Hybrid compilation between GPGPU and CPU targets for Rust

NIEK AUKES, University of Twente, The Netherlands

Compiling code using both the CPU and GPU has been gaining traction over the last decade. Especially Python has seen many of these innovations added via packages, such as Numba. However, compiled languages have not seen this same growth. This research proposes a new compiler extension to allow writing GPU kernels in native Rust – a language for reliable, high-performance systems programming. Presently, it is impossible to share code between the host (CPU) and device (GPU) within the Rust solutions for hybrid programming. Our approach represents an initial advancement in this area: We made modifications to the Rust compiler to establish the foundation for a hybrid compilation process. These modifications include: (i) enabling the parsing of kernels defined in Rust for subsequent GPU compilation, and (ii) developing a method for accessing the generated kernel bytecode (GPU) from the host Rust code. Additionally, we created a set of libraries to facilitate kernel launching and execution, along with ensuring safe memory management. Our extended compiler and the libraries necessary for executing the GPU kernels in Rust are publicly accessible.

Additional Key Words and Phrases: GPU Acceleration, Compilers, GPU Kernels, Rust, Hybrid Compilation

1 INTRODUCTION

Over the past decade, integrated hybrid programming solutions for GPU acceleration have become more popular in languages like Python. The advent of packages such as Numba, with millions of downloads each month [2], has sparked a revolution in high-performance computing within the Python ecosystem. With support for writing General Purpose GPU code inside Python, using GPU acceleration in Python programs has become more commonplace for high-performance applications like simulations and scientific models. However, this revolution has not yet come to full fruition in compiled languages such as C, C++ or Rust.

Some advancements on GPU acceleration have been made for some compiled languages. However, the current state of the art in most compiled languages is still lagging behind Python. Raw OpenCL API calls and programs are usually the standard for cross-compatible programs [17]. NVIDIA however, does provide a hybrid compiler [6] for C++ to interface with their CUDA-enabled acceleration devices. This hybrid compiler compiles the source code in a 2-step process, where GPU functions are compiled and linked in a preprocessing stage and the rest of the code is compiled afterwards. In Rust, the libraries `rust-cuda` [1] and `rust-gpu` [8] emulate this behaviour, albeit with limitations on the placement of kernels. In these libraries, for example, it is currently not possible to write kernels alongside host code such as in Python or CUDA C++.

TScIT 41, July 8, 2024, Enschede, The Netherlands

`rust-cuda` and `rust-gpu` have made significant advances in hybrid compilation techniques, however, it is still noticeably more complex to write kernels in Rust compared to Python with Numba. We hypothesize the reason for this might be the lack of an integration mechanism that fully binds the host and device code together: in `rust-cuda` and `rust-gpu` the programmer needs to actively separate kernels from the host code. This is an issue Numba and CUDA C++ have successfully solved by allowing the programmer to define kernels alongside the host code.

We propose a new method of hybrid compilation that emulates these successes with hybrid compilation in Rust. This new method builds kernels defined in Rust from within the compiler, which improves the linking between host and device code. In this paper we show how this form of hybrid compilation is achieved, and how we can write a kernel with our prototype as shown in Listing 1.

Intuitively, in Listing 1, we declare the kernel `gpu64` with 1 array argument. This kernel sets each element's value to the index of the element, resulting in the array `[0,1,2,3,...]`. In `main()`, we declare the array we want to pass to the kernel, declare how many threads and blocks we want to use [3], and launch the kernel with the arguments we just defined. The result is then printed to the console. The engine declaration at the top specifies which `launch(..)` functions to import.

```rust
#![engine(cuda::engine)]

#[kernel]
fn gpu64(a: &mut [i32]) {
    let i = tid();
    a[i] = i;
}

fn main() {
    let mut arr: Vec<i32> = vec![1; 256];
    let threads_per_block = 64;
    let blocks_per_grid = 4;

    gpu64.launch(
        threads_per_block,
        blocks_per_grid,
        arr.as_mut_slice());

    println!("Result: {:?}", a);
}
```

Listing 1. Defining and executing a kernel with our prototype

In this work we investigate the following research questions:

(1) Which components of the compiler need to be changed or added in order to support the proposed hybrid compilation method for integrating GPU kernels into CPU code?
(2) To what extent can the execution of kernels be simplified in Rust using our proposed hybrid compilation method?

