GPU Programming in Rust: Implementing High Level Abstractions in a Systems Level Language

Eric Holk Milinda Pathirage Arun Chauhan Andrew Lumsdaine
Email: {eholk,mpathira,achauhan,lums}@cs.indiana.edu
School of Informatics and Computing, Indiana University, Bloomington, IN 47405

Nicholas D. Matsakis
Email: nmatsakis@mozilla.com
Mozilla Research, Mountain View, CA 94043

Abstract—**Graphics processing units (GPUs) have the potential to greatly accelerate many applications, and yet programming models still remain too low level. Many language-based solutions to date have addressed this problem by creating embedded domain-specific languages that compile to CUDA or OpenCL. These targets are meant for human programmers and thus are less than ideal compilation targets. LLVM recently gained a compilation target for PTX, NVIDIA's low-level virtual instruction set for GPUs. This lower-level representation is more expressive than CUDA and OpenCL, making it easier to support advanced language features such as abstract data types or even certain closures. We demonstrate the effectiveness of this approach by extending the Rust programming language with support for GPU kernels. At the most basic level, our extensions provide functionality that is similar to that of CUDA. However, our approach seamlessly integrates with many of Rust's features, making it easy to build a library of ergonomic abstractions for data parallel computing. This approach provides the expressiveness of a high level GPU language like Copperhead or Accelerate, yet also provides the programmer the power needed to create new abstractions when those we have provided are insufficient.**

Index Terms—**GPUs; Mozilla Rust; Harlan; LLVM; PTX**

I. Introduction

The architecture of graphics processing units (GPUs) is very well suited for data-parallel problems. They support extremely high throughput through many parallel processing units and very high memory bandwidth. For problems that match the GPU architecture well, it is not uncommon to easily achieve a 2× speedup over a CPU implementation of the same problem, and tuned implementations can outperform the CPU by a factor of 10 to 100. Programming these processors, however, remains a challenge because the architecture differs so significantly from the CPU. Current GPU programming methods tend to fall into one of two categories: systems level, in which the programmer manages architectural details such as memory placement and the number and arrangement of threads; and high level, which typically provide a set of operators over arrays such as `map` or `reduce` from which kernels are composed.

Both approaches have strengths and weaknesses. High-level approaches are typically more productive for the programmer because they do not have to be as concerned with architectural details. Not all problems are naturally expressed in terms of high-level data parallel operators, however, and these operators may miss out on specific features of the available hardware. Systems-level approaches give the programmer much more control over the execution of their program, and thus the code can be tuned to maximize the features of the hardware. This extra control leads to better performance in the hands of an expert programmer, but at a significant cost to the maintainability of the code.

Ideally, both approaches could exist in the same language. The language would provide access to the low-level architectural features of the GPU hardware, but also provide higher level features to capture common programming patterns. These patterns could be provided in a library that would meet the needs for many programs and yet the programmer could still produce highly tuned application-specific code using the lower level features when necessary.

We explore this very idea in this work using the Rust programming language [1]. Rust provides an ideal mix of systems-level features, while also including efficient implementations of many features from functional languages, such as, higher order functions and type classes. Furthermore, Rust compiles using LLVM [2], which means we can easily generate GPU kernels from Rust using the recently released LLVM PTX target. We have implemented support for writing GPU kernels in Rust, and built several higher level abstractions that make writing GPU kernels feel very similar to writing normal Rust code.

We make the following contributions.

- We show that by translating from LLVM to PTX we can quickly support advanced languages features in GPU kernels, including algebraic data types and some types of closures and higher order functions.
- We show how supporting these advanced features enables us to provide both systems-level abstractions and more convenient high level operators in the same language (Section III).
- We show that this approach still yields performance that is comparable to hand-written OpenCL kernels (Section V).
- We comment on our experience working with LLVM's PTX backend and OpenCL, and discuss how this process can be simplified in the future (Section IV).

II. Background

A. General-Purpose GPU Computing

Graphics Processing Units (GPUs) are specialized processors whose development has primarily been driven for the demand for stunning visuals in video games. As GPUs have become more powerful, they have evolved into fairly general-purpose data-parallel processors and are seeing increased use in scientific computing and other disciplines. Writing efficient GPU programs requires a working knowledge of GPU architecture, and so we provide a brief introduction here. We focus on NVIDIA GPU architectures, although other GPUs are similar [3].

NVIDIA GPUs are made up of several Streaming Multiprocessors (SMs). Each SM supports many execution contexts that are scheduled in terms of *warps*. A warp can be thought of as 32 threads, but these threads all execute in lock step. GPUs emphasize throughput over latency, so there is little to no use of speculation and out of order execution. Instead, these features manifest through the way warps are scheduled. It is very inexpensive for an SM to start executing a different warp, so when the current warp stalls for some reason (for example, waiting on memory), the SM can simply execute a different warp that is ready for execution. This is a form of simultaneous multithreading.

The most popular framework for programming NVIDIA GPUs is CUDA [4], which presents the programmer with the illusion of a virtually unlimited set of threads. These threads are grouped into warps that execute in lock step, and the warps are grouped into blocks. All warps assigned to a single block execute on the same SM. High performance GPU kernels are structured to take advantage of these divisions.

Memory management on GPUs is more complicated than CPUs. Most of the memory falls into the global memory category, which resides in off-chip DRAM. GPUs also provide a small amount of local memory for each SM, which is akin to L2 cache on CPUs. Changes to global memory are visible to all CUDA threads, while local memory changes are only visible to a single thread block. Local memory is very fast, but also limited in size. Writing efficient GPU codes requires judicious use of local and global memory. There are additional memory divisions as well, including constant and texture memory, but we do not consider them in this work.

