Towards GPU Parallelism Abstractions in Rust: A Case Study with Linear Pipelines

Leonardo Gibrowski Faé
School of Technology
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, Brazil
leonardo.fae@edu.pucrs.br

Dalvan Griebler
School of Technology
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, Brazil
dalvan.griebler@pucrs.br

ABSTRACT

Programming Graphics Processing Units (GPUs) for general-purpose computation remains a daunting task, often requiring specialized knowledge of low-level APIs like CUDA or OpenCL. While Rust has emerged as a modern, safe, and performant systems programming language, its adoption in the GPU computing domain is still nascent. Existing approaches often involve intricate compiler modifications or complex static analysis to adapt CPU-centric Rust code for GPU execution. This paper presents a novel high-level abstraction in Rust, leveraging procedural macros to automatically generate GPU-executable code from constrained Rust functions. Our approach simplifies the code generation process by imposing specific limitations on how these functions can be written, thereby avoiding the need for complex static analysis. We demonstrate the feasibility and effectiveness of our abstraction through a case study involving linear pipeline parallel patterns, a common structure in data-parallel applications. By transforming Rust functions annotated as source, stage, or sink in a pipeline, we enable straightforward execution on the GPU. We evaluate our abstraction's performance and programmability using two benchmark applications: `sobel` (image filtering) and `latbol` (fluid simulation), comparing it against manual OpenCL implementations. Our results indicate that while incurring a small performance overhead in some cases, our approach significantly reduces development effort and, in certain scenarios, achieves comparable or even superior throughput compared to CPU-based parallelism.

KEYWORDS

Programming Languages, Domain-Specific Language, OpenCL, Procedural Macros

1 Introduction

As the rate at which data is created and must be processed grows, programmers have begun to resort to parallel computation to cope with these requirements [7]. A Graphical Processing Unit (GPU) is a specialized hardware tailor-made to consume large amounts of data and execute massive computations in parallel. Though they are called "graphical", they have for a long time been used for general-purpose computation, where they are referred to as General Purpose GPUs (GPGPUs) [15]. GPGPUs are used to accelerate workloads in current artificial intelligence developments [5], mathematical computing [24], and image processing [26].

Programming GPUs can be challenging, however, as they operate under a fundamentally different programming model than Central Processing Units (CPUs) [27]. Typically, one must learn either CUDA [28] or OpenCL [19], with all the intricacies of CPU-GPU interaction. There is a long history of work that has tried to simplify this, such as HIP [1], Kokkos [39], GSParLib [32], GPotion [9], and so on. Many of these works are geared towards C++ (though GPotion is an Elixir library) and approach the problem through different angles, such as creating libraries or relying on code generation. Automatic code generation for executing code on the GPU has been explored in works like SPar [13, 33]. These works tend to run into limitations when it comes to static analysis when they try to adapt the semantics of a language designed for the CPU (in this case, C++) to one that must be executed on the GPU.

In the meantime, the programming language Rust came to be as a modern low-level programming language promising safety and performance [23]. Its capabilities for high-performance computing has already been demonstrated in many recent works [4, 11, 12, 29–31, 38]. Also in recent times, Rust's potential to substitute C/C++ has also been demonstrated in other areas, such as its acceptance in the Linux kernel [8]. This is driven by the understanding that Rust avoids a whole category of bugs through its safety guarantees, assuring that no program written in Rust without the use of the "`unsafe`" keyword has undefined behavior.

Since Rust has been encroaching on many areas traditionally occupied by C/C++, many are also trying to use it in the GPU space [10, 34]. In this scenario, we propose a new Rust abstraction, relying on the language's procedural macros to automatically generate GPU code from Rust code. Our approach differs from many previous works because it does not attempt to perform advanced static analysis or compile regular Rust to a GPU-executable format. Instead, we limit upfront the types of code that the user can write, and use those limitations to greatly simplify the code-generation process. Our procedural macros look very similar to function annotations in other languages. More specifically, our main contributions are:

- A new high-level programming abstraction in Rust for executing certain functions on the GPU.
- An approach that is based on limiting how the functions to be transformed can be written, so that the code generation can be very straightforward and does not need to involve complex static analysis. This basic idea could be transferred to other programming languages.
- A comparative analysis of programmability and performance using two benchmark applications. We use two implementations: a manual one and one using our proposed abstraction.

We use the linear pipeline parallel pattern as a case study for our approach. Our abstraction will create a processing pipeline

whose stages will run on the GPU. We chose the linear pipeline because it is conceptually simple but still generally useful in practical applications.

This article is organized as follows. Section 2 contains the essential theoretical background to understand our research. Section 3 contains the full explanation of how our programming abstraction works, followed by an example taken from one of our testing applications. In Section 4, we show performance and programmability metrics that indicate our programming abstraction reduces development effort while incurring some measurable, albeit small, overhead. We discuss related work in Section 5 and finalize with our conclusions in Section 6.

2 Background

In this Section, we outline the most important concepts necessary to understand this work. These include basic notions of the Rust programming language, GPU programming, and the linear pipeline parallel pattern.

2.1 Rust

Rust is a modern, low-level programming language with a focus on performance and safety. As a language, Rust has many features, but we will focus on the two most important ones to understand this work: safety, which many would claim is one of Rust's main selling points, and procedural macros, which are the mechanism we will use to do code generation.