We provide some background for the Rust language and its compiler in section 3. In section 4, we detail how our hybrid compilation method is implemented in the Rust compiler. Additionally, we describe the challenges that come with the implementation of our method. Furthermore, we also explore how the kernels generated from our hybrid compilation method can be executed from the host code. Section 5 expands on how we execute kernels with the CUDA API. Section 6 describes how one can install our prototype on their computer. Finally, in section 7, we discuss the advantages and disadvantages of our proposed method and compare them to other hybrid compilation techniques in Rust, C++ and Python.

2 RELATED WORK

2.1 Early history of compute kernels

GPUs started out, as the name implies, as graphics accelerators. During the early 2000s, programmable shaders and floating point operations were introduced to GPUs, which greatly advanced their computing capabilities. In 2003, a significant milestone was achieved when 2 research groups independently discovered GPU programs for linear algebra problems that ran faster on the GPU compared to the CPU [5, 13].

In 2006, NVIDIA launched the CUDA (Compute Unified Device Architecture) framework [16]. CUDA allows programmers to write general purpose compute code for highly parallel compute problems, such as matrix multiplication. A few years later, Apple and Khronos group developed OpenCL (Open Computing Language), a language for defining code on (graphical) acceleration devices [17]. Both CUDA and OpenCL use the concept of *kernels*, functions executed on the acceleration device that can be queried from the host code.

2.2 History of integrated GPGPU programming in Python

During the late 2000s, Python's popularity among programmers started to increase, with many people choosing Python for its presumed accessibility and ease of use, particularly for those with limited programming experience. However, as projects expanded, the performance limitations of Python became a problem. This sparked a need for faster, parallelized Python code.

In 2010, Garg and Amaral [9] were the first to propose and describe a hybrid compilation and execution model that uses both the CPU and GPU from a single code source in Python. Their solution aimed to increase the performance of Python by providing an easy-to-use framework that automatically extracts sections of code that can be executed on the GPU. Although their solution successfully sped up Python code, it failed to resonate with Python programmers.

In 2015, Lam et al. [14] launched the Numba project. This library provides tools for accelerating numerical computations, specialising in optimizing code for execution on CPUs and GPUs. In their research paper, Lam et al. describe the same demand for accelerating the performance in Python as Garg and Amaral [2010] did. Their solution, however, has clung on and is massively popular within the Python community. The Numba library garners roughly 17 million downloads per month [2] and Lam et al.'s article on the implementation of Numba has reached 600 citations at time of writing[1]. An important reason for the popularity of this library is its presumed ease of use and syntactic simplicity, making it ideal for experimenting with GPU kernels within existing code bases. Numba makes heavy use of Python's decorator system which allows it to access the original source code and compile entire functions to bytecode.

2.3 Rust as a GPU programming language

While Python has had major developments on GPGPU computing, Rust's developments have been a bit more modest, only consisting of minor advancements and experimental features. In 2013, Holk et al. created a prototype for compiling Rust to a GPU target, specifically targeting the PTX (Parallel Thread eXecution) intermediate language compatible with the NVIDIA CUDA kernels [10].

Up to this point, Rust kernels were mostly written using the OpenCL interface. Kernel code is written in the OpenCL programming language which is then compiled and executed at runtime (see appendix A for an example). However, this method exposes a lot of complexity and requires the programmer to know OpenCL, the intrinsics of the GPU, and how kernels operate behind the scenes[2]. Except for very specific use cases, this complexity is unnecessary and complicates the development process.

Holk et al. details this problem in their paper and proposed a hybrid compilation process to resolve it [10]. Their approach is similar to Garg and Amaral's proposed hybrid compilation model [9].

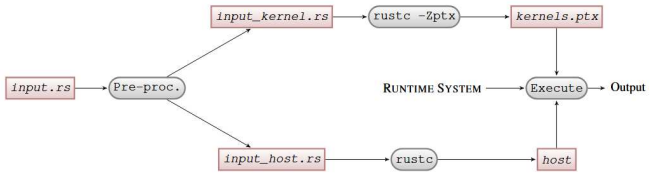

Fig. 1. Overall workflow of Holk's compiler. The pre-processor separates kernels from the host code. The kernels are compiled separately from the host code in order to generate GPU code. [10]

Holk's method aims to preprocess the input file such that host code and device code can be compiled using different compilation targets. The separated code bases are then linked again during runtime. Figure 1 illustrates the workflow of Holk's method, during the preprocessing stage, the input file is separated into 2 source files, one for GPU code and one for CPU code. The GPU code is then

