Accelerating fiTQun with GPUs

- Richard Calland – Kavli IPMU (UTokyo)

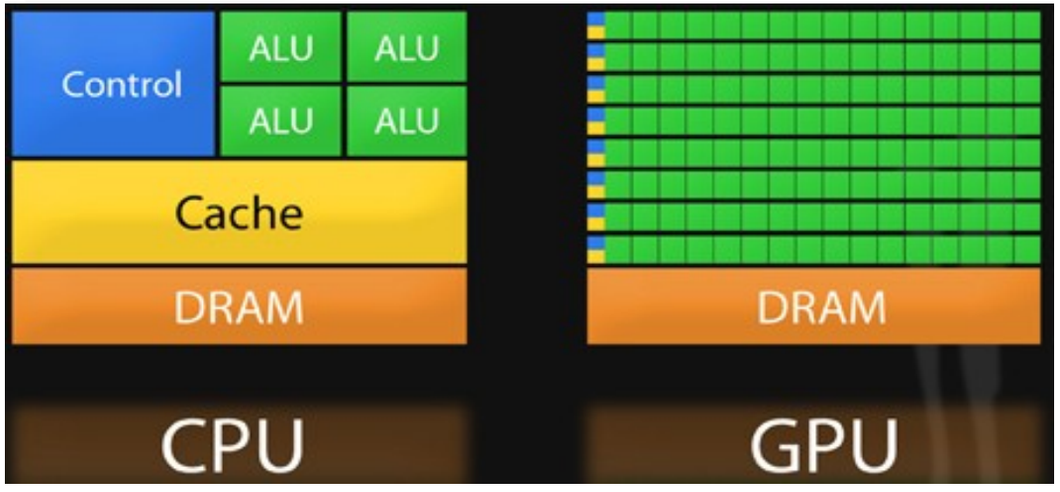
- fiTQun Workshop @TRIUMF - 18th August 2015