2.1.1 Safety. Rust's safety guarantees can be summarized in one sentence: *as long as the keyword "unsafe" is not used, if the program compiles, then it does not have undefined behavior* [23, 37]. Rust ensures this through both statically-enforced rules, such as its borrow checker and scoped-bounded lifetimes, and runtime checks, such as verifying we are not indexing an array beyond its bounds. This means the program may panic, but it will panic every time, as opposed to C/C++, where it may raise a segmentation fault in some executions, but not others. A full purview of what safe Rust entails and how safe and unsafe Rust interact is outside the scope of this paper. However, it is important to keep in mind that safety (i.e., the lack of undefined behavior) is one of Rust's main selling points, and therefore it is generally encouraged to use as little unsafe Rust as possible. Typically, one would profile the program to see if any regions of code would benefit from, for example, bypassing bound checks on array accesses, and only then resort to using unsafe. Of course, in certain low-level programs, unsafety is inevitable, such as when the programmer must make use of inline assembly to attain the exact results they desire, as may happen sometimes in operating systems or drivers. In this work, we use almost no unsafe code. The main exception is in the code we generate for the functions that interact directly with the GPU. Since they are marked as unsafe functions, they can only be called within an unsafe block.

2.1.2 Rust Procedural Macros. Rust has two types of macros: procedural and declarative. Declarative macros perform advanced text substitution. Though they are quite versatile, they are not as permissive as C/C++ macros. For example, they can not use variables that are not available in the scope where the macro was defined. Procedural macros (proc-macros), on the other hand, have no limitations in the code they may generate. When the programmer makes a call to a proc-macro, the compiler calls upon a small program (developed by the proc-macro's creator) and feeds it the proc-macro's argument as raw Rust tokens. These tokens may then be parsed and processed arbitrarily by the proc-macro's implementation. There are 3 kinds of procedural macros:

(1) Function-like — these look like functions, except they must be called with an extra `!` token. An example is the print macro from the standard library: `println!(...)`. It is important to keep in mind that the arguments this function-like macro accepts are parsed as *raw Rust tokens*. Type checking is only done after the proc-macro has already generated the code.
(2) Derive — these are procedural macros that can be used in `#[derive(...)]` annotations. These annotations can be placed in a `struct`, an `enum`, or a `union`. Their primary use case is creating standard function implementations automatically. For example, a `struct` with `#[derive(Debug)]` will have a generic implementation for printing debug information.
(3) Attribute — these look like annotations in other programming languages. They are called with the syntax: `#[macro_name]`, and can be attached to anything the Rust Reference considers an "Item" [36].

In this paper, we work primarily with attribute procedural macros. We perform function transformations, and the attribute procedural macros can be used to annotate the relevant functions in a very natural manner. In our implementation, we also make use of a declarative macro, but it is very simple and not of great academic interest.

2.2 GPU Programming

GPUs have a fundamentally different architectural design from Central Processing Units (CPUs) [22]. They are highly parallel systems, with less potent individual cores, but a much higher amount of them, thus being optimized for high throughput, at the expense of having higher latency than a typical CPU.

Each thread in a GPU is part of a group of 32 threads called a warp. Due to the simpler individual cores, the threads in a warp all share the same control flow. This means that if one thread in the warp needs to take a different branch when executing the code, the GPU will typically execute the code twice: once to collect the other 31 threads' results, and once to collect the result of the one divergent thread. This fact must be taken into consideration if one wishes to program GPUs efficiently.

There are two currently widely used general-purpose, low-level GPU programming APIs: CUDA [28] and OpenCL [19]. We currently only target OpenCL, though generating CUDA should not be challenging. Both of these APIs offer programming languages that are very similar to C in their syntax. We will not use any of their more advanced features and will instead focus on the simple syntactic adaptations needed to make Rust code executable on the GPU.

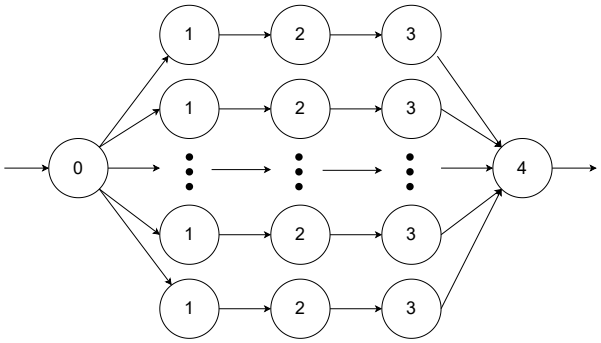


Figure 1: An example of a linear pipeline.

2.3 Linear Pipelines

A linear pipeline is a parallel pattern described by a directed acyclic graph that represents the flow of a computation [25]. Figure 1 has an example of a linear pipeline. Linear pipelines can explore parallelism by operating on different units of work at the same time. In Figure 1, this corresponds to the nodes that have the same number; they are all executing the same code, in parallel. As explained in the previous Subsection, GPUs are particularly good at this. Another way that linear pipelines exploit parallelism is inter-stages: while stage 1 is executing, stage 0 can already process the next bit of data simultaneously. This kind of parallelism is heavily relied on in modern CPUs [35]. This second kind of parallelism is not very relevant for this work, because, as we explain in the next Section, we will not be exploring it, instead leaving it for possible future work.

It is common to call the nodes that begin and end, respectively, *source* and *sink*. These nodes are often special because they must interact with the surrounding environment, often by either reading input or writing output. The rest of the pipeline can, theoretically, be completely hermetic (though in practice, it is not unusual to have the inner nodes write to log files to help diagnose problems).

Our work uses linear pipelines as a case study for GPU code generation. We focus on this pattern because it is both simple to explain and implement, but general and powerful enough to model non-trivial, computationally costly applications. Furthermore, the way it explores parallelism, through stage replication, is very amenable to GPU execution.

