Lecture 22 — GPU Programming

Patrick Lam and Jeff Zarnett
patrick.lam@uwaterloo.ca, jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 4, 2022

Introduction

CUDA: coding on a heterogeneous architecture. No longer just programming the CPU; will also leverage the GPU.

CUDA = Compute Unified Device Architecture.
Usable on NVIDIA GPUs.

Note: RTX 3080 not available at the time of writing.
People probably sell pictures of them on eBay.

SIMT

Another term you may see vendors using:

- **S**ingle **I**nstruction, **M**ultiple **T**hreads.
- Runs on a vector of data.
- Similar to SIMD instructions (e.g. SSE).
  However, the vector is spread out over the GPU.

Other Heterogeneous Programming Examples

- PlayStation 3 Cell
- OpenCL

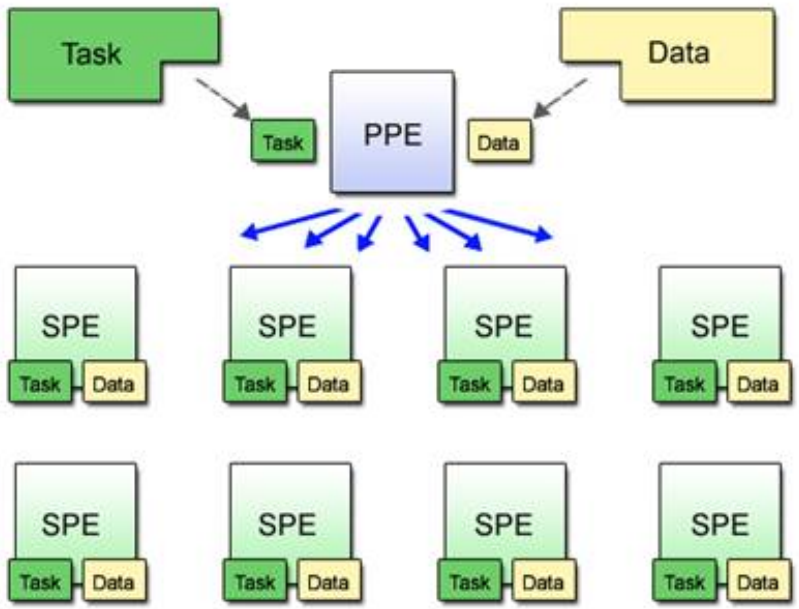
See `https://www.youtube.com/watch?v=zW3XawAsaeU` for details on why it was hard to program for the PS3!

[PS4: back to a regular CPU/GPU system, albeit on one chip.]

(PS3) Cell Overview

Cell consists of:

- a PowerPC core; and
- 8 SIMD co-processors.



(from the Linux Cell documentation)

CUDA Overview

Compute Unified Device Architecture:
NVIDIA's architecture for processing on GPUs.

"C for CUDA" predates OpenCL,
NVIDIA supports it first and foremost.

- May be faster than OpenCL on NVIDIA hardware.
- API allows you to use (most) C++ features in CUDA;
  OpenCL has more restrictions.

GPU Programming Model

The abstract model is simple:

- Write the code for the parallel computation (*kernel*) separately from main code.
- Transfer the data to the GPU co-processor (or execute it on the CPU).
- Wait ...
- Transfer the results back.

Drive or Fly?

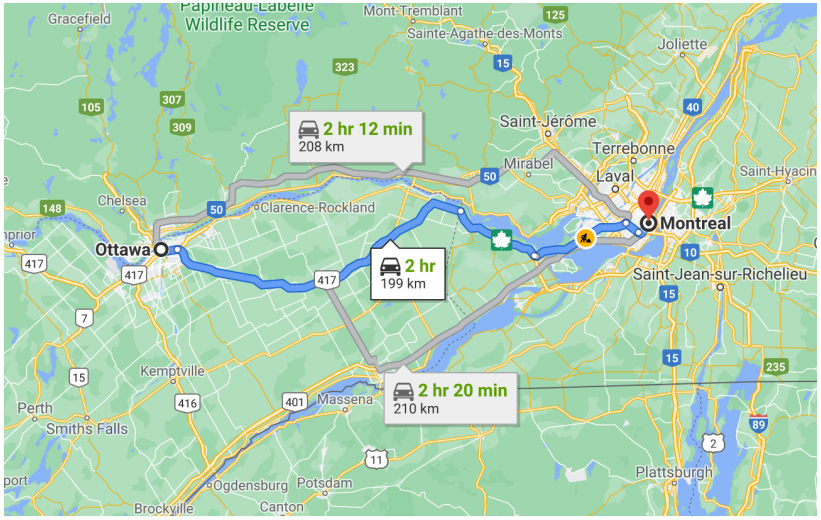
It makes sense to hand it over to the GPU because there are a lot of cores.