[1] Sourced from https://dl.acm.org/doi/abs/10.1145/2833157.2833162
[2] This includes: building the kernel, supplying appropriate arguments, setting up the output buffer and enqueueing the built kernel for execution

```
#[kernel]
fn add_kernels(++x: ~[float],
               ++y: ~[float],
               ++z: ~[mut float])
{
    let i = ptx_ctaid_x() * ptx_ntid_x() +
             ptx_tid_x();
    z[i] = x[i] + y[i];
}
```

Listing 2. An example of a kernel declaration using Holk's prototype. This kernel function takes three arrays x, y, and z as inputs. It performs element-wise addition of x and y, storing the results in z. Each thread computes one element based on its unique index determined by the block and thread IDs.

compiled to GPU bytecode using a special compiler variant. The CPU code is compiled to a binary which loads the generated GPU bytecode from a file. Holk et al. demonstrated a partial prototype excluding the preprocessor stage.

Since 2020, Embark Studios has maintained the `rust-gpu` library [8]. This library extends to Holk's methods on the compilation of kernels. The `rust-gpu` library compiles (separated) Rust GPU code to SPIR-V [11], a new widely supported low-level language intended for GPU devices. This code can then be embedded into the host code for execution. The `rust-gpu` library has extensive support for programmable shaders, a small program injected into the rendering process to alter what is shown on the screen. This project has already been in development for a few years and is generally regarded as the most stable library of all the available options as of 2024.

Rust syntax, however, is designed as a general-purpose language for CPU computing. Therefore, not all language constructs can be fully implemented for use on the GPU. As Sudwoj described in 2020 [18, p. 10], many basic operations would be unsafe or even unsound due to the differences in the GPU memory model compared to Rust's memory model. Especially pointer aliasing within GPU kernels would be problematic. It should, however, be noted that relatively little research has been done on this topic.

3 BACKGROUND

Rust has a few unique language concepts that makes it stand out among other languages. These concepts include variable ownership, the type system, and fearless concurrency [12].

3.1 Type System and Ownership

Rust has a unique type system taking inspiration from both imperative and functional programming languages. Rust prioritises safety with the design of its type system. This is achieved with the concepts of variable ownership and borrowing.

Ownership is Rust's unique method of managing memory. It allows the compiler to derive when memory should be allocated and deallocated in the program, removing the need for a garbage collector without leaving memory management to the programmer. This system allows Rust to allocate more variables on the stack compared to other languages. Ownership, however, does need a set of rules all variables need to abide for the system to work [12, ch. 4.1]. They are:

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be deallocated

An *owner* of a variable can be anything from functions to other variables or sometimes even the program itself. The owner of a variable may be transferred during function calls by passing the variable as an argument.

Borrowing is Rust's way to pass references to functions and other objects without transferring ownership. In Rust, a reference must always be valid during the duration of its use. This property is checked by the compiler, which refuses to compile the program if a borrowed reference may not be valid when used. This system proves to be a very effective way to manage memory in a reliable and safe manner [12].

3.2 Fearless Concurrency

One of the major goals of Rust is to provide safe and efficient concurrent programming [12, ch. 16]. Rust extends the borrowing system to provide what Rust defines as *fearless concurrency*. Fearless concurrency aims to present concurrency issues as compilation errors, allowing the programmer to fix issues while implementing the code rather than discovering concurrency problems later on. The compiler checks extensively by analysing which variables are shared among threads, whether they are properly guarded from data races, and other analysis techniques.

The Rust team found that ownership system already checks many of these properties. Just like in a single threaded program, borrowing rules prevent normal references being passed to different threads, because the compiler cannot be sure the reference stays valid.

In short, fearless concurrency aims to allows the programmer to write code free of small and subtle bugs and making it easy to refactor code without introducing new bugs.

3.3 Rust Compiler Structures

This section describes the general compiler structure of `rustc`, the Rust compiler. This summary is based on sections of The Rust Compiler Development Guide [7] and focuses on the parts most relevant to this paper.