3 GPU code generation with Procedural Macros

Our primary goal is to adapt code inside a Rust function so that it may be executed on the GPU. As a case study, we are not concerned with optimizing certain programming patterns, such as map and reduce, but rather simply wish to show the feasibility of our approach. We will be working exclusively with applications that can be represented through a linear pipeline. In our previous work, we formally defined linear pipelines in Rust, expressing parallel computations and targeting multi-core and cluster [Omitted for blind review]. Here, we use very similar ideas, but adapted to work for the GPU. For example, we will not be using threads in our implementation (in our previous work, the *source* and *sink* stages always executed in a separate thread), and all parallelism will be exploited through the GPU. This means the multiple stages will not be executed at the same time, as they should in a proper pipeline. We leave that exploration for future work. Our strategy is to make it so that every inner stage of the pipeline executes in the GPU, while the outer stages (the source and the sink), will the responsible for getting the data in and out of the GPU (and therefore will execute on the CPU). This means we need at least 3 different function transformation routines: one for the source, one for the sink, and one for the inner stages. Listing 1 shows the resulting API. There are three different annotations, one for each different function transformation. Of these, only the `stage` transformation will generate GPU executable code.

```
#[source]
fn source(/*inputs*/) -> /*outputs*/ {
    /* sequential source logic */
}
#[stage]
fn stage(/*inputs*/) -> /*outputs*/ {
    /* sequential stage logic */
}
#[sink]
fn sink(/*inputs*/) -> /*outputs*/ {
    /* sequential sink logic */
}
```

Listing 1: Desired application level code

To transmit data to and from the GPU, we use the `rust-gpu-tools` library[1], which lets us create buffers and compile and execute kernels[2].

As a general-purpose programming language, Rust has many constructs that simply can not be reproduced effectively on the GPU, such as hash maps. As such, we force the programmer to only use fundamental types or vectors of those fundamental types: `i32`, `u8`, `Vec<u32>`, etc. These types all have natural equivalents in OpenCL: `f32` → `float`, `u32` → `unsigned int`, and so on. Vectors are transformed into a pointer of the relevant type, followed by its length. The original functions must only receive these types as parameters (with some exceptions for the source and sink functions), and can only produce these types as output. We validate this requirement in our proc-macro implementation and output an error when it is violated.

The rest of this Section is dedicated to explaining each of the function transformations in Listing 1 in more detail.

3.1 Source and Sink

Because the source and sink execute in the CPU, they have different requirements for the types they may use. The source function can accept any type as a parameter, but it must output a type that implements the `Iterator` trait[3]. The items produced by this iterator must comply with the GPU type limitations we mentioned earlier. Our function transformation consists of copying the original function's implementation, and then mapping every item outputted by the iterator onto its GPU equivalent. Most importantly, this includes putting the content of every Vector in a GPU buffer. The result is that the new, transformed function generates items that are ready to be processed by the GPU in subsequent stages in the pipeline.

[1] https://github.com/filecoin-project/rust-gpu-tools
[2] "kernels", in this context, refer to GPU executable functions
[3] traits are Rust's name for what is usually called "interfaces" in other languages

The sink function, on the other hand, has the same limitations on the types of its parameters as the stage functions, but it can output any arbitrary type. We essentially do the reverse process from the source function: first, we map every GPU type onto its Rust equivalent (this includes transferring over GPU buffers to Rust vectors), then we execute the original function's code on that data.

3.2 Stages

Now that we can get data in and out of the GPU, we must address the stages. The GPU type limitation now extends to *every statement in the function's body*. We can not simply instantiate a binary heap in a GPU kernel, for example. Furthermore, we can not create arbitrary vectors, and so the original Rust function must only use the vectors that it received as input, or a special vector called "output". The original Rust function must be composed of exactly 3 statements:

(1) "`let mut output = vec![0; <size>];`", where `<size>` is an expression that evaluates to an "`usize`". We use this to determine the size of the output vector in the generated function.
(2) the second statement is a `for` loop in the form: "`for global_id in <range>`", where `<range>` evaluates to an "`usize`". The range will be used to calculate how many GPU threads we will have to spawn when creating the kernel. The body of the `for` loop will make up the body of the GPU's kernel.
(3) The final statement is simply the function's output.

The full set of rules and limitations we impose on the programmer for what can be in the original Rust function is listed in Section 3.3. By severely limiting the expressivity of the original Rust code, it is possible to leverage the similarities between Rust and C's syntax and generate the GPU kernel's body through simple syntactic adaptation. The most important adaptation is changing "`let`" statements (`let var: i32 = 0`) to place the correct type before the variable's identifier: (`int var = 0`).

This approach is different from other traditional techniques that try to do very complex static analysis and transformations. Instead, we keep our transformations as simple as possible and limit the ways the original program can be written. This makes it easy to reason about and rely on its correctness, since our syntactic adaptations do not really change the program's semantics. Future work can focus on relaxing some of these constraints while gradually introducing more complex static analysis alongside them. This allows for incremental progress while retaining the transformations' reliability. Section 3.5 shows a full example of what the transformation looks like.

