Accelerating fiTQun with Graphics Processing Units

- Richard Calland – Kavli IPMU

- fiTQun Workshop, Boulder - 16th December 2014

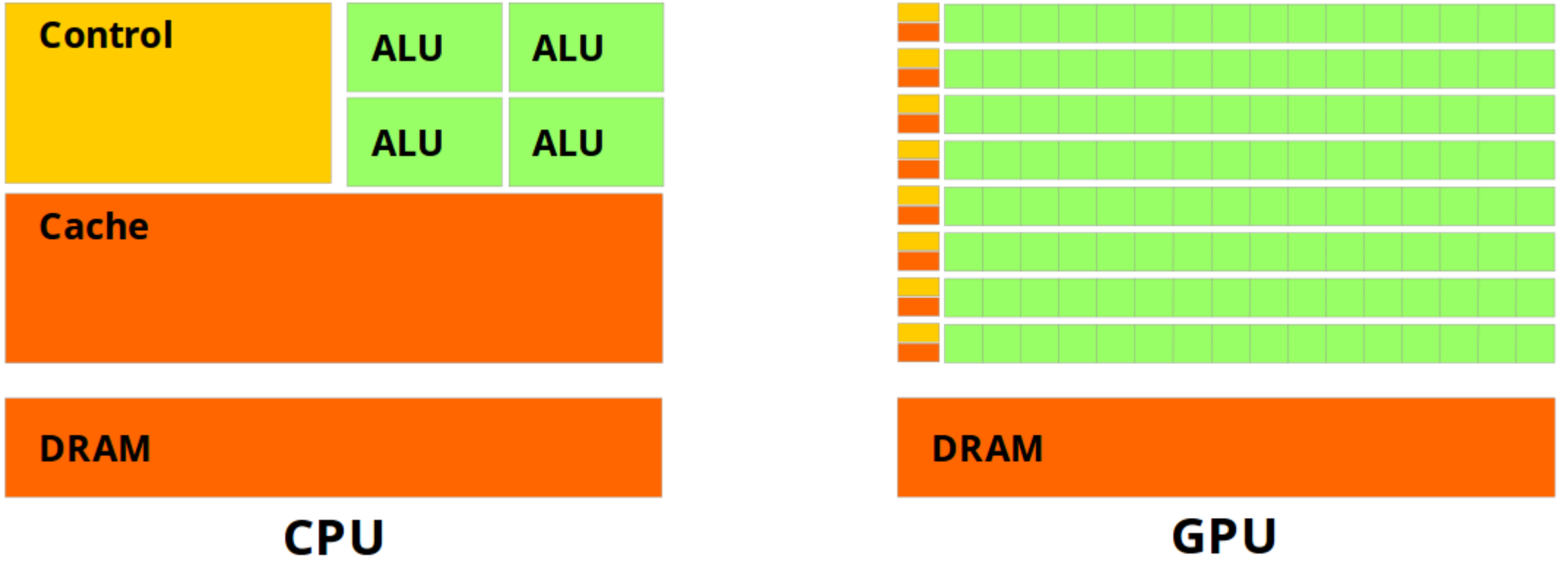
Introduction

- Introduction to GPUs
- Case study and personal experience
  - things I wish I knew at the beginning
- Preliminary fiTQun ideas
- **Discussion!**

Graphics Processing Units

- GPUs were designed to perform graphical operations
  - Vertex transforms
  - Lighting
- Eventually they became programmable
  - Games developers could write their own vertex and fragment shaders
- With modern architectures, this can be exposed for non-graphical computation

