ECE459: Programming for Performance Winter 2021

Lecture 22 — GPU Programming Continued

Patrick Lam & Jeff Zarnett *2020-10-20*

GPUs: Heterogeneous Programming

Host Code

We've learned about how a kernel works and a bit about how to write one. The next part is the host code, Now, fortunately, we don't have to write the whole program in C++ or C, even though the kernel has to be written in the CUDA variant. We're going to use the Rustacuda library from `https://github.com/bheisler/RustaCUDA`.

We'll look at a quick example of launching a very simple kernel from the Rustacuda examples[1]:

```
#[macro_use]
extern crate rustacuda;

use rustacuda::prelude::*;
use std::error::Error;
use std::ffi::CString;

fn main() -> Result<(), Box<dyn Error>> {
    // Set up the context, load the module, and create a stream to run kernels in.
    rustacuda::init(CudaFlags::empty())?;
    let device = Device::get_device(0)?;
    let _ctx = Context::create_and_push(ContextFlags::MAP_HOST | ContextFlags::SCHED_AUTO, device)?;

    let ptx = CString::new(include_str!("../resources/add.ptx"))?;
    let module = Module::load_from_string(&ptx)?;
    let stream = Stream::new(StreamFlags::NON_BLOCKING, None)?;

    // Create buffers for data
    let mut in_x = DeviceBuffer::from_slice(&[1.0f32; 10])?;
    let mut in_y = DeviceBuffer::from_slice(&[2.0f32; 10])?;
    let mut out_1 = DeviceBuffer::from_slice(&[0.0f32; 10])?;
    let mut out_2 = DeviceBuffer::from_slice(&[0.0f32; 10])?;

    // This kernel adds each element in `in_x` and `in_y` and writes the result into `out`.
    unsafe {
        // Launch the kernel with one block of one thread, no dynamic shared memory on `stream`.
        let result = launch!(module.sum<<<1, 1, 0, stream>>>(
            in_x.as_device_ptr(),
            in_y.as_device_ptr(),
            out_1.as_device_ptr(),
            out_1.len()
        ));
        result?;
    }

    // Kernel launches are asynchronous, so we wait for the kernels to finish executing.
    stream.synchronize()?;

    // Copy the results back to host memory
    let mut out_host = [0.0f32; 20];
    out_1.copy_to(&mut out_host[0..10])?;
    out_2.copy_to(&mut out_host[10..20])?;

    for x in out_host.iter() {
        assert_eq!(3.0 as u32, *x as u32);
```

[1] `https://github.com/bheisler/RustaCUDA/blob/master/examples/launch.rs`

```
    }

    println!("Launched␣kernel␣successfully.");
    Ok(())
}
```

And the kernel it corresponds to is[2]:

```
extern "C" __constant__ int my_constant = 314;

extern "C" __global__ void sum(const float* x, const float* y, float* out, int count) {
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < count; i += blockDim.x * gridDim.x) {
        out[i] = x[i] + y[i];
    }
}
```

Walk-through. Let's look at all of the code in the example and explain the terms. For a more detailed explanation of all the steps, see [Cor20].

One thing that we won't see is any explicit call to initialize the runtime. The CUDA runtime is automatically initialized when there's the first call into it. Thus, the first call might report errors that you wouldn't expect from that call, because they are really a setup problem on your system.

Right, in the example, we need to include the prelude for Rustacuda, just as we've seen previously for Rayon. This imports a bunch of commonly used types to save on having a lot of imports.

First thing we have to do is initialize he API. This has to happen at the start of the program, so no sense in delaying it! At present, there are no flags defined so the call with `CudaFlags::empty()` is the only valid argument to the initialization function.

Then we get a *device*. The device is your graphics card or any other hardware that does the work. It's obviously possible to have more than one compute device, but we'll just take the first one.

Next, in step 3, we request a *context*. The context is described by the documents as analogous to a process. When we launch something within that context, it executes in there and has its own address space, and when the context is destroyed, all its resources are cleaned up. Each host thread may have one context. The call shown is "create and push" because a host thread has a stack of current contexts. We don't actually need the context for any of the later steps, but we just need to be in possession of one. In this simple example, stays in scope because everything happens in `main`. If you want more structure to your program, then you do have to ensure it doesn't go out of scope and get dropped, because it has to exist for the other functions to work. When it does get dropped, some cleanup actions take place, of course.

The next step is to create our *module*. A module is technically a package of code and data to run on the device, but in this case it just really means our kernel. What we do here is read the compiled PTX code into a C-String, then create the module from that string. It's also possible to load from a file directly.

Once we have created a module, we then create a *stream*. The stream is where we assign work. You can think of it as being similar to a priority queue: you put items in there and work that is more important (lower priority number) can pre-empt work that is less important (higher priority number). Otherwise, work is done in the order it was scheduled, and tasks don't overlap. The stream is asynchronous, so once work has been assigned, it returns immediately. I'll point out that the Rustacuda implementation says you should set the `NON_BLOCKING` flag to prevent synchronization with the NULL stream (the details of which we'll skip).

There's one more step before launching the kernel; in step 5, we create some *data buffers*, which are used for moving data to and from the GPU. Remember, CUDA requires explicit communication, so whatever data want to

[2] `https://github.com/bheisler/RustaCUDA/blob/master/resources/add.cu`

provide as input has to be put into a buffer and then the buffer is transferred to the kernel. Whatever data comes as output will be transferred by the GPU into the output buffers we specify.

After all this setup, we can finally launch the kernel. This has to be done in an `unsafe` block, because the launch macro is unsafe (unfortunately). The good news is that the unsafe block is only the launch, limiting the area of extra scrutiny to something small. When we launch, we specify the kernel that's supposed to run as well as the arguments. Each buffer is converted using the `as_device_ptr()` so that the contents of the device buffer are provided. For scalar types like the count, no such conversion is necessary and we can just provide the value.

Great! We launched the kernel and sent it over to the GPU. This is an asynchronous process, so we could do more stuff here if we need. There's nothing else to do at the moment, so we'll wait for the items in the queue to complete by calling `stream.synchronize()`. Straightforward!

Finally, in the last step, we copy the items out of the buffer and back into host memory. Here, the example code checks that all the values are correct (3.0) and it is! Alright, we have a simple working example of how to setup, launch, and collect results from a CUDA computation!

More Complex Host Code

Here's an alternate example of the unsafe block for launching, This function form gives us some input into how we want the grid and block size to be organized.

```
// Launch the kernel again using the `function` form:
let function_name = CString::new("sum")?;
let sum = module.get_function(&function_name)?;
// Launch with 1x1x1 (1) blocks of 10x1x1 (10) threads, to show that you can use tuples to
// configure grid and block size.
let result = launch!(sum<<<(1, 1, 1), (10, 1, 1), 0, stream>>>(
    in_x.as_device_ptr(),
    in_y.as_device_ptr(),
    out_2.as_device_ptr(),
    out_2.len()
));
result?;
```

More Dimensions

Synchronization. You might define workgroups because you can only put barriers and memory fences between work items in the same workgroup. Different workgroups execute independently.

OpenCL also supports all of the notions that we've talked about before: memory fences (read and write), barriers, and the volatile keyword. The barrier (`barrier()`) ensures that all of the threads in the workgroup all reach the barrier before they continue. Recall that the fence ensures that no load or store instructions (depending on the type of fence) migrate to the other side of the fence.

References

[Cor20] Nvidia Corporation. Cuda c++ programming guide, 2020. Online; accessed 2020-10-15. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.