3.3 Limitations

The following is an exhaustive list of every limitation our approach imposes on the original Rust function for us to be able to adapt it to the GPU (some have already been discussed):

(1) every input and output must be either a fundamental type, or a vector of a fundamental type;
(2) type inference is disallowed. The programmer must always use the "`let i: <type>`" syntax. Type checking will be done later by the OpenCL compiler after we generate the code;
(3) functions are disallowed. The programmer must manually inline all functions. The exceptions are functions in the standard library's `f32` and `f64` modules, called specifically through the syntax "`f32::function`" and "`f64::function`" (not using imports), whose names match exactly to functions that exist in OpenCL;
(4) the first statement in the function **must** be in the format "`let mut output = vec![0; <size>];`";
(5) the second statement in the function **must** be in the format "`for global_id in <range>`";
(6) after the "`for`" loop's body, there must be only 1 statement: the variables the function will return;
(7) no other `for` statements are allowed;
(8) `while` and `if` statements' conditions must be surrounded with parenthesis: `while (<condition>)` and `if (<condition>)`. This is not necessary in Rust, but it is in OpenCL and CUDA;
(9) no statement can use any type that is not fundamental. The only vector types allowed are the function's input and the `output` vector; and
(10) the only operations allowed on vectors are reading and writing through indexing and getting the vector's length through "`vector.len()`".

As mentioned, these limitations were all strategically chosen so that the accepted Rust code will be syntactically and semantically similar to OpenCL, thus allowing our code transformations to be very simple. All of these constraints are statically checked in our proc-macro implementation, and errors are reported when they are violated.

3.4 Putting it all together

Since the transformed source function returns an iterator, as long as the type signatures for every stage and the sink are correct, calling the transformed functions consists of calling the source function and then mapping every item with the transformed stage functions, and the sink at the end. Listing 2 shows what that looks like. The `map` function is the one from Rust's standard library that changes an iterator of type `T1` to one of type `T2`. In our case, we do not change the type, as they are all GPU Buffers, but simply execute the code within each stage.

```
source(inputs) // this returns an iterator
    .map(|args| stage1(args))
    .map(|args| stage2(args))
    /* ... */
    .map(|args| sink(args))
```

Listing 2: Calling the transformed functions

3.5 A full example

Listings 3 and 4 show an example of generating code for the GPU. It is the Sobel filter of one of our testing applications. It shows nearly every limitation discussed in Section 3.3. We have commented the code (and formatted the GPU code) to make it easier to follow.

We can observe the many limitations imposed on the Rust code: the format of lines 5 and 9 in Listing 3 is fixed. The lines *must* always be, first, a mutable vector named "output", followed by a `for` loop. The GPU function is composed primarily of what is inside

```rust
fn sobel_filter(img_vec: Vec<u8>, width: u32, height: u32) -> (Vec<u8>, u32, u32) {
    // These first two lines MUST be in this exact general format, as we explained
    // above. We will use this first line to calculate the GPU's output buffer's size
    let mut output = vec![0u8; (width * height) as usize];
    // This line will be used to know how many threads we should spawn. We execute every element
    // in the range in a different thread. In this case, we will be executing 'width' GPU threads.
    for global_id in 0..width as usize {
        let stride: usize = 2 + width as usize;
        // For loops besides the above are not allowed, so we use a while loop instead
        let mut j: usize = 0;
        while (j < height as usize) {
            // Note all the explicit typing with ': i32' and 'as i32'. This is very
            // non-idiomatic Rust, but it is necessary for us to be able to transform it.
            let val0: i32 = img_vec[global_id + (j * stride)] as i32;
            let val1: i32 = img_vec[global_id + 1 + (j * stride)] as i32;
            let val2: i32 = img_vec[global_id + 2 + (j * stride)] as i32;
            let val3: i32 = img_vec[global_id + ((j + 1) * stride)] as i32;
            let val5: i32 = img_vec[global_id + 2 + ((j + 1) * stride)] as i32;
            let val6: i32 = img_vec[global_id + ((j + 2) * stride)] as i32;
            let val7: i32 = img_vec[global_id + 1 + ((j + 2) * stride)] as i32;
            let val8: i32 = img_vec[global_id + 2 + ((j + 2) * stride)] as i32;
            let gx: f64 = ((-val0) + (-2 * val3) + (-val6) + val2 + (2 * val5) + val8) as f64;
            let gy: f64 = ((-val0) + (-2 * val1) + (-val2) + val6 + (2 * val7) + val8) as f64;
            // calling the min and sqrt functions is allowed because
            // they are prefixed by 'f64::'. No other functions can be called.
            let mag: f64 = f64::min(f64::sqrt((gx * gx) + (gy * gy)), 255.0);
            output[global_id + (j * width as usize)] = mag as u8;
            j += 1;
        }
    }
    (output, width, height)
}
```

Listing 3: Sobel filter in Rust (commented)

```rust
// The first four parameters are the same as those of the Rust function. `output` is a new parameter
// that all generated GPU functions have. Note how every array has a length accompaning it.
__kernel void sobel_filter(
    __global unsigned char* img_vec, unsigned int img_vec_len,
    unsigned int width,
    unsigned int height,
    __global unsigned char* output, unsigned int output_len) {
    // We begin by calculating the global ID and seeing if it is in Rust's for loop's range. As
    // explained, we distribute the work by sending each value in the range to a different GPU thread.
    unsigned int global_id = get_global_id(1) * get_global_size(0) + get_global_id(0);
    if (global_id < 0 || global_id >= width) return;
    unsigned long stride = 2 + width;
    // This is the while loop that was in the Rust code.
    unsigned long j = 0;
    while  (j < height) {
        // in here, we use only indexing operations on vectors, which
        // are one of the only 2 operations allowed on them.
        int val0 = img_vec[global_id + (j * stride)];
        int val1 = img_vec[global_id + 1 + (j * stride)];
        int val2 = img_vec[global_id + 2 + (j * stride)];
        int val3 = img_vec[global_id + ((j + 1) * stride)];
        int val5 = img_vec[global_id + 2 + ((j + 1) * stride)];
        int val6 = img_vec[global_id + ((j + 2) * stride)];
        int val7 = img_vec[global_id + 1 + ((j + 2) * stride)];
        int val8 = img_vec[global_id + 2 + ((j + 2) * stride)];
        double gx = ((-val0) + (-2 * val3) + (-val6) + val2 + (2 * val5) + val8);
        double gy = ((-val0) + (-2 * val1) + (-val2) + val6 + (2 * val7) + val8);
        // here, we call the sqrt and min functions. This is allowed because in
        // Rust these functions are prefixed by 'f64::', which we can transform.
        double mag = min(sqrt((gx * gx) + (gy * gy)), 255.0);
        output [global_id +  (j * width)] = mag;
        j += 1;
    }
}
```