B. Rust

Rust is a new, multi-paradigm programming language being developed at Mozilla Research [1]. Rust targets large systems applications such as web browsers. The goal of Rust is to offer performance comparable to C++ while ensuring type safety and data-race freedom.

Rust's design intentionally draws inspiration from both functional and imperative languages. Many of the higher-level type system features, such as algebraic data types, pattern matching and bounded polymorphism, were pioneered in languages like Haskell and ML. However, as in C and C++, Rust gives users complete control over memory layout and allocation. Data structures can be embedded within one another without pointer indirection and stack allocation is encouraged. A region-based type system ensures that pointers into the stack never outlive the stack frame.

There are several aspects of Rust's design that work together to make it an attractive host for GPU programming. We focus on three of those aspects here: concurrent programming, owned versus managed memory, and zero-cost abstractions.

a) Concurrent programming: Rust includes native support for concurrent programming in the form of an actor model, similar to Erlang. Programs are structured as independent, lightweight tasks with no shared memory. All communication takes place via message passing over statically typed channels.

Given that a GPU usually works as a coprocessor running independently from the main thread of control in the CPU, it is logical to model the GPU as a task in Rust. A proxy task on the CPU mediates data communication. Computations are initiated by sending messages to this task and the results are sent back over the same channel. Rust programmers are already used to thinking about memory layout and it would likely be a minor step to make Rust aware of the GPU memory hierarchy.

b) Owned vs managed memory: Rust divides heap allocations into two categories: *managed pointers* and *owned pointers*. Managed pointers are similar to the pointers found in other safe languages like Java: they are garbage collected and may be freely aliased. However, unlike Java, managed pointers in Rust are always confined to a single task, in order to avoid the possibility of data races.

Owned pointers, in contrast, cannot be freely aliased but instead are transfered from place to place. The region system is used to guarantee that any aliases to owned data are temporary and cannot escape from a given method call. Since, each owned pointer has exactly one owning reference at any time, it can be safely sent from one task to another without risking data races.

The distinction between owned and managed memory also means that garbage collection is effectively optional. Aliasing of owned pointers is carefully controlled using the region system, allowing the compiler to identify points in the program when owned pointers become dead and freeing the memory immediately.

Rust programs can therefore avoid garbage collection entirely by using owned pointers and the stack frame for their storage needs. In the kinds of programs that Rust is targeting, it is often crucial to be able to guarantee that certain inner loops of the code will have very low latency (for example, the compositor loop in a browser must not stall or the user will experience undesirable glitches and slow performance). The ability to avoid garbage collection is also helpful for the GPU, which is not well-suited to running a garbage collector.

c) Zero-cost abstraction: Like C++, Rust uses aggressive optimization to permit building higher-level abstractions that compile down to lower-level code. As a simple example, the standard idiom for iteration in Rust is to use a higher-order function with a closure argument, as shown here:

```
vec.iter(|e| io::println(e.to_str()));
```

If this code were to be executed naively, it would require constructing a closure and then repeatedly invoking this closure on each element in the vector. Although closure creation in Rust is very cheap—when possible, closures are allocated directly on the stack—this would still be a relatively expensive way to iterate over a loop. In practice, however, aggressive inlining and type specialization means that code like this is compiled into a tight loop comparable to what one would expect from a C `for` loop.

Rust also features an expressive *trait system* that can be used to construct abstractions. Rust's traits are comparable to C++ concepts [5] or Haskell's type classes [6]. Traits permit users to write generic functions that operate over many different types of data. Whenever a generic function is invoked with a specific set of types for its arguments, the compiler will generate a specialized variant (much like C++ templates). Generating specialized variants ensures that the generated machine code is fully statically linked, enabling inlining and other optimizations to be applied.

C. LLVM

LLVM, or Low-Level Virtual Machine, is a "collection of modular and reusable compiler and toolchain technologies." [2], [7]. The goal of LLVM is to provide a modern compilation strategy capable of supporting both static and dynamic compilation of multiple languages. LLVM has gained remarkable popularity in recent years, and is being used in several mainstream compilers, including gcc. Several other related projects have evolved around LLVM [7]. Rust also uses LLVM as its backend.

D. PTX

Parallel Thread Execution, or PTX, is an Instruction-Set Architecture (ISA) and a virtual machine that enables general purpose computing on NVIDIA GPUs [8]. PTX can also be considered as an intermediate language from the perspective of the compiler. The PTX code generated by compiler is translated into the target device instruction set just-in-time, during execution. PTX enables portability across different generations of GPUs and acts as a device-independent ISA that can be used as a code generation target by compilers.

We make use of the PTX back-end of LLVM to generate PTX code directly from our embedded language within Rust.

III. Approach

There are two main parts to our approach. The first is to add support for low-level GPU programming to Rust. The second is to build a set of abstractions on top of these features to simplify the programmer's experience.

```
#[kernel]
fn add_float(x: &float,
             y: &float,
             z: &mut float)
{
  *z = *x + *y
}
```

Fig. 1. A first kernel in Rust.

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

Fig. 2. Vector addition on GPU.

A. Low-level GPU Programming

Rust's already low-level nature makes the first step relatively straightforward. Rust front-end compiles it to LLVM intermediate representation. By adding a mode that uses LLVM's PTX backend we can build basic support for GPU. Fig. 1 shows a simple kernel that adds two floating point numbers and returns the result through the parameter `z`. Rust has built-in support for adding arbitrary annotations. GPU kernels are marked by the `#[kernel]` annotation. In the code, the variables x, y, and z are declared as pointers to `float`. The `mut` keyword specifies that `z` is mutable.