3.3.1 High-level Compiler Architecture.

In addition to unique language concepts, Rust's compiler also has some unique methods to compile Rust code to binaries. In general, the Rust compiler can be broadly divided into several stages, with an intermediate representation between stages [7, ch. 26]. We summarise these stages in the following list:

(1) **Parsing**: The Rust compiler starts with parsing raw Rust code into an Abstract Syntax Tree (*AST*). The *AST* roughly

represents the Syntax written by the programmer as a tree structure.
(2) **AST Lowering**: In this stage, the *AST* is converted to the High-level Intermediate Representation (*HIR*). During this process, all symbols used by the programmer are resolved and given a *DefId*, which is a unique number that represents every definition in the compiled code.
(3) **HIR Analysis**: After lowering the *AST* to the *HIR*, Type checking and other analysis is performed on the *HIR*. This process creates the typed *HIR*.
(4) **HIR Lowering**: The next stage involves lowering the typed *HIR* to the Mid-level Intermediate Representation (*MIR*). The *MIR* is a simpler, control-flow graph-based representation that captures the semantics of the program. This representation is designed for analysis and optimization of the program's core logic.
(5) **MIR Optimization**: During *MIR* optimization, various analyses and transformations are applied to improve the efficiency and performance of the code. This includes borrow checking[3], as well as other optimizations that simplify and streamline the control flow and data usage in the program.
(6) **Code Generation**: The final stage is the translation of the *MIR* to *LLVM IR* (Low Level Virtual Machine Intermediate Representation) [15]. The LLVM IR is then processed by the LLVM backend, which generates machine code for the specified target.

3.3.2 *Queries.*

Unlike many other compilers, `rustc` does not execute these stages sequentially after each other. Instead, it uses *queries*, which are functions with some extra properties. They are designed to be memoized and context-free, meaning that once a query has been computed, its result is cached and can be reused without recomputation. Queries in `rustc` can be thought of as a series of interconnected questions and answers. Each query corresponds to a specific compiler operation or piece of information, such as type checking a particular function or generating the *MIR* for a module.

In `rustc`, all the major compiler steps described in section 3.3.1 consist of a series of queries calling each other. All queries are defined on the `TyCtxt` struct, which acts as a central compiler context most entities in the compiler have access to. Queries can therefore be used to facilitate information transfer from one part of the compiler to any other part.

Queries also facilitate incremental compilation. Most source code is not changed during compilation sessions and can theoretically be reused if the initial inputs have not changed. `rustc` analyses the inputs of queries for possible changes and decides if results from the previous session can be reused for this query[4].

[3] *Borrow checking* is part of Rust's memory management model. It is explained in chapter 4 of [12]
[4] The algorithm is explained in chapter 28.2 of [7]

4 MODIFICATIONS TO THE RUST COMPILER

For a hybrid compilation process to work, the following key steps are needed to convert and link kernels with the host code:

(1) Identification of kernel code in the source code.
(2) Compilation of kernel code to GPU bytecode.
(3) Integration of the compiled kernel bytecode back into the host code.

In this section, we describe in detail how these steps are implemented in the Rust compiler.

4.1 Overview

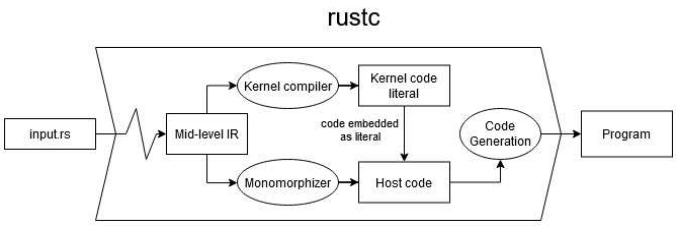

Fig. 2. The overall workflow of our compiler-based method.

Figure 2 describes the overall workflow of our modified compiler. When the compiler needs to compile a file like `input.rs`, it needs to go through all the steps as described in section 3.3.1. In figure 2, the first four steps (parsing, AST lowering, HIR analysis, and HIR lowering) are represented by the single arrow between the input file and the Mid-level IR (MIR). These steps have been simplified in the figure, since they do not differ structurally from the unmodified compiler.

When the compilation process reaches the Mid-level IR, the CPU and GPU code are separated and put into different compilation tracks. The lower track in figure 2 describes the original compilation track that is used for CPU code. The upper track is followed by GPU code, such as user-defined kernels.

The GPU code, following the upper track, is compiled by a newly written code generator, which converts the GPU code MIR into GPU-compatible bytecode. This yields a code string which can be embedded into the host code.

The CPU code, following the lower track, encounters the monomorphiser, which further prepares the MIR for (parallel) code generation. Just before the the prepared MIR is compiled to LLVM-IR, the GPU bytecode is merged back into the CPU code as an embedded literal, which can be accessed during execution. The LLVM-IR code generator then continues compiling the program as normal.

4.2 Language Design

A kernel can be declared by using the `#[kernel]` attribute on a function, as shown in Listing 1. This tells the compiler that the annotated function is a kernel, and should be compiled for the GPU.