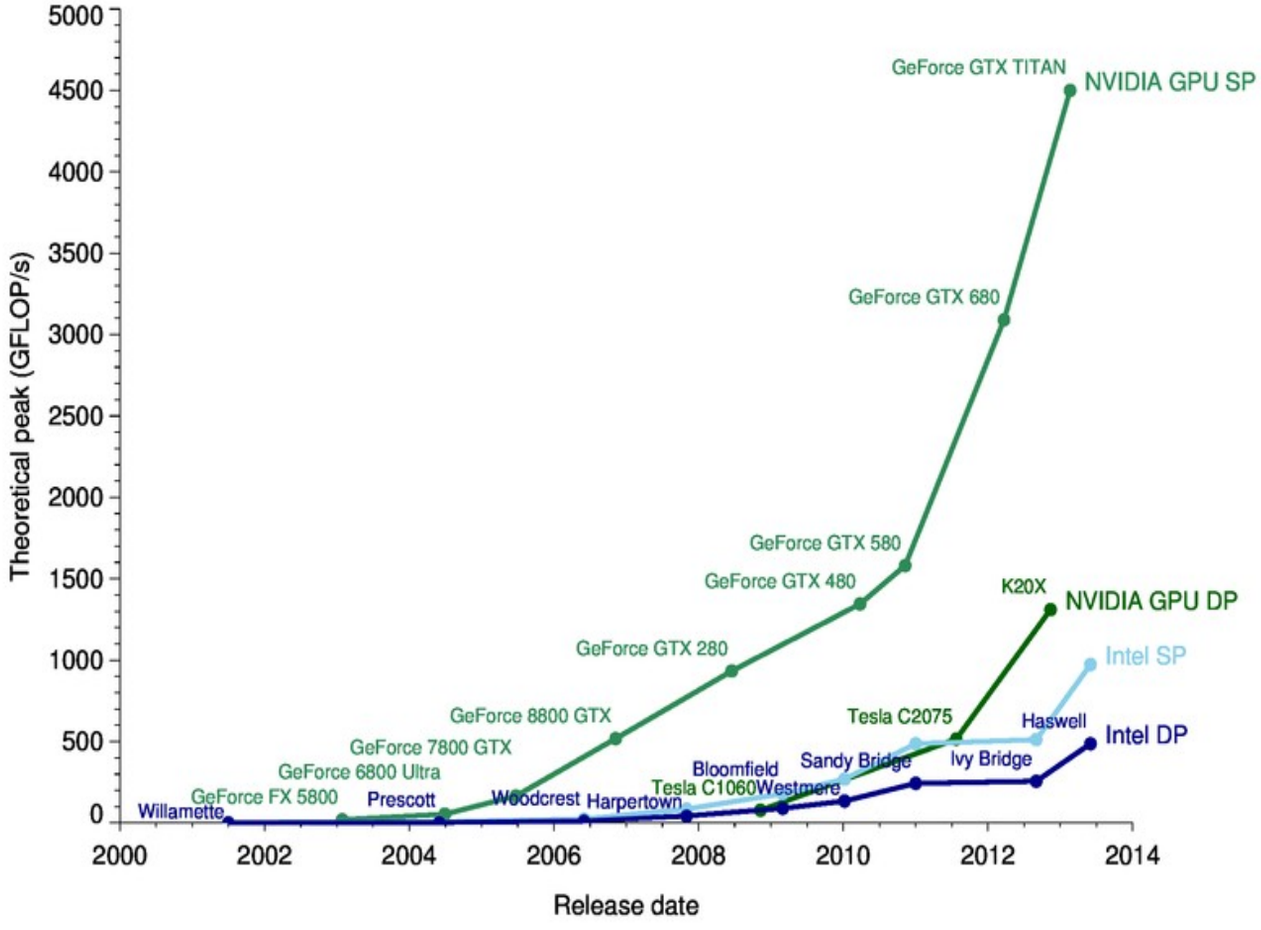
Outline of talk

- GPU architecture and how it is applicable to fiTQun
  - Breakdown of bottlenecks
- Scattering table
- Cherenkov profile
- General program flow
- Validations
- Benchmarks
- Future work

GPU Architecture

- GPUs have evolved into very powerful parallel machines for scientific computing
- 2000 cores vs 4 cores
- 300 Gb/s vs 6 Gb/s memory bandwidth





GPU issues

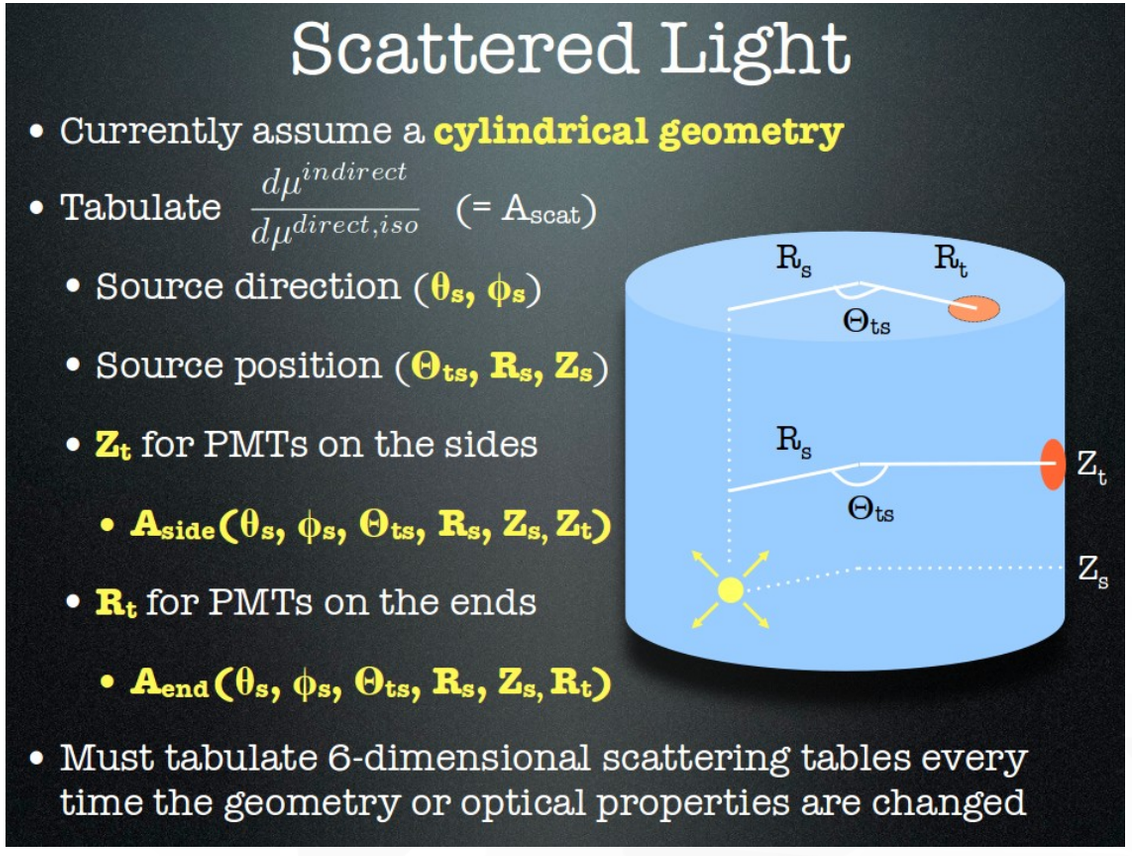
- The GPU dedicates far less transistors to control logic and cache
- This means the programs they run cannot be as intelligent as on a CPU; **commands are issued per 32 threads**
  - Memory accesses must be coalesced (i.e. not random)
  - Code that branches must be executed twice (e.g. once for *true*, once for *false*)
- However, if you can expose enough parallelization in your code, you can hide these latencies

Application to fiTQun

- FiTQun at its core is made from a for-loop that operates independently on each PMT
  - For each PMT, do:
    - A bunch of arithmetic (e.g. sqrt())
    - Indirect light calculation (scattering table)
    - Direct light calculation (cherenkov profile)
- We want to map each PMT onto a GPU core