In order to write more powerful kernels, we add a set of intrinsics—functions that expand directly into LLVM instructions—to access low-level GPU-specific values, such as the current thread ID. Using the intrinsic `ptx_tid_x`, we can have each thread operate on a different element of a vector, leading to the vector addition kernel in Fig. 2. Here, the syntax `++x` indicates that pointer `x` is passed by value, but its ownership is not transferred to the callee. The syntax `~[float]` indicates that `x` is a *unique* pointer to an array of `floats`, which precludes aliasing of the array.

We currently do not restrict the language allowed inside kernel functions, although all but a very specific set of forms are likely to be unsupported on current generation of GPU hardware. For example, there is no efficient way to report out of bounds errors from kernels, so the compiler disables array bounds checks when generating PTX code. It is unlikely that we will ever be able to run arbitrary Rust code on the GPU, so the compiler should include a pass to ensure that kernels

```
#[kernel]
fn add_vectors(  N: uint,
               ++A: ~[float],
               ++B: ~[float],
               ++C: ~[mut float])
{
  do gpu::range(0, N) |i| {
      C[i] = A[i] + B[i]
  };
}
```

Fig. 3. Using a range abstraction.

do not include code that cannot be executed on the GPU.

B. Building Abstractions

The second part of our approach is to use the low level features added to the Rust language itself and build easier abstractions. As a trivial example, we can combine the `ptx_ctaid_x() * ptx_ntid_x() + ptx_tid_x()` code from our previous snippet into a function called `thread_id_x`. We can go further, however, and start capturing common patterns in a way that feels more like programming in Rust. One such pattern is seen above, where the kernel receives several vectors as input, determines the current thread ID, and computes one result. We have captured this pattern in the ‘gpu::range‘ function, which mimics Rust’s ‘int::range‘ iterator, as illustrated in Fig. 3.

Another common pattern is reduction, which combines many values into one using a user defined operator. High-performance GPU reduction algorithms typically have three phases to best exploit parallelism as the problem size shrinks closer to its final value. This is often hand-coded, but we simply provide a reduction function in the library that can be specialized for a particular operation. Fig. 4 shows the function signature. Such a reduction abstraction can then be used to add all the elements in a vector as shown in the lower part of the figure. The angular brackets syntax is similar to C++ template syntax and serves an analogous purpose.

Note that `reduce_into` is polymorphic, so it can work on data of any type, provided an appropriate reduction operator is provided.

As another example, Fig. 5 shows an abstraction for five-point stencil, such as the one that could be used in 2D Jacobi iterations. The figure also shows the use of the abstraction. The `shape` argument gives the dimensions of the matrix. Note that the kernel that uses the abstraction does not need to index into the 2D arrays using thread IDs, but instead can use mnemonics for up, down, left, and right neighbors. Clever ordering and formatting of the arguments makes the mnemonics self-explanatory.

```
fn reduce_into<T>(target: &mut T,
                  init: T,
                  source: &[const T],
                  op: fn&(T, T) -> T);
```

```
#[kernel]
fn vector_sum(++src: ~[float],
                dst: &mut float)
{
    gpu::reduce_into(dst, 0f, src,
                     |a, b| a + b);
}
```

Fig. 4. Defining and using the reduction abstraction.

```
fn stencil_into_5pt<T>(
  shape: (uint, uint),
  target: &[mut T],
  source: &[const T],
  op: fn&(T, T, T, T, T) -> T);
```

```
#[kernel]
fn jacobi(++src: ~[float],
               dst: &mut float)
{
    do gpu::stencil_into_5pt((N, N),
                             src,
                             dst)
    |   u,
     l, c, r
        d   | {

     (u + l + r + d) / 4f
    }
}
```

Fig. 5. Defining and using five-point stencil abstraction.

Following the principle of making common cases easy and uncommon cases possible, we envision supplying a library of common patterns. Since our approach enables programmer to write low-level GPU code, they can build their own library of patterns to supplement this library. Note that it is not necessary to first construct an abstraction in order to write a kernel that does not make use of a standard pattern.

IV. Implementation

We have implemented a prototype of our approach within the Mozilla Rust compiler infrastructure. The open source infrastructure is available for download from the Rust website [1].

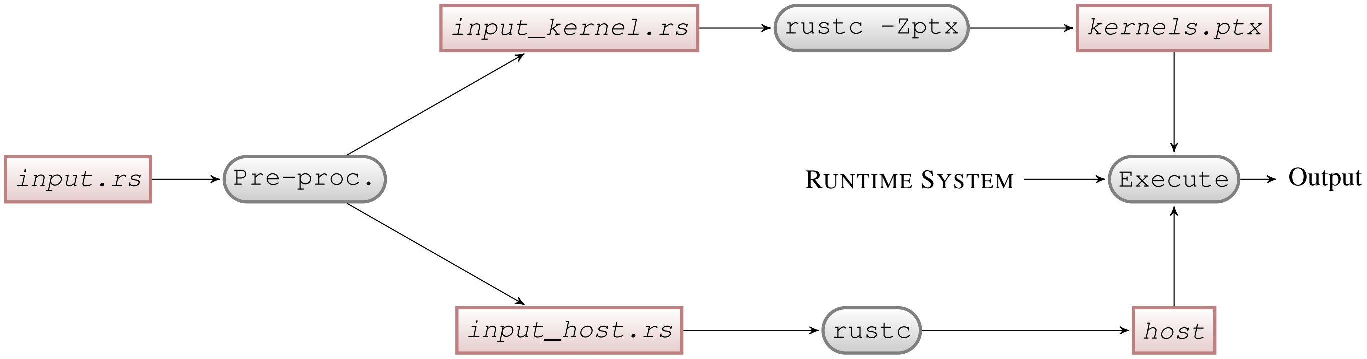

Fig. 6. Overall workflow of our compiler. The pre-processor separates the kernels from the host code. The kernels are compiled with a special flag to generate PTX code and the host code updated with their invocation. Our current implementation is complete except for the preprocessor.