The function name is reused to refer to the final bytecode of the GPU kernel. This can be seen in Listing 1 as well, the gpu64 symbol in `main()` refers to the static bytecode that was generated during the compilation process. This behaviour, however, is problematic for the compiler due to what we refer to as the 'Type Masking' problem.

4.3 The Type Masking Problem

In the context of our hybrid compilation method, the type masking problem is a unique challenge that arises from the dual nature of kernel functions. Specifically, the type of a kernel function must be interpreted differently depending on the context in which it is used.

When a kernel is declared, it should be treated as a regular function definition. However, any part of the code referencing the kernel should yield the kernel bytecode rather than the original function definition. This duality in type interpretation creates, what we call, the type masking problem.

The primary challenge of type masking lies in the compiler's need to manage the dual nature of kernels. This requires careful handling within the compiler infrastructure to ensure that each kernel function is interpreted correctly depending on its context. To address this problem, we provide a second type definition for kernels that captures the original function's type, which can be accessed by compiler components that work with the original function definition of the kernel.

4.4 Code Separation and Integration

With type masking issues resolved, we are ready to separate kernel code from host code and integrate kernel bytecode back into host code. To achieve this, we use Rust's query system to our advantage.

Rust's query system is designed to facilitate on-demand computation and caching of compiler artifacts[5]. When the code generator needs the MIR of a particular function, it issues a query for that function. The query system then retrieves and optimizes the MIR when needed by the code generator.

We extend this query-based approach to handle kernel definitions. When the code generator requests the MIR of a kernel function, we intercept this request to perform kernel code generation and literal embedding, yielding the compiled kernel bytecode as a literal. This literal is then embedded within the host code.

Figure 3 describes this process for the example in Listing 1. In this example, the code generator needs to generate LLVM-IR code for the `main()` function and the `gpu64(..)` kernel. The `main()` function is compiled normally following the original queries. However, the `gpu64(..)` kernel is passed to another query that specifically compiles kernels.

[5]Artifacts are anything generated during the compilation process that can be reused in other steps of the process

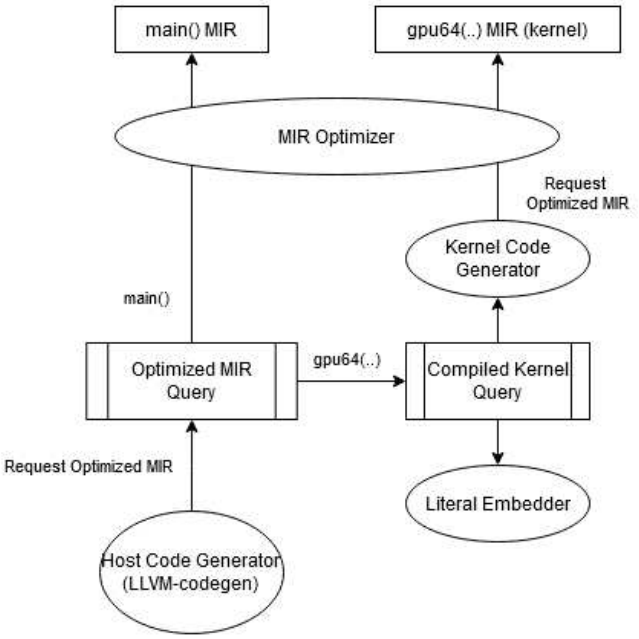

Fig. 3. This diagram highlights the demand-driven aspect of how queries are used, and how we use queries to separate and merge kernel code.

4.5 Code Generation

Before code generation can be performed, the *MIR* needs to undergo a process called *monomorphisation*. Monomorphisation involves resolving generic functions into their specific implementations. In Rust, this process also collects which functions are used within the program to filter out unused functions.

Once monomorphisation is complete, the next step is to generate code from MIR to *NVVM IR* (NVIDIA Virtual Machine Intermediate Representation). There are existing implementations for generating NVVM IR code from MIR, but we encountered several challenges integrating those into our solution. Many existing solutions, for example, rely on foreign APIs, such as LLVM. Integrating these APIs into the Rust compiler proved to be problematic due to compatibility and portability issues.

We attempted to reuse the code generators in [1] and [8]. However, these code generators were built for different versions of `rustc`, requiring us to update these libraries to resolve compile errors. Additionally, we experienced issues with binding the third-party tools used by these code generators into our modified compiler.

Due to these challenges, we were unable to implement the code generation phase within our prototype. Instead, we opted to return a hand-written kernel program, ignoring any code written by the programmer. This approach serves as a placeholder to demonstrate the overall workflow, but it is clearly an area for future improvement.