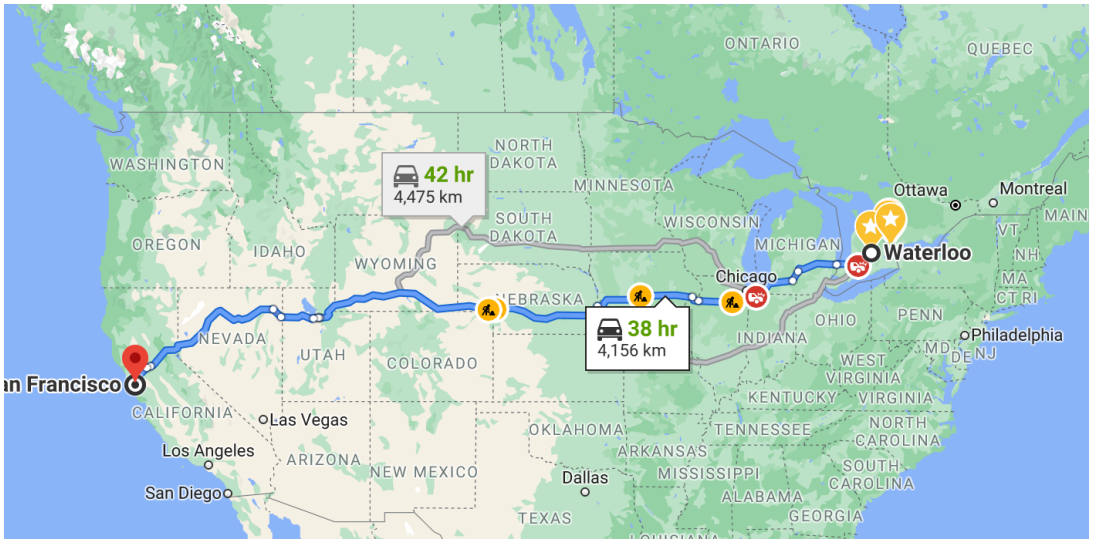
`ecetesla2`: 4 core 3.6 GHz CPU; 1920 CUDA cores that run at about 1.8 GHz.

So, half the speed, but 480 times the workers.

Drive or Fly?

There's significant runtime overhead but the GPU is fast once started.
This is like driving vs flying.





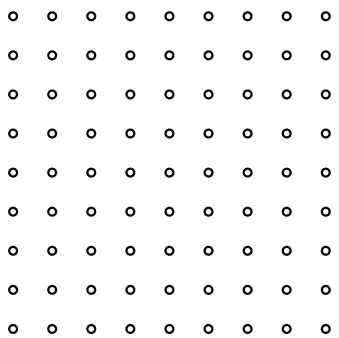
CUDA Parallelism

CUDA includes both task parallelism and data parallelism, as we've discussed earlier in this course.

Data parallelism is the central feature.

Data Parallelism

- Key idea: evaluate a function (or *kernel*) over a set of points (data).



Another example of data parallelism.

- Another name for the set of points: *index space*.
- Each point corresponds to a **work-item**.

Note: CUDA also supports *task parallelism* (using different kernels), but we won't focus on that in this course.

Work-Items

Work-item: the fundamental unit of work in CUDA.

These work-items live on an *n*-dimensional grid (ND-Range).

"Ready to work!"

Work-Items

You get your choice about block size.

Usually, we say let the system decide.

However, for some computations it's better to specify.

Ff you do, however, you want to make best use of the hardware and use a multiple of the size of the *warp*.

Shared Memory

There are many different types of memory available to you:

- private
- local
- global
- constant
- texture

Choosing what kind of memory to use is important!

There is also host memory (normal memory);
usually contains app data.

A kernel of my very own...