Fig. 6 denotes the overall workflow of compiling a Rust program embellished with GPU code. A preprocessor separates the kernels, to be compiled separately, using the `-Zptx` flag. This produces a `kernels.ptx` file containing the PTX code for the Rust kernels. The host program (`input_host.rs` in the figure) contains code to allocate GPU buffers, transfer data, invoke kernels, etc. This is compiled using the standard Rust compiler. Upon execution, the resulting binary will load `kernels.ptx` and execute that code.

A. Rust Compiler Architecture

Rust currently uses a self-hosted compiler that generates native binaries using the LLVM infrastructure. The compiler consists of a front end and a middle. The front end performs standard tasks like parsing, type checking and other context-sensitive analysis. The middle is mainly responsible for translating a Rust abstract syntax tree (AST) to LLVM assembly.

Rust features a powerful macro system that is based on Schemes syntax-rules. Macro expansion happens in the front end after parsing but before type checking. While this mechanism gives more opportunities for a flexible GPU language embedding, it is hampered in some ways by the fact that macros cannot use type information to control their expansion. It is not possible for a macro to generate different code depending on the type of the expressions it is working with. For this reason, we do not rely on macros other than to provide a nicer syntax on top of features we have implemented in a library.

B. Rust to PTX

The fact that Rust uses LLVM simplifies our task of creating GPU kernels from Rust code. LLVM includes a code generator for PTX, the virtual instruction set for NVIDIA GPUs. Although the PTX code generator is still marketed as experimental, NVIDIA has used an LLVM-based compiler for CUDA since version 4.1. This means the code generator has at least enough support for the code generated by the CUDA compiler.

Compiling Rust to PTX was largely a matter of configuring the Rust compiler to use a different LLVM target, although this required many minor changes. By default, Rust does not enable the PTX code generator, so we had to change the build scripts and LLVM initialization code to include this. The PTX code generator also requires a special `ptx_kernel` calling convention to indicate which functions may be invoked from host code. We modified the compiler to use this calling convention for functions carrying the `#[kernel]` annotation.

One area of particular difficulty was Rust's use of LLVM address spaces. In Rust, different types are assigned to different address spaces in order to track which pointers the garbage collector must be aware of. On the other hand, the PTX code generator uses address spaces to indicate whether pointers point to local memory, global memory, or some other memory region on the GPU. Unfortunately, these two uses of address spaces are incompatible. According to the LLVM Language Reference Manual, the semantics of address spaces are target-specific, which implies the PTX code generators use of address spaces is acceptable while Rust's is not.

We made some modifications to the way Rust assigns pointers to address spaces to be more compatible with the PTX back end's use of address spaces. These modifications have been rather ad hoc thus far, which means many kernels may not compile due to incorrectly using address spaces.

Beside the issues of generating correct code from Rust, we found several LLVM forms that the PTX code generator did not handle. We suspect this is because these are forms that are not generated by the C front-end, Clang, which makes the most extensive use of LLVM, although they are part of LLVM's intermediate language. One example is using literal structs in arguments. We added small patches for these cases to LLVM, and have also contributed the fix back to the mainline LLVM repository.

CUDA kernels have access to several automatically defined variables like `threadIdx` and `blockIdx`. GPU kernel developers using Rust to develop their kernels also need to have access to these variables to write proper GPU kernels. We make them available to Rust kernel developers through compiler intrinsics. These compiler intrinsics (`ptx_tid_x`, `ptx_tid_y`, etc.) provide access to most basic CUDA variables like `threadIdx`, `blockIdx`. During code generation these compiler intrinsics get mapped to respective LLVM intrinsics and finally the PTX backend generates necessary

	Original	*Simplified*
1	clGetPlatformIDs(1, ptr::addr_of(&p_id), ptr::addr_of(&np));	create_compute_context_types([GPU]);
2	clGetDeviceIDs(p_id, CL_DEVICE_TYPE_GPU, 1, ptr::addr_of(&device), ptr::addr_of(&nd));	
3	clCreateContext(ptr::null(), nd, ptr::addr_of(&device), ptr::null(), ptr::null(), ptr::addr_of(&r));	
4	clCreateCommandQueue(ctx, device, 0, ptr::addr_of(&r));	
5	clCreateBuffer(ctx, CL_MEM_READ_ONLY, (sz * sys::size_of::<int>()) as libc::size_t, ptr::null(), ptr::addr_of(&r));	Vector::from_vec(ctx, Ah);
6	clEnqueueWriteBuffer(cque, A, CL_TRUE, 0, (sz * sys::size_of::<int>()) as libc::size_t, vec::raw::to_ptr(vec_a) as *libc::c_void, 0, ptr::null(), ptr::null());	
7	clCreateProgramWithSource(ctx, 1, ptr::addr_of(&(vec::raw::to_ptr(bytes) as *libc::c_char)), ptr::addr_of(&(vec::len(bytes) as libc::size_t)), ptr::addr_of(&r));	let program c̄tx.create_program_from_binary(include_str!("add-vector-kernel.ptx"));
8	clBuildProgram(prog, nd, ptr::addr_of(&device), ptr::null(), ptr::null(), ptr::null());	program.build(ctx.device);
9	clCreateKernel(prog, vec::raw::to_ptr(bytes) as *libc::c_char, ptr::addr_of(&r));	let kernel p̄rogram.create_kernel("add_vector");
10	clSetKernelArg(kernel, 0, sys::size_of::<cl_mem>() as libc::size_t, ptr::addr_of(&A) as *libc::c_void);	execute!(kernel[(size, size), (16,16)], &Ab, &Bb, &Cb, &size);
11	clEnqueueNDRangeKernel(cque, kernel, 1, ptr::null(), ptr::addr_of(&global_item_size), ptr::addr_of(&local_item_size), 0, ptr::null(), ptr::null());	
12	clEnqueueReadBuffer(cque, C, CL_TRUE, 0, (sz * sys::size_of::<int>()) as libc::size_t, buf, 0, ptr::null(), ptr::null());	Cb.to_vec();

TABLE I
We abstracted the Rust OpenCL API.