Listing 4: Sobel filter GPU generated code (commented and formatted manually)

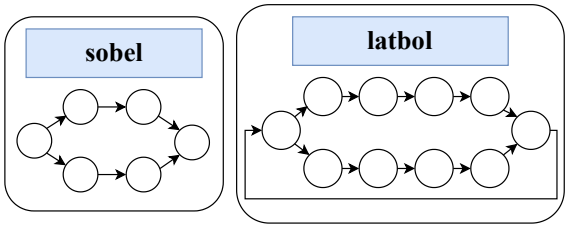


Figure 2: Applications processing graphs

this `for` loop. Each element of the loop will execute in a different thread. Within the loop, we see that we had to use a `while` loop, even though another `for` would make for more idiomatic Rust code. Furthermore, all the types are specified with the syntax `let var: <type>`, which is also not idiomatic Rust. Finally, we use two functions prefixed by `f64`: `f64::min` and `f64::sqrt`. These functions have a direct OpenCL equivalent, and so we can transform them by simply removing their prefixes in the GPU code.

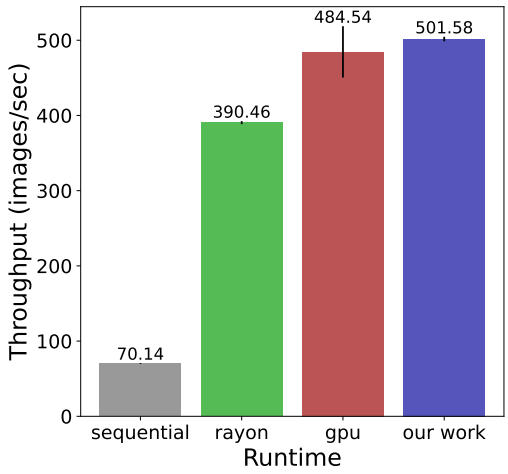
4 Results

This Section will evaluate the effectiveness of our transformations by comparing them to hand-written implementations in terms of performance and programmability. We will use two programs to that end: `sobel`, which transforms a list of images into gray-scale, and then applies a sobel filter [18] on them[4], and `latbol`, which performs a fluid simulation using the Lattice Boltzmann method [6][5]. Figure 2 contains the applications' processing graphs. Note that `latbol` runs in a loop, feeding previous results back into the pipeline. In this case, we managed to model the application naturally by firing the pipeline every iteration, with some caveats, which we discuss below.

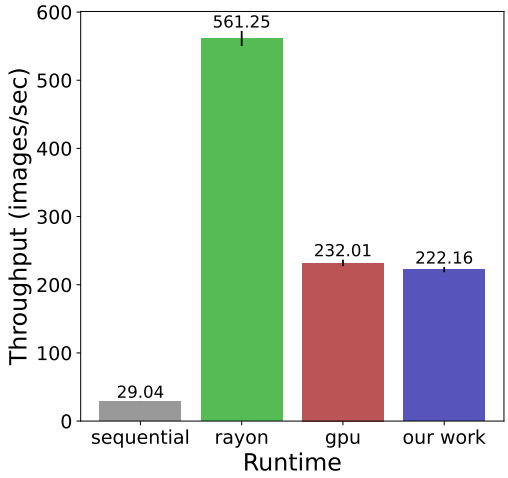
4.1 Performance

We analyze the applications' performance in two different machines: System A with a AMD Ryzen 5 5600X 6-Core, 32GB of RAM and a NVIDIA GeForce RTX 3090 GPU; and System B with two Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz, 190GB of RAM, and an NVIDIA Tesla M40 GPU. Therefore, System A has a rather weak CPU and a powerful GPU, while System B has two powerful CPUs and a comparatively weaker GPU, thus making for two distinct testing environments.

Figures 3 and 4 show the results of executing the programs on both machines. The manual GPU implementation in OpenCL we use as a baseline is called "gpu". We also compare the GPU results with Rayon [31], a widely-used Rust library for multi-threaded parallelism, based on work-stealing. The idea is to see whether using the GPU is better than simply executing the application using the maximum number of CPU threads. And, indeed, Figure 3 shows that, for the `sobel` application, which has a smaller workload, in System B, which contains two powerful CPUs and a relatively weaker GPU, we can actually attain greater throughput by executing the program in the CPUs instead. On the other hand, System A already



(a) System A



(b) System B

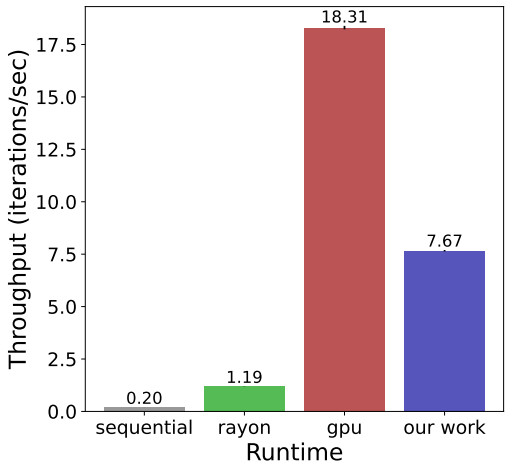
Figure 3: `sobel` GPU benchmark application results

benefits from executing on the GPU, even in an application with a relatively small workload. Furthermore, we see that in both systems our implementation reaches a throughput very close to the manual implementation (for System A, its average is higher, but within the manual implementation's standard deviation). The `latbol` application, which has a heavier workload, is where the advantages of the GPU truly shine. In System A, GPU throughputs are around 20 times higher on average than using Rayon. In System B, the GPU is still around 3 times faster, with our automatic code generation being around 1.75 less efficient than the manual implementation.