Here's the C++ code we start with:

```
void vector_add(int n, const float *a, const float *b, float *c) {
  for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
}
```

The same code looks like this as a kernel:

```
__global__ void vector_add(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

The loop became implicit!

Going Backwards?

You can write kernels in a variant of C++.

A large number of features of more recent versions of the language are supported.

How much will vary based on your release of the kernel compiler (nvcc).

It's unlikely that we'll be doing too many things that are exotic enough to be forbidden by the compiler.

The Compiled Kernel

The kernel is compiled into PTX – Parallel Thread eXecution – instructions.

Can we write our own? Yes. Will we? No.

Use the `nvcc` compiler.

Recompile if you change machines!

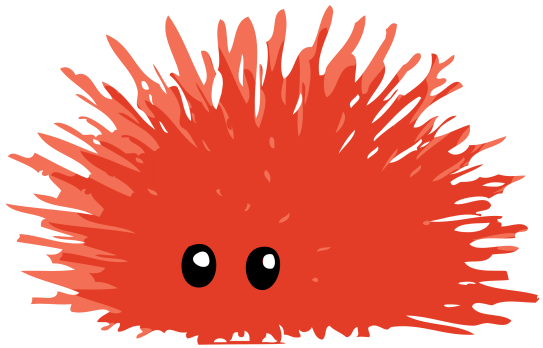
Running nvcc

The `nvcc` compiler has specific requirements for the underlying compiler it relies upon.

For example, on `ecetesla2` at the time of writing, `nvcc` fails without specifying `gcc` version 6 instead of the default.

With the command `nvcc -compiler-bindir /usr/bin/gcc-6 -ptx nbody.cu`, it will compile.

Danger! Danger!



The kernel has none of the safety guarantees that we normally get in Rust!

We can easily read off the end of an array, forget to initialize a variable, allocate memory that we forget to deallocate...

So, just because it compiles, doesn't mean it will work.

I broke it...

Here's something I got in testing:

