Project Polyglot

Henri Malahieude

Project Idea / Overview

Explore the different GPU programming libraries available in different languages. The initial list contains Rust-CUDA Project, WebGPU in Rust (wgpu.rs), and Zig Cuda (cudaz). What will be explored is the methodology of the libraries/APIs and the maturity of the tool. If the experimental library is successfully deployed/installed, performance will be gauged with a Tiled Matrix Multiplication implementation.

How is the GPU used to accelerate the Application?

It will be used to explore the maturity of each library. If the software library can utilize shared memory, two dimensional gridding, and simply executing the kernel.

Implementation Details

The basic process was simple: go through each library and API, get as far as I could, and repeat for each project until it was done. All while learning the language as I was going through it.

Problems

The first problem I faced was uncompilable code, specifically Rust-CUDA was the first experimental library that I tried. Unfortunately, I could no longer compile the code because it was no longer maintained.

Another issue is that some are primarily graphics libraries, so their tooling is not focused on General Purpose uses. They may have compute shaders, but they're not the primary focus of the libraries so there isn't support for controlling the amount of threads that are launched.

And another annoying issue is the way some libraries implement non-graphics related computation. Namely, in a different language and file. Meaning that I'd need to learn an extra "language" before being able to actually compute anything.

Documentation on Running Code

Unfortunately, Rust-CUDA cannot be run. Theoretically, it should be as simple as installing rust, then running **cargo run** in the Executor folder (which represents the CPU side code). However there is a package which refuses to compile because of competing rust versions.

For WebGPU, simply go into the folder of the respective program (hello-compute or matrix) you'd like to run, have rust installed, and **cargo run** for the fastest way to test. Hello Compute is the example that wgpu.rs shows off to get started with compute shaders. Matrix is my implementation of the naive matrix multiplication kernel.

Cudaz cannot be compiled, at least not with the most modern version of Zig (0.11.0). There isn't much that can be done but investigate the source code. Theoretically, the compilation should be as simple as **zig build**, or even **zig run**, within the folder of cudaz to first make the library, and then including that into your project's folder.

Results

Rust-CUDA Exploration

The tool with the most potential is most definitely Rust-CUDA. It is an effort to simply take rust, and compile it down to the PTX assembly. This PTX file could then be theoretically used in any system, without need for recompilation. Unfortunately the person maintaining the github is not maintaining it currently due to having other priorities at the moment, so there isn't much to really do about the failure of compilation for the package.

WebGPU with Rust (wgpu.rs)

The tool with the most functionality is wgpu.rs, the Rust Implementation of WebGPU. I could actually get the library to compile and run this time. That's as far as the pros go though. At its core, the library is really a graphics api with its key feature being the ability to compile for any Web Browser. It has its own Shader Language, which inhibits ease of use even if the language is rather similar to Rust or Javascript. In terms of the processing, WebGPU specification allows for "compute shaders" which are convoluted to work with compared to CUDA.

The big summary is that there is a ton of boilerplate code to integrate the two files, the main kernel launching Rust file, and the shader file. Here is a step by step walkthrough:

1. Launch a default instance of wgpu
2. Request device adapter
3. Acquire a device and an execution queue for the device
4. Create a shader module using the '.wgsl' shader file you designed

5. Create a storage buffer for every single parameter you'd like to pass through and receive
6. Create a "Compute Pipeline" representation of the connection to the Kernel
7. From this pipeline, attach each storage buffer to their associated group id and binding id
    a. This is called a "Bind Group Layout"
8. Then create a command encoder from the device
9. Use this command encoder to create a "compute pass" representing the kernel itself
10. Set the pipeline, on the compute pass
11. Set the bind group layout made earlier to the bind group id, on the compute pass
12. Have the compute pass "dispatch_workgroups" which is defining the amount of thread blocks to have the kernel launch
13. Add the encoder that "houses" the compute pass into the execution queue, so that it's executed
14. Specify a new "staging buffer" to copy data over from the Kernel back to the CPU
15. Specify a copy from buffer to buffer between the staging and the storage
16. Apply the equivalent to "_deviceSynchronize()" in wgpu terms
17. Jump 3 or 4 hoops to convert the staging buffer to useable output data

Steps 2-9 may be useful in a graphics shader context, but are completely unnecessary boilerplate.The entire project may be 250-300 lines for just a naive implementation of matrix multiplication, but it could be optimized thoroughly by simplifying the amount of random calls to "represent" everything. Just looking at the code I have implemented:

