Lecture 22 — GPU Programming Continued

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

February 4, 2024

The 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!

Rust Host Code Example

```rust
#[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::DEFAULT, 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])?;
```

Rust Host Code Example

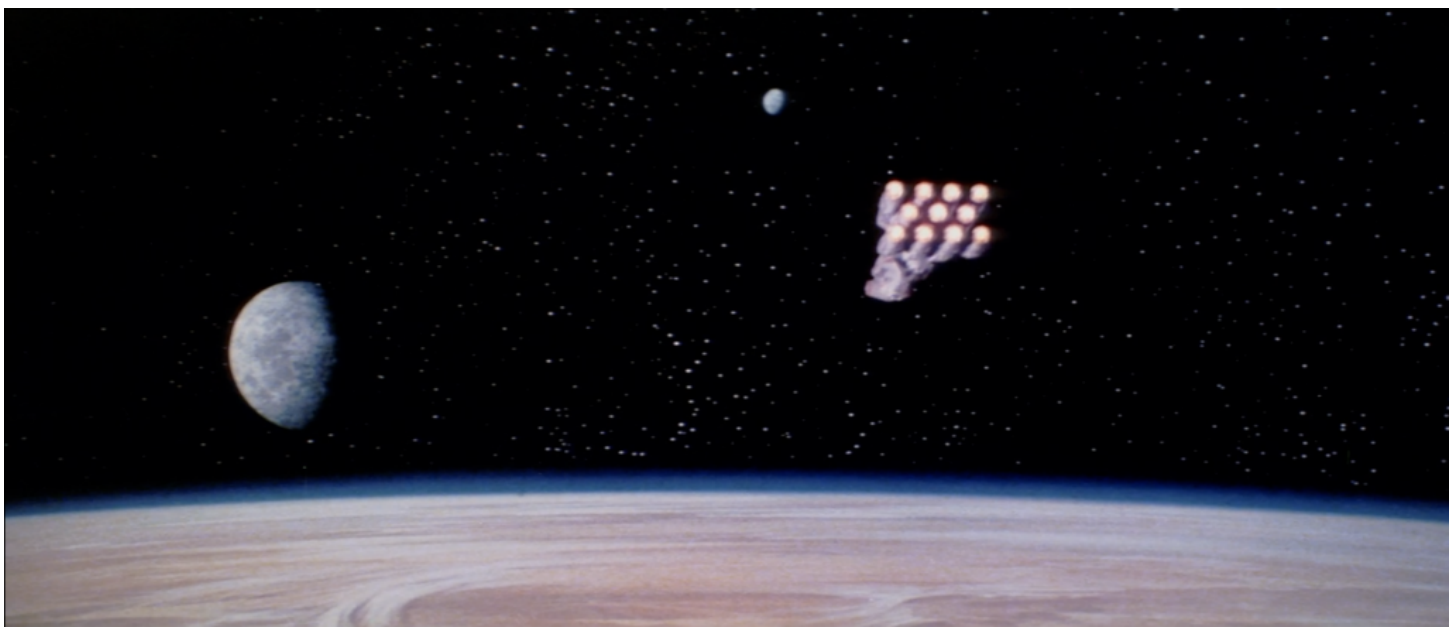
```
// This kernel adds each element in `in_x` and `in_y` and writes the
    result into `out`.
unsafe {
    // Launch the kernel with one block of one thread, no dynamic shared
        memory on `stream`.
    let result = launch!(module.sum<<<1, 1, 0, stream>>>(
        in_x.as_device_ptr(),
      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

Rust Host Code Example

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

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

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

Corresponding Kernel