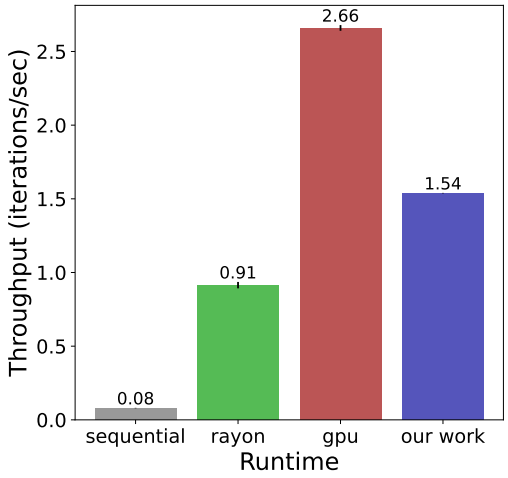
4.2 Programmability

Table 1 summarizes the programmability comparisons between the manual GPU implementation and the automatically generated one. We measure significant lines of code (SLOC) with the `scc` tool[6].

[4] code adapted from https://github.com/dangreco/edgy
[5] code adapted from https://github.com/ndbaker1/bloe
[6] available at: https://github.com/boyter/scc



(a) System A



(b) System B

Figure 4: `latbol` GPU benchmark application results

	sobel			latbol		
	SLOC	Hours	U	SLOC	Hours	U
sequential	40	2.42	N	86	7.59	N
gpu	95	9.06	**Y**	199	75.8	N
our work	58	8.1	N	311	55.44	**Y**

Table 1: GPU programmability metrics. "SLOC" stands for "Significant Lines Of Code", "Hours" for Halstead estimated development effort and "U" for whether unsafe Rust was used in the implementation.

We also use the Halstead [14] method to estimate development time, which is based on operand and operator tokens. It counts the number of operators and operands, their total occurrences, and uses that to calculate a development effort estimate (in estimated hours of work). More specifically, we used Andrade's work in [2], which extended Halstead to be more suitable for measuring parallel programming complexity. Since Andrade's tool was focused on C++, we had to add the relevant keywords for Rust and our use case.

Despite our effort to choose a suitable methodology, it is important to understand that OpenCL involves programming in two languages: Rust and OpenCL (which is a modified C). Furthermore, our proc-macros demand that the functions they will transform be written in a very specific way (see Section 3.3). These factors can make measuring programmability somewhat unreliable because these methods do not take into account the extra cognitive load of having two languages in the same project or of programming while complying with all our imposed limitations.

Keeping these caveats in mind, we see in Table 1 that Halstead estimated development hours give an edge to our code generation approach. Interestingly, this remains true even in `latbol`, where we have around $\frac{1}{3}$ more lines of code. Furthermore, our extra layer of abstraction allowed us to forego the use of unsafe Rust in the `sobel` application, but we were forced to include it in the `latbol` application to attain good performance (we used it to reinterpret a Vector of float arrays as a Vector of floats, so we can send the entire thing at once to the GPU).

5 Related Work

There are still not many academic works regarding Rust and GPUs. Of more recent efforts, we can cite [3], which proposes a new compiler extension to allow writing GPU kernels in native Rust. The authors concluded that while their approach simplified kernel execution for the end programmer, it necessitated many changes to the compiler with regard to type checking and code generation. As Rust is still an evolving language, and the compiler's internal implementation is constantly changing, this seems challenging to accomplish in practice, though perhaps it could be done with an entirely custom compiler. We consider patching the compiler to be too difficult to maintain long-term without a dedicated, specialized team. As such, our approach does not rely on modifying the compiler. In fact, as procedural macros have properly specified behavior, they should work in any correct Rust compiler. Our approach also focuses specifically on ease of implementation, by limiting how the to-be-transformed Rust code can be written.

Another work that attempted something similar is [16], which tried to compile Rust to PTX, NVIDIA's low-level virtual instruction set for GPUs. It is an old work (2013) – before Rust even had a stable release. Indeed, their code examples no longer compile because they use keywords that changed later during Rust's pre-release development. This work is more general than [3], but it has a similar set of problems and limitations, with the additional obvious issue that it was made before Rust's first stable release.

Rust also has promising non-academic works worth mentioning. The Rust CUDA Project [34] is somewhat of a continuation of [16], trying to compile Rust to PTX. Development had stagnated, though it seems to have been picked up again recently. `rust-gpu` [10] is trying to do something similar with SPIR-V [20] as their compilation target. SPIR-V is a binary language primarily used with Vulkan [21] and OpenCL. Vulkan is generally geared towards graphical applications, though it could also be used for general compute.