PTX instructions.

Rust code typically makes extensive use of higher order functions, and this is true in GPU kernels written in Rust as well. Successfully implementing these on the GPU relies heavily on LLVM's optimization strength. Often the closure passed to a function can be entirely eliminated, giving the programmers to ability to write in a similar style to what they are used to while still being able to generate code for the GPU.

C. Rust OpenCL Bindings

In addition to the kernel language, the host CPU code needs a way to execute kernels. We currently do this through Rust bindings to the OpenCL API. OpenCL provides APIs to load binary kernels, which happen to be PTX files for NVIDIA cards. We also use OpenCL for allocating memory on the GPU and transferring data between the CPU and GPU. We have worked to make this fairly ergonomic, but in the future we would like to move more of this into the Rust runtime to keep these details hidden from the user in common cases.

We started with an existing set of OpenCL bindings for Rust, but it was a direct mapping of C API to Rust and users of that API needed to use C level pointer manipulation libraries in Rust to get something done. To make it easy to program and make it easy to integrated with any Rust code/library we came up with a new wrapper layer for the Rust OpenCL binding. Table I compares the simplified API with the original OpenCL API supported by the Rust compiler.

Below are brief comparisons about each API method we listed above.

- Low-level OpenCL API call clGetPlatformIDs should be called twice. The first call determines the number of platforms and the second enumerates these platforms. The new API method get_plaforms greatly simplify this by handling those low-level details internally.
- Similar to what we discussed for clGetPlatformIds, clGetDeviceIds should be called twice to get the list of devices available in the system. New API method get_devices simplify this by doing that internally.
- The create_context, create_commandqueue and create_buffer API methods handle all the low level details like null pointers and Rust to C pointer conversions internally and provide a nice rustic API to the GPU programmer.
- All the other new API methods listed above also hide some of the low-level details like type casting and C pointer conversions internally and provide clear and concise API to the programmer. But there is room for improvements in this improved version of the OpenCL wrapper too.

D. Challenges

There are certain language features that our current implementation does not yet handle fully. One particular case is passing vector slices into kernels. Vector slices in Rust are a way to provide views into various types of vectors, regardless of which heap the vector resides in. They are represented as pairs of pointer and length. Unfortunately, OpenCL provides very few guarantees about addresses of buffers on the GPU, especially between kernel invocations. This makes pointers to pointers and structures with practically impossible to implement. For vector slices, we are unable to construct the pointer and length tuple on the CPU to pass to the kernel because we cannot predict the address of the buffer on the GPU ahead of time. CUDA does not have these same problems, so for these scenarios it would be useful to rely on CUDA API or the NVIDIA driver API to gain more expressiveness. While OpenCL's portability is attractive, the use of PTX ties us to the NVIDIA hardware, in which case using NVIDIA driver API has no disadvantage. Another way to get around this restriction in OpenCL is to pass the pointer and length as separate kernel parameters and use these to reconstruct a slice in the kernel.

V. Evaluation

We wrote several benchmarks both in Rust and OpenCL in order to show that the benefits of programming kernels in Rust do not come at a prohibitive performance penalty. We tested vector addition, matrix multiplication, Cholesky factorization and Jacobi iteration. For each benchmark, we wrote a version purely in Rust, a version that uses the Rust OpenCL bindings to call a kernel written in OpenCL C, and a version that uses a C program as the host for calling OpenCL kernels. All of our experiments were conducted on a Linux PC running Gentoo Linux with 8GB PC17000 RAM, Intel Core i7 2600k 3.4GHz processor and NVidia GTX 460 with 1GB RAM GPU.

Fig. 7 shows the execution time for each of these kernels, excluding the time to transfer data between the CPU and GPU memories. In general, the execution times are close between all three implementations. This demonstrates that our strategy is able to generate efficient kernels and efficient interface to Rust. This is an important consideration since performance is a major motivation behind leveraging GPUs, and Rust aims at providing abstractions without the abstraction penalties.

The vector addition case shows variability for small vectors. Since the vertical axis is logarithmic, these times are much smaller than the rest of the times in the graph (of the order of a few microseconds). As a result, the confidence intervals are relatively large. However, for larger sizes, and all other benchmarks, the times are consistent and confidence intervals are tight.

In the case of Jacobi iteration, the Rust kernel slightly outperforms the OpenCL kernel. This seems surprising at first. However, we reap the benefits of using a backend that is capable of aggressive optimizations and generating high performance code. For example, in our observations of the generated PTX code in both cases, we found that the LLVM backend was far more likely to generate fused-multiply-add (FMA) instructions than the standard OpenCL compiler.

Often data transfer costs can dominate when computations are off-loaded onto GPUs. This is especially true if the transfer costs cannot be hidden. In order to estimate the relative overheads imposed by Rust, we also measured data transfer times for each run of each benchmark. Fig. 8 shows the time to transfer data to and from the GPU memory. In general, the pure OpenCL version is the fastest. When the kernel is invoked from within Rust, there are certain overheads due the extra layers of API. An additional cost an artifact of our current Rust OpenCL library design, that ends up unnecessarily copying output buffers to the GPU before running the kernel. However, we emphasize that the main goal of this paper has been to design an embedded language to express GPU computations without computational penalties, in which we have succeeded as Fig. 7 has demonstrated.