Comparison of CPU and GPU

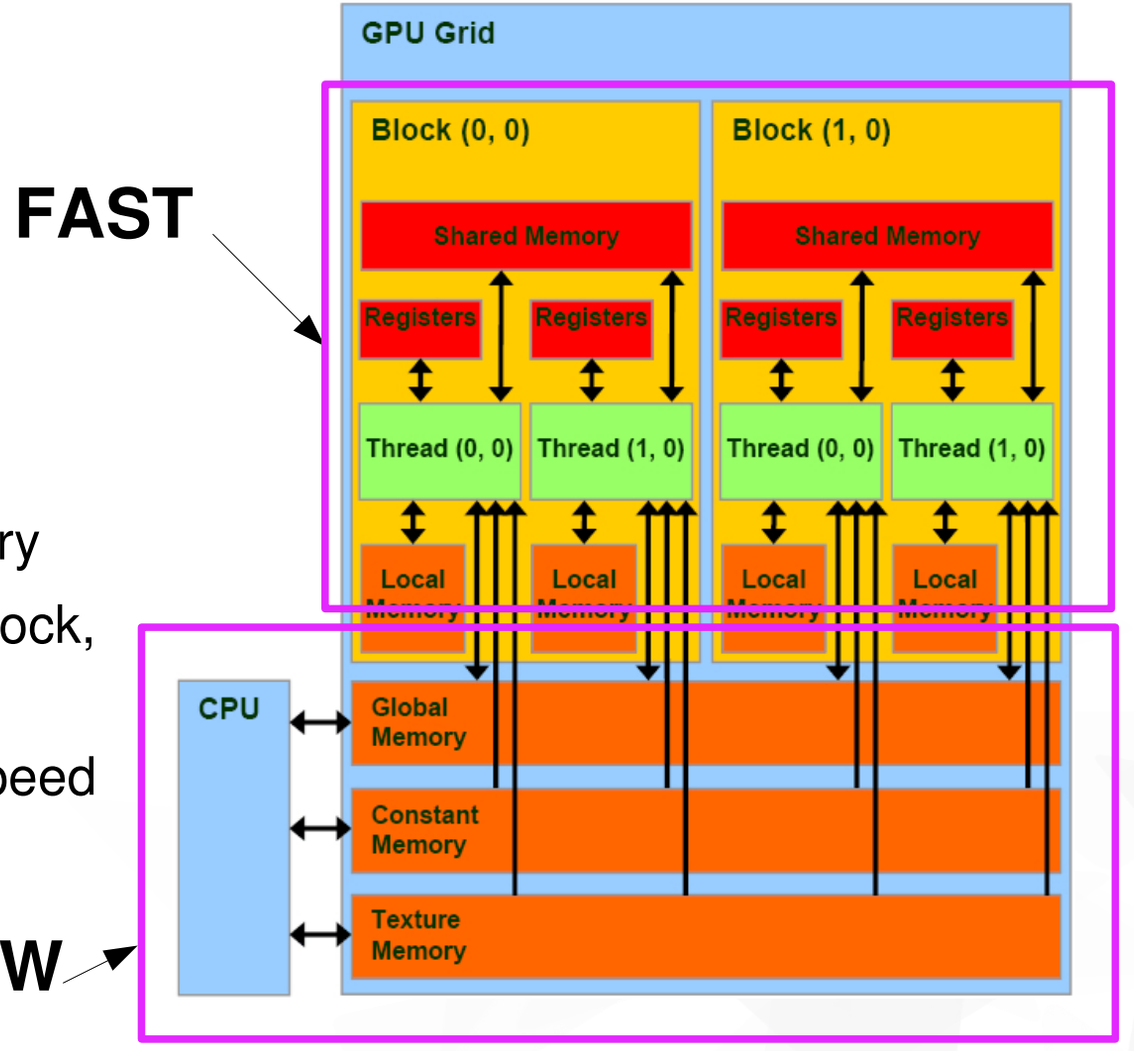


- CPU dedicates more transistors to program control, GPUs dedicate most to ALU
- CPU has ~4 powerful cores, GPUs have 100-1000s of less powerful cores
- Its like having 4 lamborghinis, vs. 500 scooters; Which would win in a drag race? Which would deliver 500 pizzas the fastest in a big city?