```
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];
    }
}
```



N-Body Problem

Let’s take a look at the N-Body Problem code in the repo.

Results

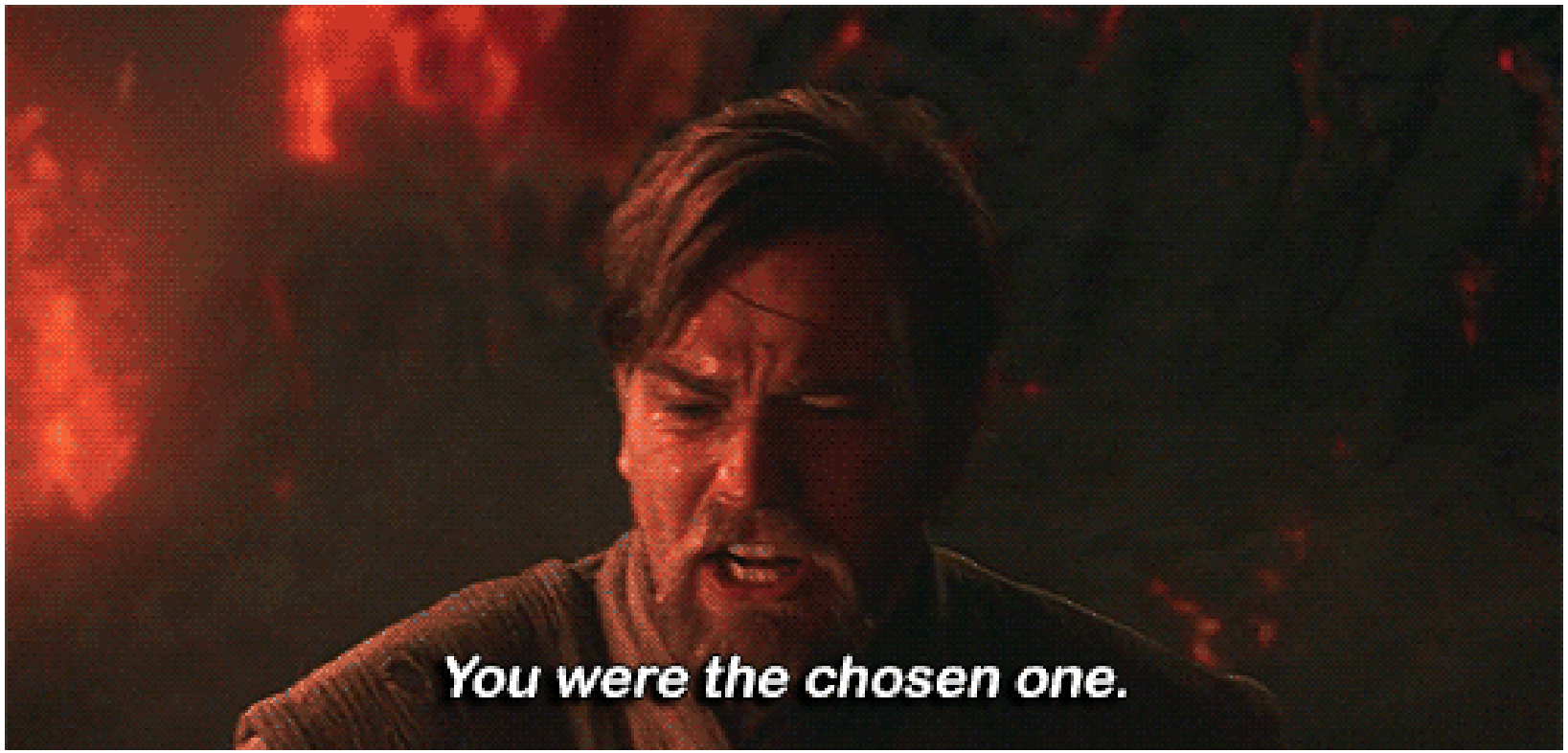
That's great, but how much does it speed up?

I ran this on ecetesla1.

With 100 000 points:

- `nbody` (sequential, no approximations): 40.3 seconds
- `nbody-parallel` (parallel, no approximations): 5.3 seconds
- `nbody-cuda` (parallel, no approximations): 9.5 seconds

Wait a Minute

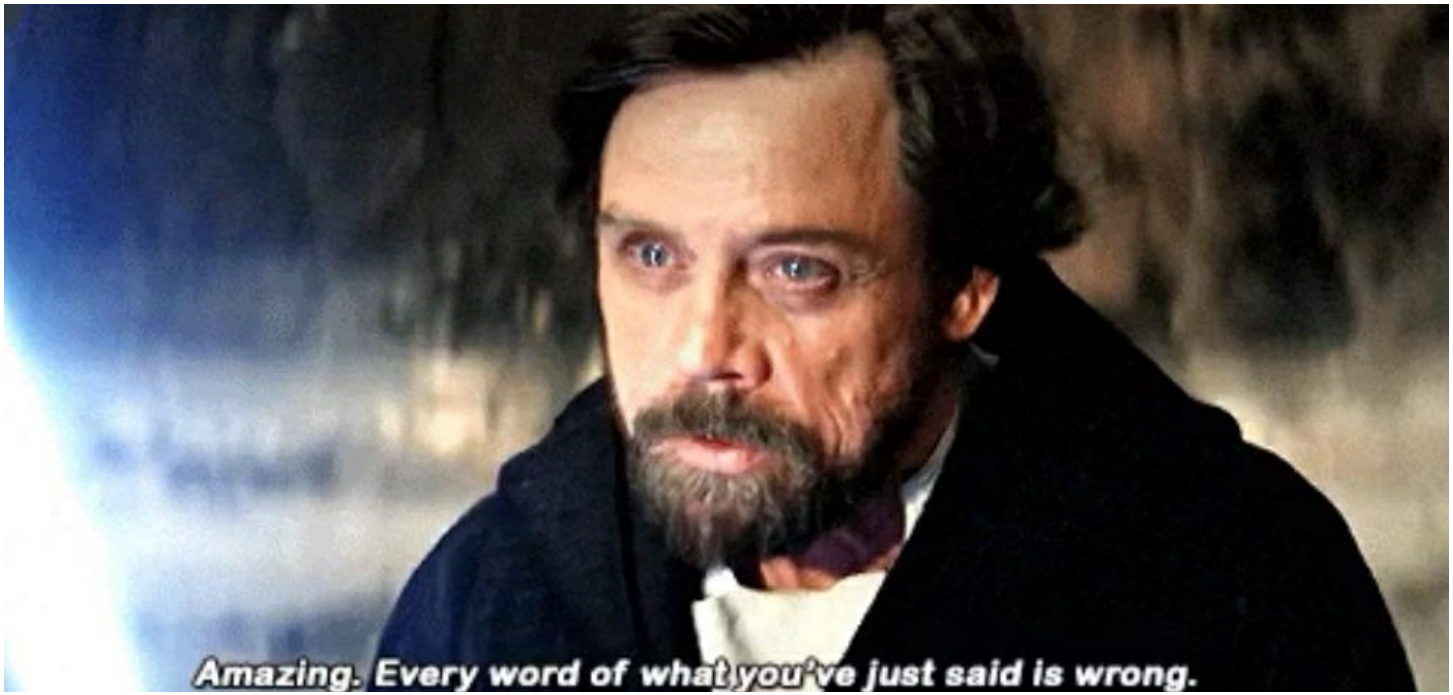


That’s worse than the CPU version.

Kylo Ren is very Meme-Able

Theory: 100 000 points is not enough to overcome the overhead of setup and transferring data to and from the device.

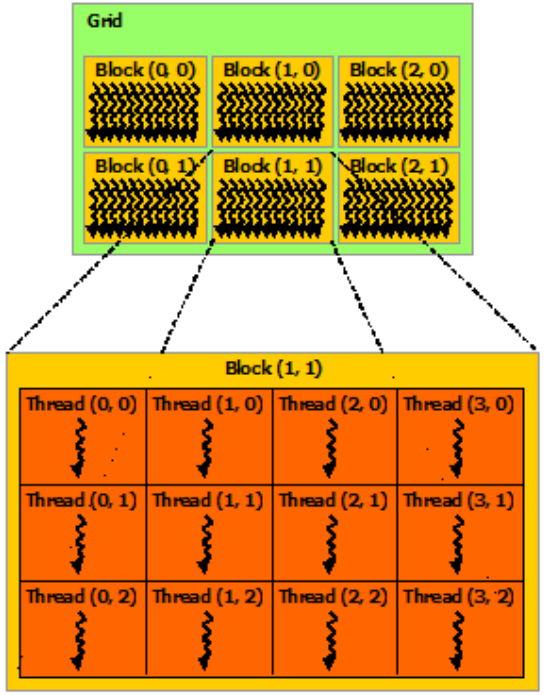
Sequel Memes!



This turned out to be incorrect.

Most of the time was going to the kernel execution of the `calculate_forces` – as expected?

This Was Important



The documentation isn't super great about how grids and blocks work.

A lot of the guidance on the internet says is "experiment and try".

Grid Guidance

The initial approach that I wrote had each work-item be its own grid.

That's inefficient, because we're not taking advantage of all the hardware that's available (the warp hardware).

The advice that I can find says the number of threads per block should always be a multiple of 32 with a maximum of 512 (or perhaps 1024).

The second guidance I can find is that numbers like 256 and 128 are good ones to start with, and you can tweak it as you need.

Then you have to adjust the grid size: divide the number of points by the threads per block to give the number of blocks.

Change the Launch