These results show that our strategy is capable of generating high performance GPU code. Combined with Rust's zero-cost abstraction philosophy this provides a compelling environment for programming GPUs.

VI. Related Work

There are many current research and commercial projects to add support for implementing GPU kernels in high-level languages while CUDA [9] and OpenCL [10] are becoming more and more popular among the heterogeneous parallel programming community. One popular way of providing GPU kernel support in existing languages is directive based frameworks like OpenACC [11]. Other methods vary from totally new languages to language extensions with some form of modification to its compiler. Lime [12] and Chestnut [13] are examples for new programming languages which provide GPU support and C++ AMP [14] is set of C++ extensions and template libraries which enables data-parallel programming. Work by Hormati et al. [15] and work by Cunningham et al. [16] are some examples of new compiler frameworks or modifications to existing compilers to incorporate first class GPU programming support in high-level languages.

Our work stands out from these works because we translate Rust kernels directly from Rust's native LLVM representation into PTX code. All of the above-mentioned work either translate kernels into CUDA, OpenCL or some other intermediate language or API.

Even though we see lot of activity around supporting GPU kernel development in high-level languages in present day, there were projects, mainly Sh [17] and Accelerator [18] which tries to allow programmers to write GPU programs using high-level constructs in C++ opposed to pixel shading languages. Sh is a C++ library which support implementation of shader programs as well as general purpose stream computations utilizing most of the capabilites in C++ including classes, templates, and user defined types. Sh compiles to both GPU and CPU in order to fully utilize the all the available system resources by proper load balancing. The main difference between Sh and Accelerator is Accelerator was built around a data type

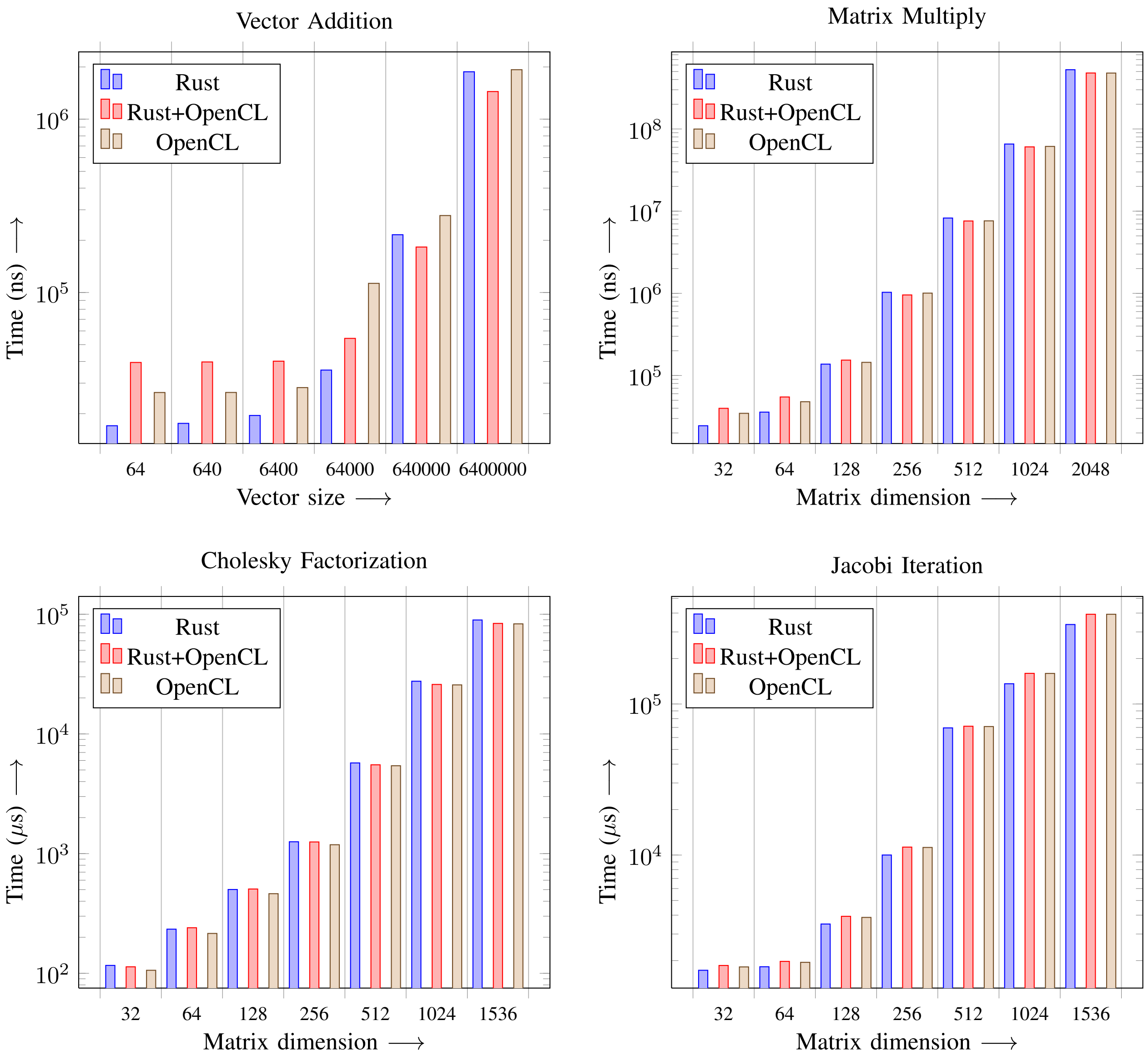

Fig. 7. Kernel execution times, excluding data transfer times.

abstraction called data-parallel arrays and the framework takes care of automatically generating kernels in a pixel shader language. Accelerator completely hides the underlying GPU hardware from the user while allowing the user to focus more on the algorithm.