Compute Unified Device Architecture (CUDA)

A Closer look at the GPU architecture

- A GPU consists of many streaming multiprocessors (blocks), each containing cores (threads)
  - Threads are grouped in warps of 32(64)
  - Instructions are executed per warp
- Each block has its own shared memory cache
- Each thread has its own private registers
- All threads on the GPU can access the global memory
- Shared memory is fast; its not bound to the device clock, and allows communication between threads
- If you are use shared memory, you can gain more speed ups, although using global memory is sufficient



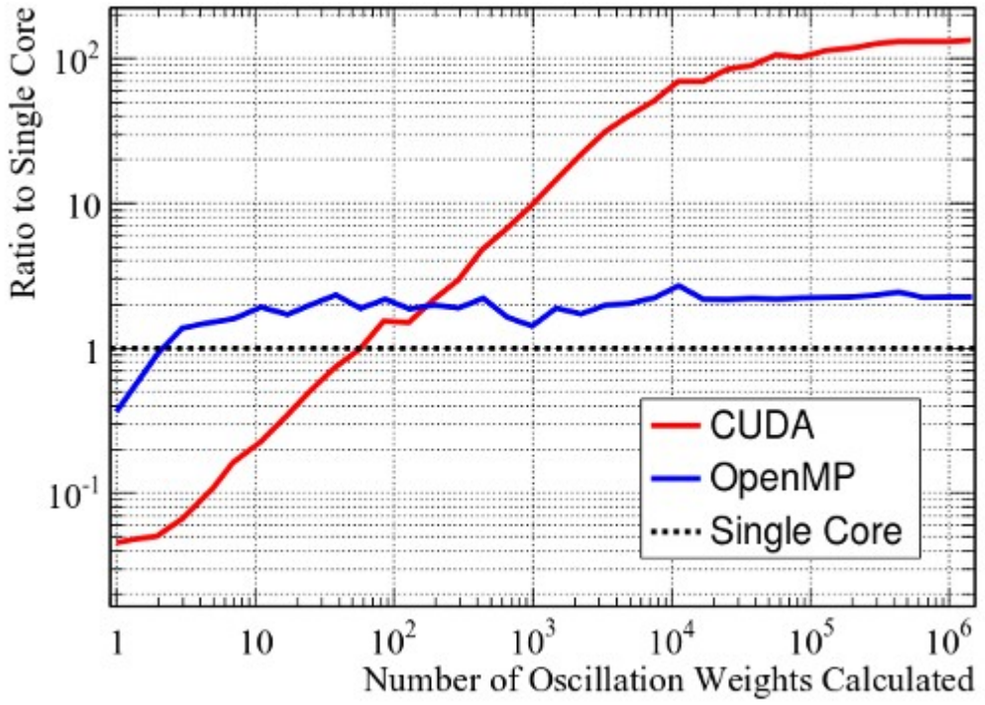
Accuracy

- GPUs can perform operations using single (float) or double precision
- Double precision is slower, but more accurate
- Comparing to CPU equivalent calculations, there will be a disagreement on the order of E-6/E-7 for floats and <E-12 for doubles
- CPU will always be more accurate due to extended arithmetic logic units (ALU) and the different hardware implementations (non-commutative arithmetic)
- Typically, floats have acceptable precision, but there are exceptions e.g. MINUIT