```
let result = launch!(module.calculate_forces<<<(NUM_POINTS/256) + 1,
     256, 0, stream>>>(
    points.as_device_ptr(),
    accel.as_device_ptr(),
    points.len()
));
result?;
```

I did have to add a +1 to the number of blocks, because 100 000 does not divide evenly by 256.

Kernel Crash

But just running it as-is didn’t work (and led to the kernel crashing. Why?

Because the indexing strategy that I used contained only the reference to the block index `blockIdx.x`.

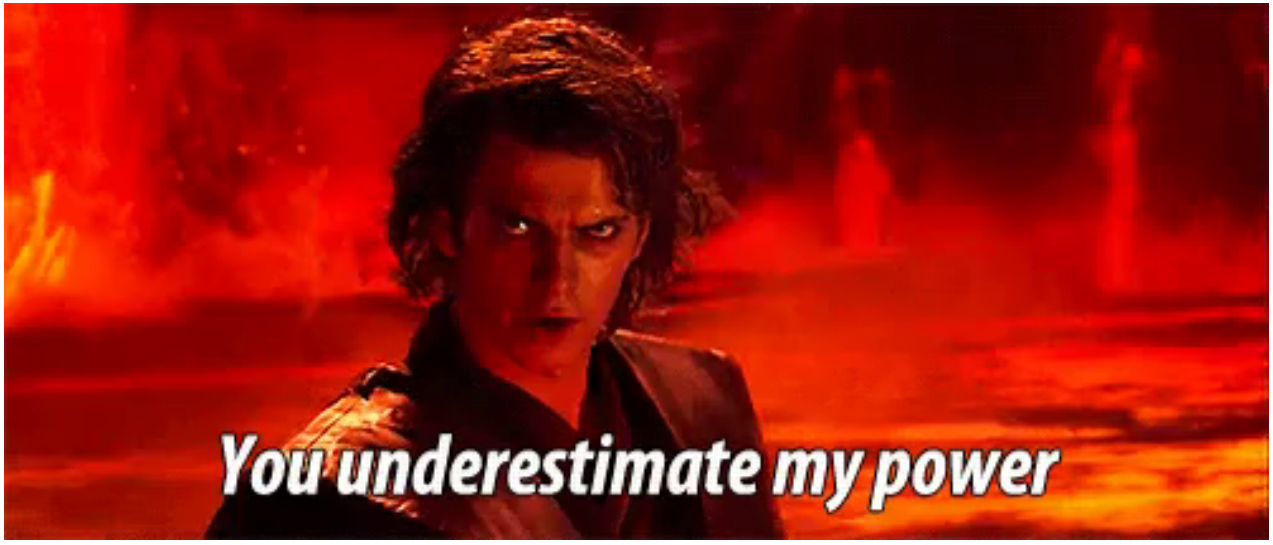
That’s fine in the scenario where every work-item gets its own block, but that’s no longer the case now: 256 work-items (points) now share each block.

Updated Kernel