C++ AMP has lot of similarities to the Accelerator framework. The C++ AMP programming model is also built around multidimensional arrays and hides the explicit data transfer required in GPU languages like OpenCL and CUDA. In addition, C++ AMP has built-in support for indexing and tiling which hides the underlying hardware specific indexing and tiling. As opposed to this model, our work tries to keep all the capabilities of CUDA available to programmer while providing integration between the high-level constructs in Rust and the GPU programming model. For example, the user can access low-level CUDA features like block synchronization or accessing thread and block index variables.

Jacket [19] is one of the closest related work to ours which provides support for developing data-parallel programs in MATLAB, and has the capability to translate MATLAB and C++ code to CUDA PTX. This framework is built around a matrix class called *garray* and its typed subclasses and provides constructs like *gfor* which is a parallel for-loop implementation.

Work by Cunningham et al. [16] shows how X10 programmers can run programms written in APGAS programing model on GPUs. Scheduling of kernels in GPU is based on APGAS programming constructs such as *places*, *asynchrony* and *order*. Allocating memory on GPU should be done in host code using special API calls and copying memory back also requires special API calls.

Sponge [15] is a compiler framework for GPUs that gener-

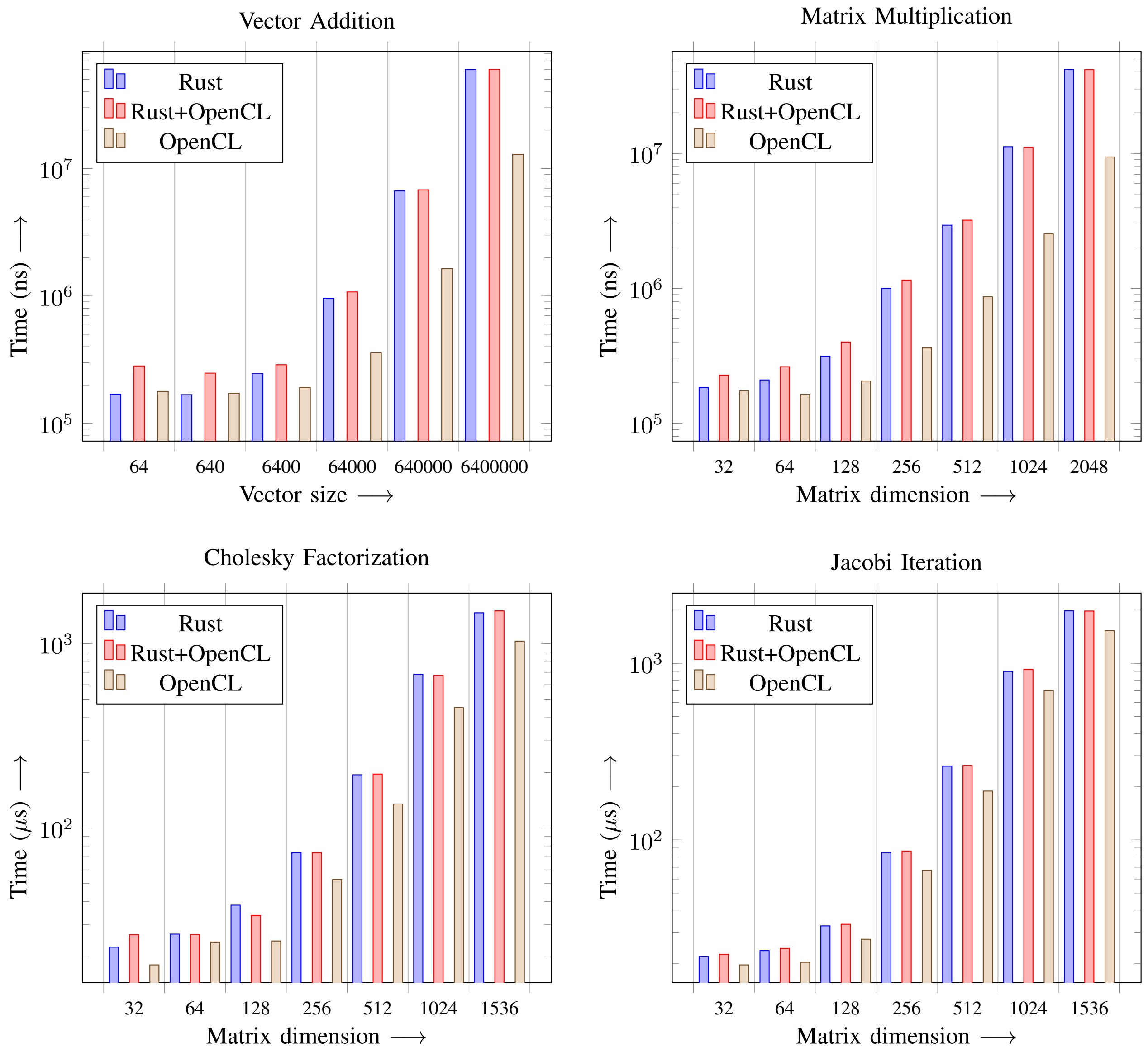

Fig. 8. Data transfer times between CPU and GPU (including both ways).

ates optimized CUDA code which is portable across different GPU generations. Sponge was designed for the sysnchronous data flow streaming languge StreamIt [20]. Accelerate [21] and Copperhead [22] are two recent projects which try to bring GPU programming to Haskell and Python respectively by providing embedded languages which provide rich constructs for GPU kernel development. Both these embedded languages use source-to-source translation where they translate high-level language constructs to CUDA and has specialied runtime libraries based on CUDA which provide various runtime optimizations which matches host language architecture.