WGPU (compute shader)	CUDA
Kernel: **28 Lines**	Kernel: **34 Lines**
Device Setup: **109 Lines**	Device Setup: **6 Lines**

Kernel refers to the amount of lines for the GPU side, Device Setup refers to the amount of lines required to set up the variables and parameters on the Device before launching the kernel on the CPU side. CUDA requires a simple, allocate these variables, copy to these variables, and that's it. Don't get me started on the wgpu method. The Kernel isn't even in Rust, but "wgsl". I had to learn a third language…

And note that I didn't even implement a properly optimized matrix multiplication. Had it been a tiled matrix multiplication, the kernel code would've been longer. This is ultimately unacceptable for any application that wants to use GPU acceleration instead of its "intended" graphics use.

An even bigger kicker is that you cannot dynamically set the amount of threads each "workergroup" (thread block) will have. It is statically set within the "compute shader" with `@workgroup_size(x, y, z)`. That's alright, I guess, but then on top of that, the limit for the amount of "workers" (threads) per group (block) the shader can spawn is limited to 256. So much for parallelism.

The rust package's "compiling" of the shader was lackluster as well. With subpar compile-error responses, having me second guess a lot and have to experiment. And since documentation is sparse, I had to dig through tons of old Stack Overflow posts, forum threads, and horribly vague specification documents to even have an idea of what was going wrong. Oh, and I didn't find out until I decided, "maybe the compiler is doing some obscene optimization of variables" after an hour of banging my head. I didn't implement the "tiled" version of the matrix multiplication for this reason.

That being said, tested with the naive implementation on a M = K = N = 1000 matrix set up, it takes 886ms compared to 32490ms from the CPU. So it's not doing "nothing". But for the work required for this simple compute shader, CUDA knocks the competition out of the water.

My final thoughts on wgpu, is that it's not suited for these tasks. It's meant to be a graphics interface for designing more modern video games for web browser compatibility. The ease of use seems to be best for writing graphics shaders, and interacting with visual aspects of an application. Rather than extreme throughput calculations, even if it "supports" functionality. Of course, this isn't the final release or specification. The specification for "WebGPU" for general implementation in all browsers is still a "working draft". And wgpu's version is still v0.16. So there could still be some developments to improve this. However I did not enjoy this foray, especially considering where I came from (CUDA), and I sincerely hope it improves in the future for other non-graphics related purposes.

Cudaz (Zig Cuda Wrapper)

This is unfortunately another tool that cannot be compiled. It's a very interesting idea, which bases itself off of the main selling point of Zig.

The Zig Language places itself as a modern language like C/C++, but it is different than Rust or Go in its execution. Zig can be translated into C, and the Zig compiler can translate Zig into C. Zig can import C libraries with no issue, and Zig can be imported into C programs with little overhead. The basic concept is that Zig is trying to make it as easy as possible for C/C++ programmers to shift over to Zig.

The main problem with Zig is that it is very new, as in v0.11.0, and there isn't a clear roadmap for when v1.0 will be released. Not to mention that there is a new release every 6 months, so I'm not too sure which version of Zig this library was created for. It's fascinating though that the library first compiles the kernel into a PTX file that's then linked into the system. This seems to be a reoccurring theme.

Conclusions

The current state of coding for GPUs with Rust is unfortunately not stable. Rust posits itself as an extremely safe, performative, and robust language. Unfortunately, the CUDA implementations for Rust are either uncompilable, or convoluted. But I still recommend CUDA C if you are looking for the fastest way to program, with yourself having to catch your own errors instead of a fancy compiler. A principled programmer can avoid most pitfalls that a Rust compiler could catch anyways.

As for Zig, it's a fascinating concept, that I cannot get to compile/build. If it reaches a stable codebase, and the project is ever revisited, I think it could very well be used in the future for GPU development. It's low-level and the integration with C is almost flawless.

Overall, if there is one thing I learned from this project: All the other libraries required a PTX/GPU file first, which is then added into the CPU code later. There is a specific barrier between GPU and CPU code, which I imagine stems from their use of the NVIDIA dlls to compile into PTX assembly.

Project Work Breakdown

Learning the Languages:	10%
Rust CUDA Project:	5%
WebGPU (w/ Rust):	80%
Zig Cuda (cudaz):	5%
Total:	**100%**