Experiences with GPUs

Calculating Oscillation Probabilities on GPU

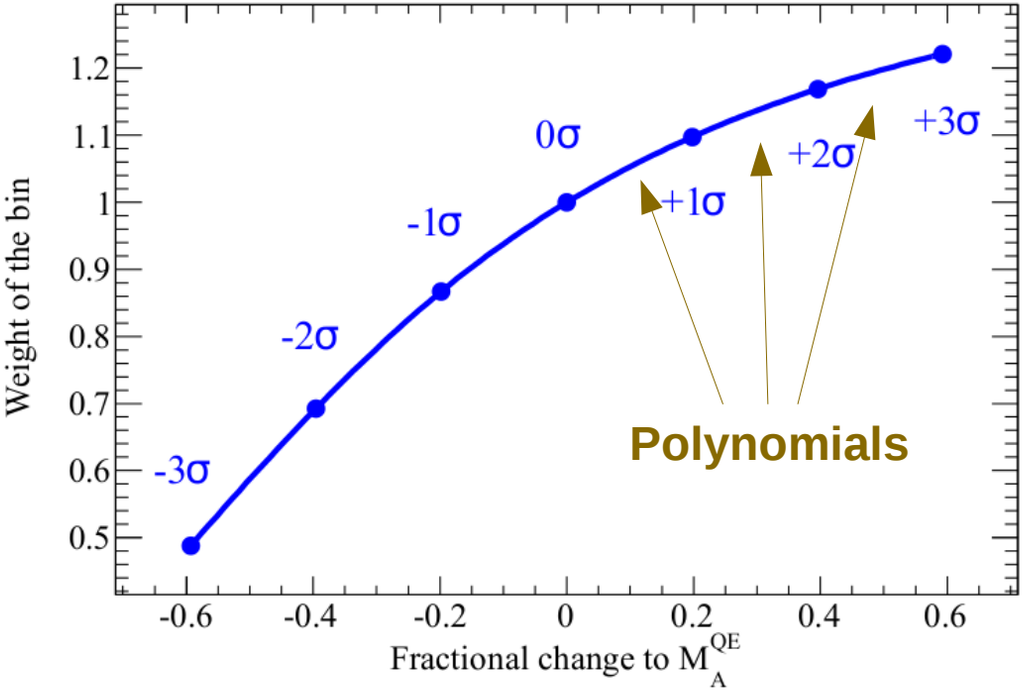
- In the MaCh3 oscillation analysis, we used GPUs to calculate the oscillation probabilities with matter effects on an event-by-event basis
- At numbers appropriate for the amount of T2K-SK MC, saw ~180x speed up compared to CPU execution in a standalone benchmark



`http://hep.ph.liv.ac.uk/~rcalland/probGPU/`

Parallel Cubic Spline Evaluation

- Offload ND280 splines for MaCh3 fit (~1 Gb)
- **First attempt:** evaluate one spline per thread
  - Find correct polynomial & evaluate
- **Second attempt:** evaluate one spline per block (group of threads)
  - Each thread evaluates a polynomial, *even though we only want the answer from one*
  - Correct polynomial is chosen afterwards, the rest are discarded
- Counter-intuitively, the second attempt was twice as fast!



Why was it faster?

- As mentioned, GPUs dedicate less transistors to control logic
  - As such, there is no branch prediction
  - Branches are when the code can go both ways, e.g. `if (true) {do this} else {do this}`
  - If GPU code branches, each group of 32 threads (warp) must evaluate both routes!
- Access to global memory is relatively slow; should minimize read/writes
- Each thread can evaluate a polynomial and store the result in shared memory
- Select the correct polynomial, but don't branch!

Branching code

```
if (x < y)
    {return a;}
else
    {return b;}
```

GPU must compute both routes*

Non-branching code

```
return (x < y) * a + !(x < y) * b;
```

Boolean logic, is either 1 or 0

*** If all threads in a warp `return a`, then there is no branching**

How to apply this to fiTQun

How to identify parallelizable code

- Look for *single instruction multiple data* (SIMD) operations
  - Calculating the same thing for many different independent elements
- GPUs are more suited to compute intensive vs. data intensive problems
- Fussy about data: structure of arrays, **not** arrays of structures
  - i.e. no arrays of C++ objects!
  - `TVector3 *v_array = new TVector3[10000]` **Slower**
  - `float x[10000]; float y[10000]; float z[10000]` **Faster**
- Sensible access patterns!
  - Thread `<n>` should access `x[<n>] y[<n>]` and `z[<n>]`
  - Accessing like `x[<n>] y[<n+1>] z[<n-10>]` will cause significant latency
- Random access patterns are bad (small constant cache memory is available for this)

Areas of fiTQun code that could be parallelized

- PMT loops in the likelihood (typically we are looking for >1000 parallel operations to see a speed up)
- Scattering table evaluation (GPU clusters typically have at least 6Gb VRAM)
  - Will require work to minimize latencies with memory access
- Change scattering tables for ray tracing (GPUs were made for this!)
  - This would be extremely fast on GPU
- **Any ideas from people with more experience with fiTQun?**