Finally, C++ has traditionally enjoyed many proposals for abstracting parallel programming in general. As examples, both Kokkos [39] and HPX [17] can execute parallel code on multiple runtimes, including GPUs. These works focus primarily on code portability, defining abstract programming models that allow generating parallel code for multiple backends at once. They then implement an API based on their programming model, which the end developer can use to build their parallel applications. These works do not focus on ease of programming, an explicit concern of ours, and demand that the programmer learn their own abstract model and API, which can often have non-trivial semantics. Our work has no focus on portability (we only execute on GPUs with support for OpenCL), but instead tries to make programming for that one supported platform as easy as possible (although we do have plans to extend this work to also generate CUDA). Furthermore, our proposed abstraction is simpler and easier to implement in general — we do not need a whole programming model to do so — at the price of limiting what kinds of code we can transform.

Also, for C++, there are the works of SPar [13]. SPar wishes primarily to simplify writing parallel stream-processing applications in C++. It uses C++11 annotations to express stream parallelism. These annotations are then parsed by a custom C++ compiler, which generates the parallel code. SPar can generate code to execute in the GPU through [33]. The code the developer writes looks similar to what we have created here, with the main difference that SPar annotates `for` loops, not functions. However, where they differ significantly is in the implementation: the SPar developers had to put a great deal of effort into implementing an entire custom compiler to make their approach work. In contrast, because we limited the kinds of code we allow in our functions, our implementation can be much simpler.

6 Conclusion

In this work, we have proposed a Rust abstraction based on code generation with procedural macros to execute code on the GPU. We have explained our approach, outlining its limitations, and also demonstrated its effectiveness at reducing code complexity while retaining most of the gains of GPU parallelism. The main advantage of our methodology compared with other work is that we do not have to perform any complex static code analysis, adapt the Rust compiler, or define an extra abstract programming model to parallelize the code. By instead simply limiting the possible Rust constructs that can be used within the code to be transformed, our transformation can be very straightforward, and we need not engage with complicated ways of assuring semantic equivalence between the original and generated code.

As future work, we would like to extend the set of benchmarks and testing machines we have used to validate our results. We would also like to implement a CUDA backend, which should not be very challenging, as it would consist of simply changing some of the OpenCL-specific generated code, since CUDA and OpenCL have nearly identical, C-based syntax. A more difficult, but also more interesting direction would be trying to relax the limitations described in 3.3. The more we manage to eliminate, the more ergonomic and natural using our procedural macros would feel. Unfortunately, this would also entail more complex static analysis, defeating one of the main advantages of our approach. We could also use threads to allow multiple stages to offload their calculations to the GPU at the same time, thus further exploring the parallel processing capabilities of the linear pipeline. Finally, we could plan larger-scale studies to evaluate programming productivity more reliably.

ACKNOWLEDGMENTS

This work is funded by PUCRS University, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CNPq Research Program (Nº 306511/2021-5), and FAPERGS 09/2023 PqG (Nº 24/2551-0001400-4).

REFERENCES

[1] AMD. 2025. HIP - Heterogeneous-computing Interface for Portability. https://rocm.docs.amd.com/projects/HIP/en/latest/

[2] Gabriella Andrade, Dalvan Griebler, Rodrigo Santos, Christoph Kessler, August Ernstsson, and Luiz Gustavo Fernandes. 2022. Analyzing Programming Effort Model Accuracy of High-Level Parallel Programs for Stream Processing. In *48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2022) (SEAA'22)*. IEEE, Gran Canaria, Spain, 229–232. https://doi.org/10.1109/SEAA56994.2022.00043

[3] Niek Aukes. 2024. Hybrid compilation between GPGPU and CPU targets for Rust. http://essay.utwente.nl/100981/

[4] Valerio Besozzi. 2024. PPL: Structured Parallel Programming Meets Rust. In *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 78–87. https://doi.org/10.1109/PDP62718.2024.00019

[5] Gianpiero Cabodi, Paolo Camurati, Alessandro Garbo, Michele Giorelli, Stefano Quer, and Francesco Savarese. 2019. A Smart Many-Core Implementation of a Motion Planning Framework along a Reference Path for Autonomous Cars. *Electronics* 8, 2 (2019). https://doi.org/10.3390/electronics8020177

[6] Shiyi Chen and Gary D. Doolen. 1998. LATTICE BOLTZMANN METHOD FOR FLUID FLOWS. *Annual Review of Fluid Mechanics* 30, Volume 30, 1998 (1998), 329–364. https://doi.org/10.1146/annurev.fluid.30.1.329

[7] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2020. An overview of end-to-end entity resolution for big data. *ACM Computing Surveys (CSUR)* 53, 6 (2020), 1–42.

[8] Kees Cook. 2022. Git Pull that introduces Rust to the Linux Kernel. https://lore.kernel.org/lkml/202210010816.1317F2C@keescook/

[9] Andre Rauber Du Bois and Gerson Cavalheiro. 2023. GPotion: An Embedded DSL for GPU Programming in Elixir. In *Proceedings of the XXVII Brazilian Symposium on Programming Languages* (<conf-loc>, <city>Campo Grande, MS</city>, <country>Brazil</country>, </conf-loc>) *(SBLP '23)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3624309.3624314

[10] EmbarkStudios. 2025. rust-gpu. https://github.com/EmbarkStudios/rust-gpu

[11] Leonardo Faé and Dalvan Griebler. 2024. An internal domain-specific language for expressing linear pipelines: a proof-of-concept with MPI in Rust. In *Anais do XXVIII Simpósio Brasileiro de Linguagens de Programação (SBLP'24)*. SBC, Curitiba/PR, 81–90. https://doi.org/10.5753/sblp.2024.3691

[12] Leonardo Faé, Renato Barreto Hoffmann, and Dalvan Griebler. 2023. Source-to-Source Code Transformation on Rust for High-Level Stream Parallelism. In *XXVII Brazilian Symposium on Programming Languages (SBLP) (SBLP'23)*. ACM, Campo Grande, Brazil, 41–49. https://doi.org/10.1145/3624309.3624320

[13] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. 2017. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters* 27, 01 (March 2017), 1740005. https://doi.org/10.1142/S0129626417400059

[14] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., USA.