4.6 Literal Embedding

To access the final kernel code, we need to embed the compiled bytecode into the final executable. This is achieved by replacing the kernel function MIR with a MIR static that then contains the kernel bytecode.

Instead of directly embedding the code as a literal, we wrap it in a structure that carries some additional metadata of the kernel. This metadata provides the name of the kernel and the arguments the kernel expects. External libraries may extend this wrapper with their own implementations for executing kernels. To do this, one must specify the library containing these extensions with the `engine` macro, as shown in listing 1. We provide such a library for CUDA kernel executions, which will be further discussed the following section.

5 KERNEL EXECUTION ON THE GPU

After completing all these steps, we are finally able to access compiled kernels in the host code. In this section, we detail how we use the CUDA API to execute our kernels, and how a programmer can use the libraries presented in this paper.

5.1 The CUDA Execution Model

CUDA [16] is a parallel computing platform and programming model created by NVIDIA. CUDA enables developers to write programs that execute across multiple parallel threads on the GPU. The CUDA execution model is based on the concept of a *grid* of *thread blocks* [3], where each thread block contains a number of threads. thread blocks, in turn, are organised into a grid, which represents all threads used by a kernel.

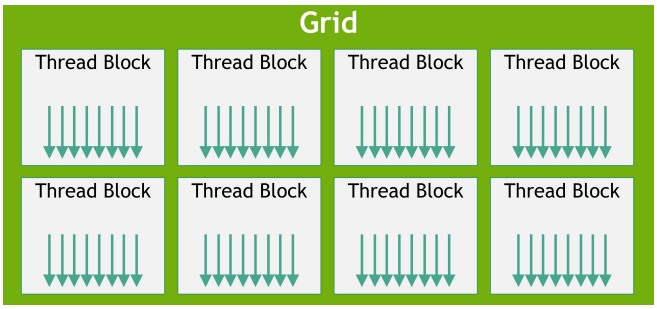

Fig. 4. A grid of thread blocks [3]

CUDA provides 3 different memory types to support the high level of parallelism. These are:

(1) **Global Memory**: Accessible by all threads in the grid, but with high latency. It is used for large data transfers between the host and the device.
(2) **Shared Memory**: Shared among threads within the same block. It has lower latency compared to global memory and is useful for data that needs to be frequently accessed by multiple threads within a block.
(3) **Local Memory**: Private to each thread, but with low latency. It is used for variables that are only accessed within a single thread.

Kernels are launched by the host device. When a kernel is launched, the programmer specifies the grid and block dimensions, which determine the number of thread blocks in the grid and the number of threads in each block. Listing 1, for example, specifies 64 threads per block with 4 blocks in total, resulting in 256 threads in total.

5.1.1 *Interacting with CUDA.*

The CUDA Toolset provides a C API to interact with CUDA-enabled GPUs from the host. This API defines functions for initializing the CUDA environment, managing GPU memory, transferring data between the host and the GPU, and launching kernel executions. With this API, any program is able to launch kernels on the GPU.

5.2 CUDA with Rust

To interact with the CUDA API, we developed a library that provides *safety abstractions* for objects and functions defined in the CUDA API. Usually, foreign APIs do not conform with Rust's rules for safety and memory management. Safety Abstractions act as wrappers around the API such that it does conform with Rust's rules, and make sure that API calls are valid. The libraries we developed for this purpose can be found at https://github.com/NiekAukes/rust-kernels, along with the relevant usage documentation.

This new library allows the library user to load kernels onto the GPU, execute them, and manage device memory for the GPU. Most of these uses are not significantly different from the original API. However, now being able to define kernels in Rust leads us to something interesting we can do with the API. Data types are now completely mirrored between CPU and GPU, this allows us to implement novel ways for handling device memory that work well with Rust's ownership system.

5.2.1 *Device Memory Modelling.*

We model device memory using a special object that represents an object in GPU memory. This object, which we call a device pointer, is created from transferring a regular object, such as an array of integers, to the GPU. Compatible data types contain a method to transfer the object in question to the GPU. This method then yields a device pointer to that object. During the transformation, the original type of the object is retained as metadata within the device pointer.

With the device pointer, data on the GPU can be easily retrieved and copied back to the host memory. This is essentially the same process as transferring to the GPU, except in reverse. Listing 3 highlights how data transfer is done in code.

