Compiling Rust for GPUs

Eric Holk and Milinda Pathirage

1. Introduction

In the last decade, *graphics processing units* (GPUs) evolved from devices purely for graphics acceleration to general purpose parallel processing units. GPUs available in market provide hundreds of computing cores while maintaining high energy efficiency for a little cost. Current GPU programming environments such as CUDA and OpenCL are based on the C programming language and programming is difficult and restrictive. These frameworks restrict the use of the modern language features and developers need to manually manage the data movement between host and the GPU. These restrictions and the necessity to manage architectural details distract the programmer from the important aspects of their algorithm. To alleviate these pain points, higher level programming languages are needed.

Rust is a new programming language that is under development at Mozilla. The goal of Rust is to provide a safe systems programming language that still gives programmers roughly the same control over the program's performance as does a language like C++. Rust borrows many ideas from languages like ML and Haskell, such as algebraic data types, pattern matching and bounded polymorphism. In addition, Rust has a concurrency-aware memory model that enforces ownership transfers between lightweight tasks that conceptually do not share memory. The Rust memory model, in particular, means the language is also well suited for GPU programming.

2. Design

The overall design of our approach is similar to that of CUDA. Kernels are written in Rust, but given a `#[kernel]` annotation to indicate that they are meant to execute on the GPU. The rest of the program uses OpenCL to manage data movement between the CPU and GPU, loading kernels, and executing kernels. These functions are similar to CUDA's `cudaMalloc`, `cudaMemcpy`, and kernel invocation syntax. Below is an example kernel written in Rust.

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

```
}
```

In an ideal scenario, CPU and GPU code could exist in the same file and the compiler would transparently divide them into CPU- and GPU-specific portions. As we will see in Section 3.4, however, our process is not yet this streamlined.

We currently do not restrict the language allowed inside of kernel functions, although all but a very specific set of forms will likely fail in code generation. It is unlikely that we will be able to run arbitrary Rust code on the GPU, so the compiler should include a lint pass to ensure that kernels do not include code that cannot be executed on the GPU.

3. Implementation

3.1 Rust Compiler Architecture

Rust currently uses a self-hosted compiler that generates native binaries using the LLVM infrastructure. The compiler can be thought of in two parts: a front end and a middle end. The front end does traditional front end tasks like parsing, type checking and other analysis passes that are necessary to ensure the safety of Rust programs. The middle end is mainly responsible for translating a Rust abstract syntax tree (AST) to LLVM assembly.

Rust features a powerful macro system that is based on Scheme's `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.

3.2 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[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 telling the Rust compiler to use a different LLVM target, although this required many minor changes. By default, Rust

[1] http://blog.llvm.org/2011/12/nvidia-cuda-41-compiler-now-built-on.html

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[2], which implies the PTX code generator's 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 very ad hoc thus far, which means many kernels may not compile due to incorrectly using address spaces.

Besides 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 Clang although they are part of the 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[3].

CUDA kernels have access to several automatically defined variables like **`threadIdx`** and **`blockIdx`**. GPU kernel developers which use 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 PTX instructions.

3.3 Rust OpenCL Bindings

There was an OpenCL library for Rust based on the OpenCL C API, 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 Rust OpenCL binding.

---

[2] http://llvm.org/docs/LangRef.html#id23

[3] http://llvm.org/viewvc/llvm-project?view=rev&revision=169418

The table below compares old rust-opencl API with the improved version. The main aim of the improved API is to blend in with Rust type system and Rust programming model.

	rust-opencl	Improved rust-opencl
1	**clGetPlatformIDs**(1, ptr::addr_of(&p_id), ptr::addr_of(&np));	**get_platforms**()
2	**clGetDeviceIDs**(p_id, CL_DEVICE_TYPE_GPU, 1, ptr::addr_of(&device), ptr::addr_of(&nd));	**get_devices**(platforms[0]);
3	**clCreateContext**(ptr::null(), nd, ptr::addr_of(&device), ptr::null(), ptr::null(), ptr::addr_of(&r));	**create_context**(devices[0]);
4	**clCreateCommandQueue**(ctx, device, 0, ptr::addr_of(&r));	**create_commandqueue**(&context, devices[0]);
5	**clCreateBuffer**(ctx, CL_MEM_READ_ONLY, (sz * sys::size_of::<int>()) as libc::size_t, ptr::null(), ptr::addr_of(&r));	**create_buffer**(&context, (sz * sys::size_of::<float>()) as int, CL_MEM_READ_ONLY);
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());	**enqueue_write_buffer**(&cqueue, &buf_a, &vec_a);
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));	**create_program_with_binary**(&context, devices[0], &path::Path(@"rust-ptx.bin"));