[15] Bowen He, Xiao Zheng, Yuan Chen, Weinan Li, Yajin Zhou, Xin Long, Pengcheng Zhang, Xiaowei Lu, Linquan Jiang, Qiang Liu, Dennis Cai, and Xiantao Zhang. 2023. DxPU: Large-scale Disaggregated GPU Pools in the Datacenter. *ACM Trans. Archit. Code Optim.* 20, 4, Article 55 (Dec. 2023), 23 pages. https://doi.org/10.1145/3617995

[16] Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine, and Nicholas D. Matsakis. 2013. GPU programming in rust: Implementing high-level abstractions in a systems-level language. In *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013*. 315 – 324. https://doi.org/10.1109/IPDPSW.2013.173 Cited by: 15.

[17] Hartmut Kaiser, Patrick Diehl, Adrian S. Lemoine, Bryce Adelstein Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R. Brandt, Nikunj Gupta, Thomas Heller, Kevin Huck, Zahra Khatami, Alireza Kheirkhahan, Auriane Reverdell, Shahrzad Shirzad, Mikael Simberg, Bibek Wagle, Weile Wei, and Tianyi Zhang. 2020. HPX - The C++ Standard Library for Parallelism and

Concurrency. *Journal of Open Source Software* 5, 53 (2020), 2352. https://doi.org/10.21105/joss.02352

[18] Nick Kanopoulos, Nagesh Vasanthavada, and Robert L Baker. 1988. Design of an image edge detection filter using the Sobel operator. *IEEE Journal of solid-state circuits* 23, 2 (1988), 358–367.

[19] Khronos Group. 2025. The OpenCL Specification. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html

[20] Khronos Group. 2025. SPIR-V Specification. https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html

[21] Khronos Group. 2025. Vulkan 1.3.* - A Specification (with all registered extensions). https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html

[22] David B Kirk and W Hwu Wen-Mei. 2016. *Programming massively parallel processors: a hands-on approach.* Morgan kaufmann.

[23] S. Klabnik and C. Nichols. 2023. *The Rust Programming Language, 2nd Edition.* No Starch Press. https://doc.rust-lang.org/book/second-edition/

[24] Nikolay Kondratyuk, Vsevolod Nikolskiy, Daniil Pavlov, and Vladimir Stegailov. 2021. GPU-accelerated molecular dynamics: State-of-art software performance and porting from Nvidia CUDA to AMD HIP. *The International Journal of High Performance Computing Applications* 35, 4 (2021), 312–324.

[25] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured parallel programming: patterns for efficient computation.* Elsevier.

[26] Tan D Ngo, Tuyen T Bui, Tuan M Pham, Hong TB Thai, Giang L Nguyen, and Tu N Nguyen. 2021. Image deconvolution for optical small satellite with deep learning and real-time GPU acceleration. *Journal of Real-Time Image Processing* 18, 5 (2021), 1697–1710.

[27] NVIDIA. 2024. *CUDA C++ Programming Guide.* NVIDIA.

[28] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2025. CUDA, release: 12.6. https://developer.nvidia.com/cuda-toolkit

[29] Ricardo Pieper, Dalvan Griebler, and Luiz G. Fernandes. 2019. Structured Stream Parallelism for Rust. In *XXIII Brazilian Symposium on Programming Languages (SBLP) (SBLP'19).* ACM, Salvador, Brazil, 54–61. https://doi.org/10.1145/3355378.3355384

[30] Ricardo Pieper, Júnior Löff, Renato Berreto Hoffmann, Dalvan Griebler, and Luiz Gustavo Fernandes. 2021. High-level and Efficient Structured Stream Parallelism for Rust on Multi-cores. *Journal of Computer Languages* 65 (July 2021), 101054. https://doi.org/10.1016/j.cola.2021.101054

[31] Rayon. 2025. Rayon. https://github.com/rayon-rs/rayon

[32] Dinei André Rockenbach. 2020. *High-Level Programming Abstractions for Stream Parallelism on GPUs.* Master's Thesis. School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil.

[33] Dinei A. Rockenbach, Júnior Löff, Gabriell Araujo, Dalvan Griebler, and Luiz G. Fernandes. 2022. High-Level Stream and Data Parallelism in C++ for GPUs. In *XXVI Brazilian Symposium on Programming Languages (SBLP) (SBLP'22).* ACM, Uberlândia, Brazil, 41–49. https://doi.org/10.1145/3561320.3561327

[34] Rust-GPU. 2025. Rust CUDA Project. https://github.com/Rust-GPU/Rust-CUDA

[35] J.P. Shen and M.H. Lipasti. 2005. *Modern Processor Design: Fundamentals of Superscalar Processors.* McGraw-Hill Companies,Incorporated. https://books.google.com.br/books?id=Nibfj2aXwLYC

[36] The Rust Project. 2025. The Rust Reference. https://doc.rust-lang.org/reference/

[37] The Rust Project. 2025. Rustonomicon: The Dark Arts of Advanced and Unsafe Rust Programming. https://doc.rust-lang.org/nomicon/

[38] Tokio. 2025. Tokio - The asynchronous runtime for the Rust programming language. https://tokio.rs

[39] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283