Unfortunately, not all data structures can be sent to the GPU easily. To send a data structure to the GPU, it needs to be able to capture all information this structure has access to on the host. In other words, it may not have references to host data, as those references are invalid on device memory. We need to be sure all data is accessible on the GPU. This, however, is a limitation of the current implementation, rather than the overall system. Future work

```rust
let mut arr: Vec<i32> = vec![1; 100];
let dptr = arr.to_device().unwrap();
dptr.copy_to_host(&mut arr).unwrap();
```

Listing 3. transferring arr to the device and copying it back. `arr.to_device()` transfers data to the GPU, yielding a device pointer. calling `dptr.copy_to_host(..)` copies the data back to host memory.

may experiment with these concepts and possible solutions for this problem.

5.3 Kernel Launchers

As described in section 4.6, the compiler allows extension methods to be placed on the kernel structure. In our library, we add the `launch(..)` and `launch_with_dptr(..)` methods to these kernel structures. A kernel can be launched using these methods, specifying the grid and block dimensions and supplying necessary arguments to the kernel. The normal `launch(..)` variant takes arguments that still exist on the host memory and transfers them to the GPU. The `launch_with_dptr(..)` variant takes device pointers already created by the programmer.

With these functions, we wrap back to listing 1. This listing uses the `launch(..)` variant to launch the `gpu64` kernel. listing 4 replicates the same program as shown in listing 1 while using the `launch_with_dptr(..)` variant to launch the kernel.

```rust
let device_arr = arr.to_device();
// launch_with_dptr is NOT blocking,
// CPU can do whatever it wants
// while GPU is working
match gpu64.launch_with_dptr(
    threads_per_block,
    blocks_per_grid,
    device_arr.as_mut_slice());
// get the result,
// this is a blocking operation
device_arr.copy_to_host(&mut arr);
println!("Result: {:?}", arr);
```

Listing 4. Executing the kernel with device pointers instead of raw data

6 TOOLING INSTALLATION

To use the prototype shown in this paper, one needs to install the necessary tools and libraries on their computer. This section describes how to install our prototype on a new computer. There are a few prerequisites that need to be available on the host computer. The host computer must:

- Have a CUDA-enabled GPU
- Be running Windows or Linux (Ubuntu)
- Have the CUDA Toolkit installed [3]
- Have `rustup` installed [12, ch. 1]

To install the compiler, clone the compiler from github: https://github.com/NiekAukes/rust/tree/kerneldev-mir-opt and follow the installation instructions in the README.md file.

The libraries necessary for executing kernels can be found at https://github.com/NiekAukes/rust-kernels. This repository also contains a sample project that can be used as a template for projects. To install these, download all files from the repository and unpack them into a folder. The README.md file describes a few extra steps to fully install the libraries. The sample project is already linked with the libraries provided and does not need any further modification. Kernels can now be written and executed with the provided sample project.

7 DISCUSSION

This study set out to design a new compiler-centered hybrid compilation method suitable to Rust and other compiled languages. We succeeded in creating a prototype for this new method and found that it is a valid approach for implementing a hybrid compilation process in Rust.

We found that the compiler needs some adjustments to support a hybrid compilation process as described in this paper. Many parts, such as parsing, name resolution and optimisation passes require minor to no changes. However some parts, like type checking and code generation, need more substantial changes to solve the type masking and literal embedding problems. Other problems, such as kernel code generation, requires building new components from the ground up.

We also found that, with this new method, kernel executions may be significantly simplified for the end programmer. Our approach to literal embedding allows libraries to provide their own data structures and implementations for representing and working with kernels.

The proposed method discussed in this paper is remarkably different than the already existing methods of NVIDIA and `rust-gpu`. These differences result in advantages and disadvantages for our compiler-based method compared to the preprocessor solution of the existing methods.

One advantage of our method is the increased code sharing options for the device code, especially in Rust. Currently, it is impossible to share code between the host and device within the Rust solutions for hybrid programming. This is less of an issue for a language like C++, where the CUDA C++ compiler does support code sharing.

Another noticeable advantage is the integration of device code compilation within the compiler. This could allow for unique code optimisation opportunities, as the compiler knows about both the device code and the host code at the same. Optimisations involving executing and defining kernels could be researched further. Performing static analysis to identify parallelisable loops and replace them with kernels could also be done as further research.

This integration, however, also presents a disadvantage. Due to the device code compilation process being integrated in the compiler, we are not able to separate these components anymore. With the relatively limited use cases of GPU accelerated code, this might present a logistical and motivational challenge with maintaining the compiler for the target language. We would recommend keeping the modified compiler separate from the official download channels while it is maturing, but still easily installable using official sources.