One of the main aspects of our work is to come up with rich set of abstractions(patterns) based on Rust features like macros to makes it easy to write data parallel computations for GPUS. We found that Thrust [23] which is a parallel template library based on C++ Standard Template Library(STL) is very similar to our approach from the aspect of providing standard set of patterns/templates which makes it easy to implement GPU computations in C++ while fully utlizing CUDA features.

VII. Conclusion and Future Work

We have demonstrated that it is possible to use the NVPTX backend included with LLVM to generate GPU kernels directly from Rust. These kernels can be loaded using the existing, although now improved, Rust OpenCL bindings. Because already compiles through LLVM, we very quickly gained support for some of Rust's more advanced features. These allow us to provide low level GPU programming features and rely on the expressiveness of Rust to build high level abstractions. Our performance results show that Rust's design preference towards zero cost abstractions benefits GPU code as well; we achieve performance comparable to hand-written

OpenCL kernels while gaining the benefits of programming in Rust.

We believe that compiling to PTX via LLVM is an approach that would benefit other research in GPU programming languages as well. A lower-level target than CUDA or OpenCL gives the compiler more control over data representation and program execution. The fact that many existing languages target LLVM already means that many projects will be able to reuse much of the existing compiler infrastructure, as we have done here.

There are several avenues for extending this work. We currently do not expose the GPU memory hierarchy to Rust, which means kernels cannot yet take advantage of shared memory. Supporting this properly would likely involve extensions to Rust's type system, but we believe these changes should be compatible with the existing design and philosophy of Rust. Extensions to Rust's region and type system may prove to be an elegant way to explicitly and safely manage to distinction between various levels of the CPU and GPU memory hierarchy.

We are pleased with the ability to use Rust idioms in GPU code, and the potential for building high level GPU abstractions from these. We have demonstrated some already, but there remains much potential to extend this work. Mixing low level code with high level abstractions when necessary promises to improve the productivity of GPU programmers without paying the performance penalty that often comes with high productivity programming languages.

The code used for this paper is available at https://github.com/eholk/RustGPU.

VIII. Acknowledgement

We would like to thank the members of the Rust team at Mozilla Research for their fantastic work on Rust and their helpful discussions. We would also like to give a shout out to the people on the #rust IRC channel for their interest in our work. We additionally would like to thank Luqman Aden for initially creating the rust-opencl bindings and Justin Holewinski for guiding us through the process of contributing a patch to LLVM.

References

[1] "The Rust programming language," http://www.rust-lang.org/.

[2] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[4] *CUDA C Programming Guide*, NVIDIA, Oct. 2012, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[5] G. Dos Reis and B. Stroustrup, "Specifying C++ concepts," in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 295–308. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111064

[6] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '89. New York, NY, USA: ACM, 1989, pp. 60–76. [Online]. Available: http://doi.acm.org/10.1145/75277.75283

[7] "The LLVM compiler infrastructure," http://llvm.org.

[8] *Parallel Thread Execution ISA Version 3.1*, NVIDIA, Sep. 2012, http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.pdf.

[9] N. Corporation, "CUDA C programming guide," http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[10] K. Group, "The OpenCL specification," http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[11] OpenACC, "The openACC application programming interface, version 1.0," http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf.

[12] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a high-level language for GPUs: (via language support for architectures and compilers)," *SIGPLAN Not.*, vol. 47, no. 6, pp. 1–12, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2345156.2254066

[13] A. Stromme, R. Carlson, and T. Newhall, "Chestnut: A GPU programming language for non-experts," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '12. New York, NY, USA: ACM, 2012, pp. 156–167. [Online]. Available: http://doi.acm.org/10.1145/2141702.2141720

[14] Microsoft, "C++ AMP," http://msdn.microsoft.com/en-us/library/vstudio/hh265137.aspx.

[15] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: Portable stream programming on graphics engines," *SIGPLAN Not.*, vol. 46, no. 3, pp. 381–392, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1961296.1950409

[16] D. Cunningham, R. Bordawekar, and V. Saraswat, "GPU programming in a high level language: Compiling X10 to CUDA," in *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, ser. X10 '11. New York, NY, USA: ACM, 2011, pp. 8:1–8:10. [Online]. Available: http://doi.acm.org/10.1145/2212736.2212744

[17] M. McCool, "Exploring Sh: A new GPU metaprogramming toolkit," http://www.gamasutra.com/view/feature/2120/exploring_sh_a_new_gpu_.php.

[18] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program gpus for general-purpose uses," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 325–335, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1168919.1168898

[19] G. Pryor, B. Lucey, S. Maddipatla, C. McClanahan, J. Melonakos, V. Venugopalakrishnan, K. Patel, P. Yalamanchili, and J. Malcolm, "High-level GPU computing with Jacket for MATLAB and C/C++," in *Proceedings of Modeling and Simulation for Defense Systems and Applications (SPIE) VI*, vol. 8060, 2011, pp. 806 005–806 005–6. [Online]. Available: http://dx.doi.org/10.1117/12.884899

[20] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Compiler Construction*. Springer, 2002, pp. 49–84.

[21] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating haskell array codes with multicore gpus," in *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, ser. DAMP '11. New York, NY, USA: ACM, 2011, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/1926354.1926358

[22] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 47–56. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941562

[23] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," *GPU Computing Gems*, pp. 359–371, 2011.