```
thread 'main' panicked at 'Failed to deallocate CUDA Device memory.: IllegalAddress',
/home/jzarnett/.cargo/registry/src/github.com-1ecc6299db9ec823/rustacuda-0.1.2/src/memory/device/device_buffer.rs:259:32
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
thread 'main' panicked at 'Failed to deallocate CUDA Device memory.: IllegalAddress',
/home/jzarnett/.cargo/registry/src/github.com-1ecc6299db9ec823/rustacuda-0.1.2/src/memory/device/device_buffer.rs:259:32
stack backtrace:
```

N-Body Problem Kernel

Let’s take a look at a kernel for the N-Body problem!

We’ll also use that when we examine how to launch a CUDA program.

Practice Makes Perfect

Let's think about how to write the kernel well.

We need to decide what sort of work belongs in the kernel to begin with.

Loops good, as long as they're not sequential (some coordination is okay).

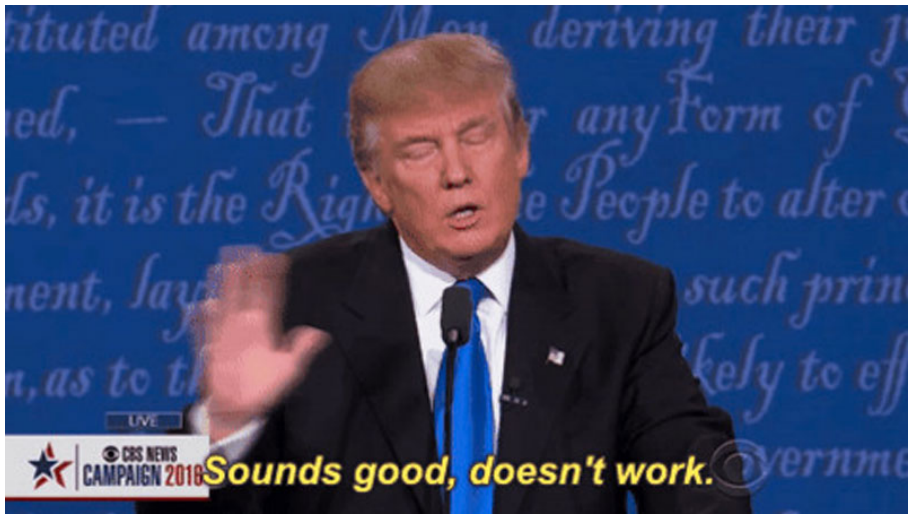
Two-Dimensions?

What if we wanted to make the N-Body kernel a two-dimensional problem?

We could instead treat each pair of points (i, j) as a point in the space, making it a two-dimensional problem.

Then you could think of it as a matrix rather than an array and provide it to CUDA like that. This might increase the parallelism!

Sounds Good, Doesn’t Work



The calculation of body-body interaction for just one pair of points is a very small amount of work, tiny even.

Having one work-item for each such calculation means there’s a lot of overhead to complete the calculation and it doesn’t make sense.

But if the calculation of the interaction were more complex, then this transformation might be an improvement!

MORE DIMENSIONS!

You can have a three-dimensional problem, but no more.

If you want something more than a three-dimensional, you have to have some loops in your code.

The limit to three dimensions is probably because graphics cards were designed for rendering images in 3-D so it seems logical?

Go the other way?

Of course, you can sometimes flatten the dimensions of your problem a bit.

A rectangular array in C/C++ is really stored as a linear array and the `[x][y]` is just notational convenience so you could easily just treat it as a linear array.

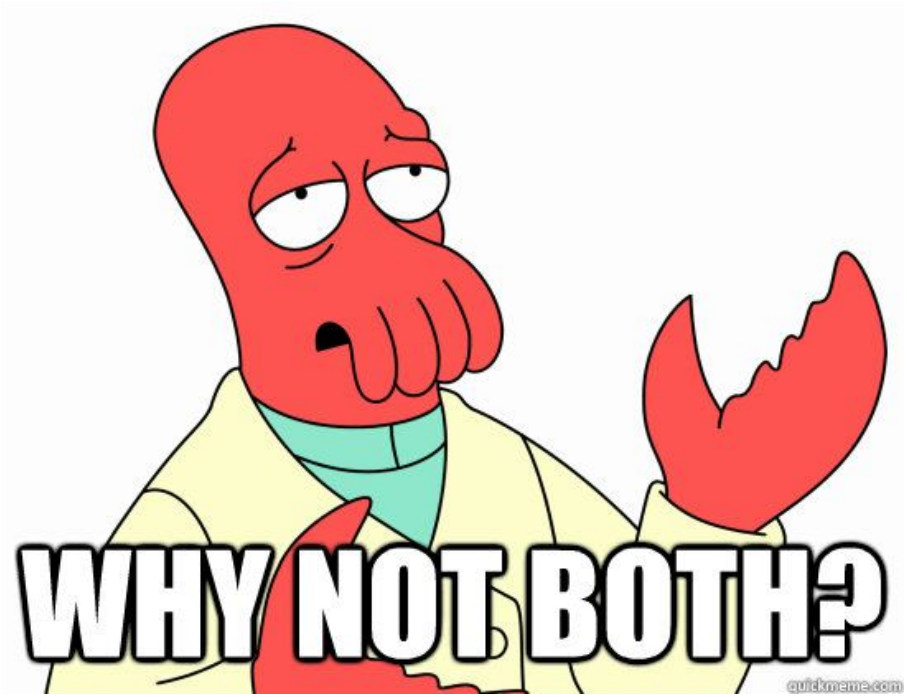
Brute Force a Password?

Consider something like brute-forcing a password of 6 characters (easy, but just for the purpose of an example).

A one-dimensional approach: generate all possibilities in host code.

Generate possibilities in kernel code

The hardware will execute *all* branches that any thread in a warp executes—can be slow!



Branches in kernels

```
__global__ void contains_branch(float *a, float *b) {
    if (condition()) {
        a[blockIdx.x] += 5.0;
    } else {
        b[blockIdx.x] += 5.0;
    }
}
```

In other words: an `if` statement will cause each thread to execute both branches; we keep only the result of the taken branch.

Loops in kernels

```
__device__ void foo(int *p1, int *p2) {
  for (int i = 0; i < 12; ++i) {
    p1[i] += p2[i]*2;
  }
}
```

A loop will cause the workgroup to wait for the maximum number of iterations of the loop in any work-item.

The compiler will try to unroll loops if it can.

Atomics

Race conditions can still occur.

This means that if you want to, say, concurrently add to the same location, you need to use atomic functions.

The atomic operations are usable on the standard primitive types.

There are the usual operations: add, sub, min, inc, dec, compare-and-swap, bitwise.

Atomic Example

```
__device__ double atomicAdd(double* address, double val) {
    unsigned long long int* address_as_ull =
                                (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;

    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                        __double_as_longlong(val +
                               __longlong_as_double(assumed)));

    // Note: uses integer comparison to avoid hang in case of NaN (since NaN
        != NaN)
    } while (assumed != old);

    return __longlong_as_double(old);
}
```

Launch?

So far, all we have covered is the theory and then how to write a kernel.

To make use of it, we'll have to look at the host code. That's our next topic.