```
extern "C" __global__ void calculate_forces(const float4* positions, float3*
     accelerations, int num_points) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx >= num_points) {
        return;
    }
    float4 current = positions[idx];
    float3 acc = accelerations[idx];

    for (int i = 0; i < num_points; i++) {
        body_body_interaction(current, positions[i], &acc);
    }
    accelerations[idx] = acc;
}
```

Did it work?

With 100 000 points:



- nbody (sequential, no approximations): 40.3 seconds
- `nbody-parallel` (parallel, no approximations): 5.3 seconds
- nbody-cuda (parallel, no approximations): 9.5 seconds
- **`nbody-cuda-grid` (parallel, no approx., grid): 1.65 seconds**

Now that's a lot better!

Trading Accuracy for Performance?

One more item from previous ECE 459 student Tony Tascioglu.

A crowd favourite in ECE 459 is trading accuracy for performance.

NVIDIA GeForce gaming GPU's don't natively support FP64 (double).

- Native FP64 typically requires $$$ datacentre GPUs.
- FP64 used to be locked in software, now missing in HW.
- Emulated using FP32 on gaming and workstation cards.

Trading Accuracy for Performance?

Using 32-bit floats rather than 64-bit doubles is typically a 16, 32 or even 64x speedup depending on the GPU!

Even more: 16 bit float instead of 32 bit is typically another 2x faster.

For many applications, double precision isn’t necessary!

Trading Accuracy for Performance?

How dramatic is the difference?

GPU	FP32 GFLOPS	FP64 GFLOPS	Ratio
GeForce RTX 3090	35580	556	FP64 = 1/64 FP32
GeForce RTX 3080	29770	465	FP64 = 1/64 FP32
Radeon RX 6900 XT	23040	1440	FP64 = 1/16 FP32
Radeon RX 6800 XT	20740	1296	FP64 = 1/16 FP32
GeForce RTX 3070	20310	317	FP64 = 1/64 FP32
GeForce RTX 3060 Ti	16200	253	FP64 = 1/64 FP32