being a hard language: we would agree that the learning curve can be somewhat steep in the beginning, but in our opinion the very thing that makes Rust hard, i.e. the compile time checks, is what can make the programmer a lot more confident in the resulting

code, since once it compiles we know we are not going to get any segmentation faults. This is even more the case in parallel code, where there is much lower risk of forgetting to release a lock, introducing very hard to debug errors and race conditions.

When compared with OpenMP the comparison in ease of parallel code development is less one sided, especially with the wide use of OpenMP in many applications and the larger feature set compared to Rayon. Still, as we saw in Chapter 4, the parallel code is very easy to obtain from the sequential one and the results are very good, making Rust a viable option.

On the bare metal side, we provided a proof of concept of what could be achieved, knowing that there is a lot more work to do. In particular it would be interesting to have a version of the code we developed running on SBI as well as on top of RTEMS, to compare the performance and ease of development of the different options. A version using the RISC-V runtime crate would also be useful, as it could help by running some code before getting to main, which would be useful for memory initialization. In general our approach has been to have the safest possible solution, which has limited the functionality of the code.

Another area where we would like to see further development is in libraries for mathematical abstraction: we considered using some crates for mathematical operations, but did not find one that satisfied our requirements, partly due to limited support. We created the `Number` trait to deal with some of these problems, and we think a package that can help with this would be instrumental for the use of Rust in high performance parallel applications.

Appendix A

Borrow checker false positives

An example of some difficult code to write for Rust was the wavelet transform. The sequential code is straight forward (a direct translation of the C code). However, trying to separate the code into functions that can be called from the parallel code, we ran into a problem.

The wavelet transform code consists of 2 loops:

- The first one, calculates the top half of the result vector using only the input vector.
- The second loop, after the first one has terminated, uses the top half of the result vector and the input vector to calculate the bottom half of the result.

This second loop is where the problems arose. When extracting a function that can be called in a loop for parallelization, the function to be called inside the second loop requires an immutable reference to the top half of the result vector (it only needs to read the top half), while the loop (or iterator) requires a mutable reference to the bottom half of the vector (that is being written over). We can clearly see that the writing and reading are happening over different sections of the vector, so from a logical point of view there is no trouble with the code. However, making this understandable to the borrow checker is not trivial, without using external libraries.

The main difference with other benchmarks is that in the other cases, we required a single reference (mutable) for each thread (or function call) to the result, while in this case the single function call needs both an immutable reference to the result, and needed to be called from a context were we have access to a mutable reference so as to modify the result vector.

The only solution, if we want to avoid using unsafe code, is for the immutable reference to be to a copy of the top half of the result vector, rather than the result itself. This is possible in our case, and the performance loss is not that high (trivial for small vectors, up to about 10-15% for a million elements of more), however this would not be the case if, for instance:

- Each call took less time, but the function was called a very large number of times.
- The type contained in the vector was not `Copy` (i.e. if it was not possible to get a copy of the value, which is true for most non fundamental types).
- The structure was big enough to make the added memory required not feasible.

While we could use unsafe code here, it is intentionally not that straight forward to do so. In fact it is not just a matter of wrapping the incriminated code in an unsafe block, but rather it requires the internal code to use raw pointers, which would increase the potential for bugs, which is one of the reasons we did not go this way.

Bibliography

[1] Dimitris Aspetakis. "Evaluation of the Ada SPARK Language Effectiveness in Graphics Processing Units for Safety Critical Systems". https://upcommons.upc.edu/handle/2117/390672. Bachelor's Thesis. Universitat Politècnica de Catalunya, May 2023.

[2] Dimitris Aspetakis. *GPU4S Benchmarks Ada SPARK port*. URL: https://gitlab.bsc.es/dimitris_aspetakis/gpu4s-bench-ada.

[3] *clap - Rust Package Registry - crates.io*. URL: https://crates.io/crates/clap.

[4] D. Steenari et al. *On-Board Processing Benchmarks*. http://obpmark.github.io/. 2021.

[5] *Determine an idiomatic way of sharing or transferring resources between Interrupt and User contexts - Issue #294 - rust-embeddedwg*. URL: https://github.com/rust-embedded/wg/issues/294.

[6] *funty - Rust Package Registry - crates.io*. URL: https://crates.io/crates/funty.

[7] Cobham Gaisler. *GRLIB IP Core User Manual*, pp. 130–139. URL: https://www.gaisler.com/products/grlib/grip.pdf.

[8] Frontgrade Gaisler. *NOEL-V Processor*. https://www.gaisler.com/index.php/products/processors/noel-v.

[9] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612. URL: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

[10] *half - Rust Package Registry - crates.io*. URL: https://crates.io/crates/half.

[11] Tony Hoare. *Null References: The Billion Dollar Mistake (Presentation Abstract)*. QCon London on 28 June 2009. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/.

[12] *Inline Const RFC*. URL: https://rust-lang.github.io/rfcs/2920-inline-const.html.

[13] *Integer Overflow Basics - gnu.org*. URL: https://www.gnu.org/software/autoconf/manual/autoconf-2.63/html_node/Integer-Overflow-Basics.html#:~:text=In%20contrast%2C%20the%20C%20standard,%2C%20division%2C%20and%20left%20shift.

[14] Leonidas Kosmidis, Dimitris Aspetakis, and Matina Maria Trompouki. *Formal methods for GPU software development using Ada SPARK, presentation at the ESA Software Product Assurance Workshop 2023*. URL: https://www.cosmos.esa.int/documents/10939403/13962862/3_updated_Leonidas_Kosmidis_2023_Software+Product+Assurance+Workshop+2023_Formal_Methods_GPUs+(1).pdf.