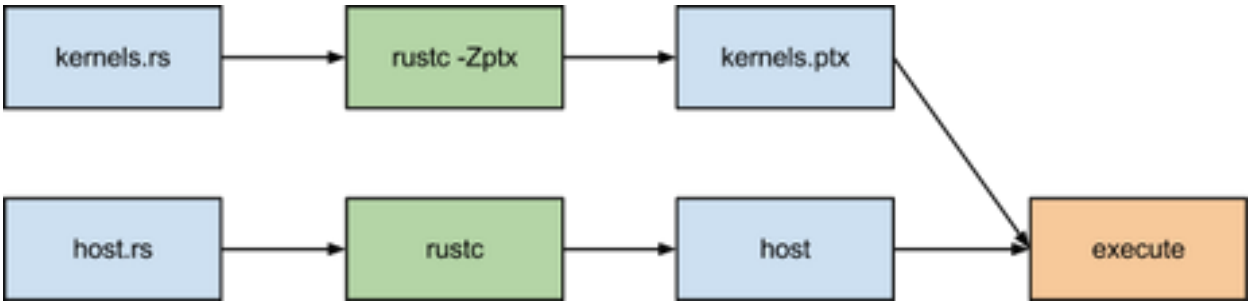
8	**clBuildProgram**(prog, nd, ptr::addr_of(&device), ptr::null(), ptr::null(), ptr::null());	**build_program**(&program, devices[0]);
9	**clCreateKernel**(prog, vec::raw::to_ptr(bytes) as *libc::c_char, ptr::addr_of(&r));	**create_kernel**(&program, @"_ZN9add_float17_d08d41c0c85935643_00E");
10	**clSetKernelArg**(kernel, 0, sys::size_of::<cl_mem>() as libc::size_t, ptr::addr_of(&A) as *libc::c_void);	**set_kernel_arg**(&kernel, 2, &buf_a);
11	**clEnqueueNDRangeKernel**(cque, kernel, 1, ptr::null(), ptr::addr_of(&global_item_size), ptr::addr_of(&local_item_size), 0, ptr::null(), ptr::null());	**enqueue_nd_range_kernel**(&cque ue, &kernel, 1, 0, sz as int, 2);
12	**clEnqueueReadBuffer**(cque, C, CL_TRUE, 0, (sz * sys::size_of::<int>()) as libc::size_t, buf, 0, ptr::null(), ptr::null());	**enqueue_read_buffer**(&cqueue, &buf_c, &vec_c);

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.

3.4 Workflow

The figure below shows an overview of the steps needed to run Rust code on the GPU.



Currently the GPU kernel code must reside in a separate file from the host code. The kernels are compiled separately, using the `-Zptx` flag. This produces a kernels.ptx file which contains the PTX code for the Rust kernels. The host program (host.rs in the figure above) contains code to allocate GPU buffers, transfer data, invoke kernels, etc. This is compiled using the normal Rust compiler. Upon execution, the resulting binary will load kernels.ptx and execute the kernel code contained within.

4. 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, OpenCL bindings. Kernels written in Rust can already support most operations involving plain-old-data (POD) types, such as scalars, tagged unions (`enum` types) and records.

Our current implementation is not without its limitations, however. We have not been able to make kernels that operate on more than one element of a vector. This is in part because Rust generates incompatible address spaces for the pointers within a vector type. A further complication is that Rust performs bounds checking on array accesses, so we would need a way to report failures from GPU kernels to the host code. Finally, it would be useful to enable support for nested pointer structures, such as trees and vectors of vectors. We are exploring solutions to some of these problems in the Harlan project.

As part of this project, we greatly improved the Rust OpenCL bindings, which were previously just direct calls into the OpenCL C API. While things are greatly improved, we still do not have a truly “rustic” API. For example, our higher level wrappers do not work as nicely with the type system as they could, meaning that the user must explicitly calculate data sizes when allocating buffers. Some of these improvements will likely come organically as the Rust OpenCL bindings see more use.

Finally, our solution is currently tied to NVIDIA GPUs by way of PTX. Most other OpenCL implementations; including those of AMD, Intel and Apple, appear to be built using LLVM technology. In theory, this means it would not be too difficult to include support

for other GPUs, although the other vendors do not seem to be publishing their specific LLVM backends.

We have demonstrated it is possible to write GPU kernels in Rust. Our work is still very much in the proof of concept stage, so many improvements are still necessary in order to have a truly useful system.