Due to the missing code generation phase, the prototype presented in this paper is not yet suitable for actual use within a project. It is, however, sufficient as a proof of concept on how a compiler could use our method to compile kernels. Previous studies and projects have already established that Rust code can be compiled to GPU targets [1, 8, 10]. The code generation issue is therefore mostly a technical difficulty, rather than a research problem. However, there is still a possibility that difficulties arise with the code generation phase of the proposed method. We therefore recommend further work to be done on integrating code generation in our method.

8 CONCLUSIONS

This study set out to design a new compiler-centered hybrid compilation method suitable to Rust and other compiled languages. We succeeded in creating a proof of concept for this new method and found that it is a valid approach for implementing a hybrid compilation process in Rust, with additional advantages on kernel execution handling. It could be a viable alternative for existing solutions for hybrid programming within the Rust language. However, further research on code generation is needed to fully cement the proposed method's viability.

We are pleased with the result of the prototype this paper presents and the ability to write host and device code alongside each other. There remains much potential in extending this work. Future efforts should focus on improving the code generation phase, enhancing language support for more complex kernel functions, and developing automated memory management solutions to simplify GPU programming. By addressing these areas, we can further optimize the performance, usability, and robustness of the proposed hybrid compilation method in Rust.

REFERENCES

[1] 2021. Rust-CUDA: Ecosystem of libraries and tools for writing and executing fast GPU code fully in Rust. https://github.com/Rust-GPU/Rust-CUDA Accessed: July 10, 2024.

[2] 2024. https://pypistats.org/packages/numba Accessed May 12, 2024.

[3] 2024. https://docs.nvidia.com/cuda/archive/12.4.1/

[4] 2024. https://docs.rs/ocl/0.19.7/ocl/index.html

[5] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.* 22, 3 (jul 2003), 917–924. https://doi.org/10.1145/882262.882364

[6] John Cheng, Max Grossman, and Ty McKercher. 2014. *Professional CUDA c programming*. John Wiley & Sons.

[7] The Rust Project Developers. 2024. *Rust Compiler Development Guide*. https://rustc-dev-guide.rust-lang.org/overview.html Accessed: May 2, 2024.

[8] Embark Studios. 2020-2024. rust-gpu: Rust as a first-class language and ecosystem for GPU graphics & compute shaders. https://github.com/EmbarkStudios/rust-gpu. Accessed: April 2024.

[9] Rahul Garg and José Nelson Amaral. 2010. Compiling Python to a hybrid execution environment. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (Pittsburgh, Pennsylvania, USA) *(GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/1735688.1735695

[10] 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 *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. 315–324. https://doi.org/10.1109/IPDPSW.2013.173

[11] John Kessenich, Boaz Ouriel, and Raun Krisch. 2018. SPIR-V specification. *Khronos Group* 3 (2018), 17.

[12] Steve Klabnik and Carol Nichols. 2023. *The Rust Programming Language, 2nd Edition*. No Starch Press.

[13] Jens Krüger and Rüdiger Westermann. 2003. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.* 22, 3 (jul 2003), 908–916. https://doi.org/10.1145/882262.882363

[14] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

[15] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[16] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2024. CUDA, release: 12.4. https://developer.nvidia.com/cuda-toolkit

[17] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[18] Michal Sudwoj. 2020-09-11. *Rust programming language in the high-performance computing environment*. Bachelor Thesis. ETH Zurich, Zurich. https://doi.org/10.3929/ethz-b-000474922

A RUST WITH OPENCL

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

fn trivial() -> ocl::Result<()> {
    let src = r#"
        __kernel void add(__global float* buffer,
        float scalar) {
            buffer[get_global_id(0)] += scalar;
        }
    "#;
    let pro_que = ProQue::builder()
        .src(src)
        .dims(1 << 20)
        .build()?;
    let buffer = pro_que.create_buffer::<f32>()?;
    let kernel = pro_que
        .kernel_builder("add")
        .arg(&buffer)
        .arg(10.0f32)
        .build()?;
    unsafe { kernel.enq()?; }

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

    println!("200007: {}", vec[200007]);
    Ok(())
}
```

Listing 5. A kernel defined with the OCL library in Rust. OCL allows the programmer to call OpenCL functions from Rust. [4]

B AI STATEMENT

During the preparation of this work the author used Github Copilot and ChatGPT in order to speed up the development process and enhance the writing of this paper. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the content of this work.