[15] Leonidas Kosmidis et al. "GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021.

[16] Leonidas Kosmidis et al. "METASAT: Modular Model-Based Design and Testing for Applications in Satellites". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation - 22nd International Conference (SAMOS)*. Lecture Notes in Computer Science. 2023.

[17] Leonidas Kosmidis et al. "The METASAT Hardware Platform: A High-Performance Multicore, AI SIMD and GPU RISC-V Platform for On-board Processing". In: *European Data Handling and Data Processing Conference for Space (EDHPC)*. 2023.

[18] Stephen Marz. *The Adventures of OS: Making a RISC-V Operating System using Rust*. URL: https://osblog.stephenmarz.com/index.html.

[19] *num_traits - Rust Package Registry - crates.io*. URL: https://crates.io/crates/num_traits.

[20] *NVIDIA Jetson Xavier Series — nvidia.com*. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.

[21] *Optimization and Wraparound - gnu.org*. URL: https://www.gnu.org/software/autoconf/manual/autoconf-2.63/html_node/Optimization-and-Wraparound.html#Optimization-and-Wraparound.

[22] Cristina Peralta. *GPU4S Benchmarks SYCL port*. URL: https://github.com/crispq95/GPU4S_Bench.

[23] Cristina Quesada Peralta, Matina Maria Trompouki, and Leonidas Kosmidis. "Evaluation of SYCL's Suitability for High-Performance Critical Systems". In: *Proceedings of the 2023 International Workshop on OpenCL*. IWOCL '23. Cambridge, United Kingdom: Association for Computing Machinery, 2023. DOI: 10.1145/3585341.3585378. URL: https://doi.org/10.1145/3585341.3585378.

[24] Cristina Peralta Quesada. "Evaluation of High-Level Programming Models for High-Performance Critical Systems". https://upcommons.upc.edu/handle/2117/380697. Master's Thesis. Universitat Politècnica de Catalunya, Oct. 2022.

[25] Alberto Perugini. *GPU4S Benchmarks Rust port*. URL: https://gitlab.bsc.es/aperugin/gpu4s_rust.

[26] *Pre-RFC: Cargo mutually exclusive features - internals.rust-lang.org*. URL: https://internals.rust-lang.org/t/pre-rfc-cargo-mutually-exclusive-features/13182.

[27] *Procedural Macros - The Rust Reference - doc.rust-lang.org*. URL: https://doc.rust-lang.org/reference/procedural-macros.html.

[28] *QEMU RISC-V VirtIO Board - github.com/qemu*. URL: https://github.com/qemu/qemu/blob/master/hw/riscv/virt.c.

[29] *RAII - cppreference.com*. URL: https://en.cppreference.com/w/cpp/language/raii.

[30] *rayon - Rust Package Registry - crates.io*. URL: https://crates.io/crates/rayon.

[31] *RISC-V Supervisor Binary Interface Specification*. 2022. URL: https://www.scs.stanford.edu/~zyedidia/docs/riscv/riscv-sbi.pdf.

[32] *riscv_rt - Rust Package Registry - crates.io*. URL: https://crates.io/crates/riscv_rt.

[33] Ivan Rodriguez et al. *GPU4S Bench: Design and Implementation of an Open GPU Benchmarking Suite for Space On-board Processing*. Tech. rep. UPC-DAC-RR-CAP-2019-1. https://www.ac.upc.edu/app/research-reports/public/html/research_center_index-CAP-2019,en.html. Universitat Politècnica de Catalunya.

[34] Rust on Embedded Devices Working Group et al. *The Embedded Rust Book*. URL: https://docs.rust-embedded.org/book/.

[35] Marc Solé et al. "Evaluation of the Multicore Performance Capabilities of the Next Generation Flight Computers". In: *Digital Avionics Systems Conference (DASC)*. 2023.

[36] Semiconductor Design Solution. *UART 16550 Manual*. URL: http://caro.su/msx/ocm_de1/16550.pdf.

[37] *std::collections - Rust - doc.rust-lang.org*. URL: https://doc.rust-lang.org/std/collections/index.html.

[38] *std::iter - Rust - doc.rust-lang.org*. URL: https://doc.rust-lang.org/std/iter/trait.Iterator.html.

[39] *std::memory_order - cppreference.com*. URL: https://en.cppreference.com/w/cpp/atomic/memory_order.

[40] David Steenari et al. "OBPMark (On-Board Processing Benchmarks) - Open Source Computational Performance Benchmarks for Space Applications". In: *2nd European Workshop on On-Board Data Processing (OBDP)*. 2021. DOI: 10.5281/zenodo.5638577. URL: https://doi.org/10.5281/zenodo.5638577.

[41] David Steenari et al. "OBPMark and OBPMark-ML – On-Board Processing Computational Benchmarks for Space Appliactions and Results". In: *European Data Handling and Data Processing Conference for Space (EDHPC)*. 2023.

[42] Carol Nichols Steve Klabnik. *The Rust Book*. URL: https://doc.rust-lang.org/book/.

[43] Aaron Turon. *Fearless Concurrency with Rust — Rust Blog - blog.rust-lang.org*. 2015. URL: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html.

[44] *What Every Computer Scientist Should Know About Floating-Point Arithmetic - docs.oracle.com*. [Accessed 02-09-2023]. URL: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

[45] *Why Discord is switching from Go to Rust - discord.com*. URL: https://discord.com/blog/why-discord-is-switching-from-go-to-rust.