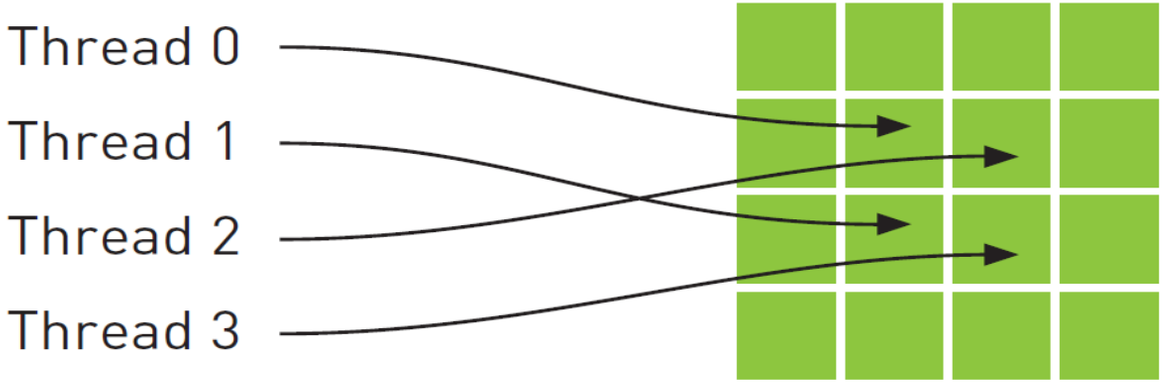
Scattering table

- Indirect light is calculated by performing a 6D linear interpolation for each PMT
- 50% of run time!
- For a given track, 3 of parameters never change
  - Source R, source Z, source theta
- This means that all PMTs will be contained within a sub-space of the 6D space
- Can perform 6D interpolation using 8 x 3D interpolations



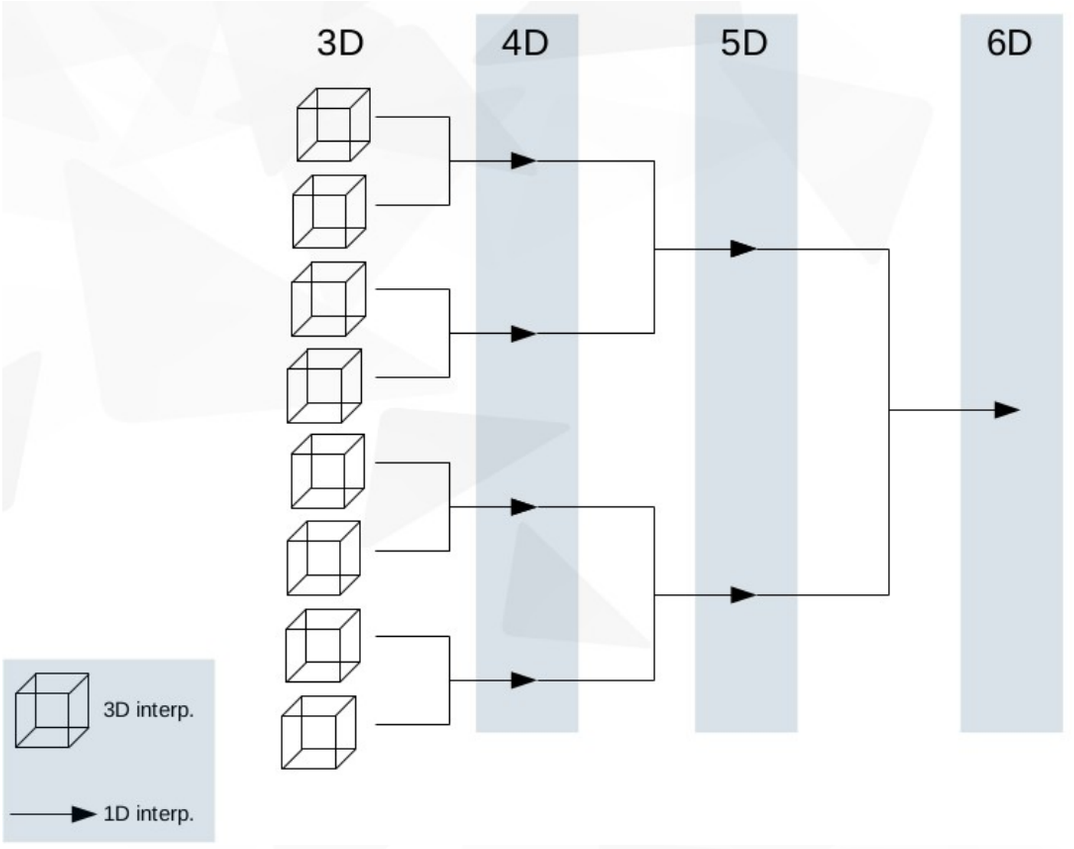
Why does this matter?

- GPUs have texture memory units
- Reads from RAM can be cached if there is spatial locality in the layout
- Can perform hardware linear interpolation
  - Free FLOPs! 3D interpolation at the cost of a single memory access
- Limited to 9-bit precision
- Corresponds to $\pm 5 \times 10^{-4}$

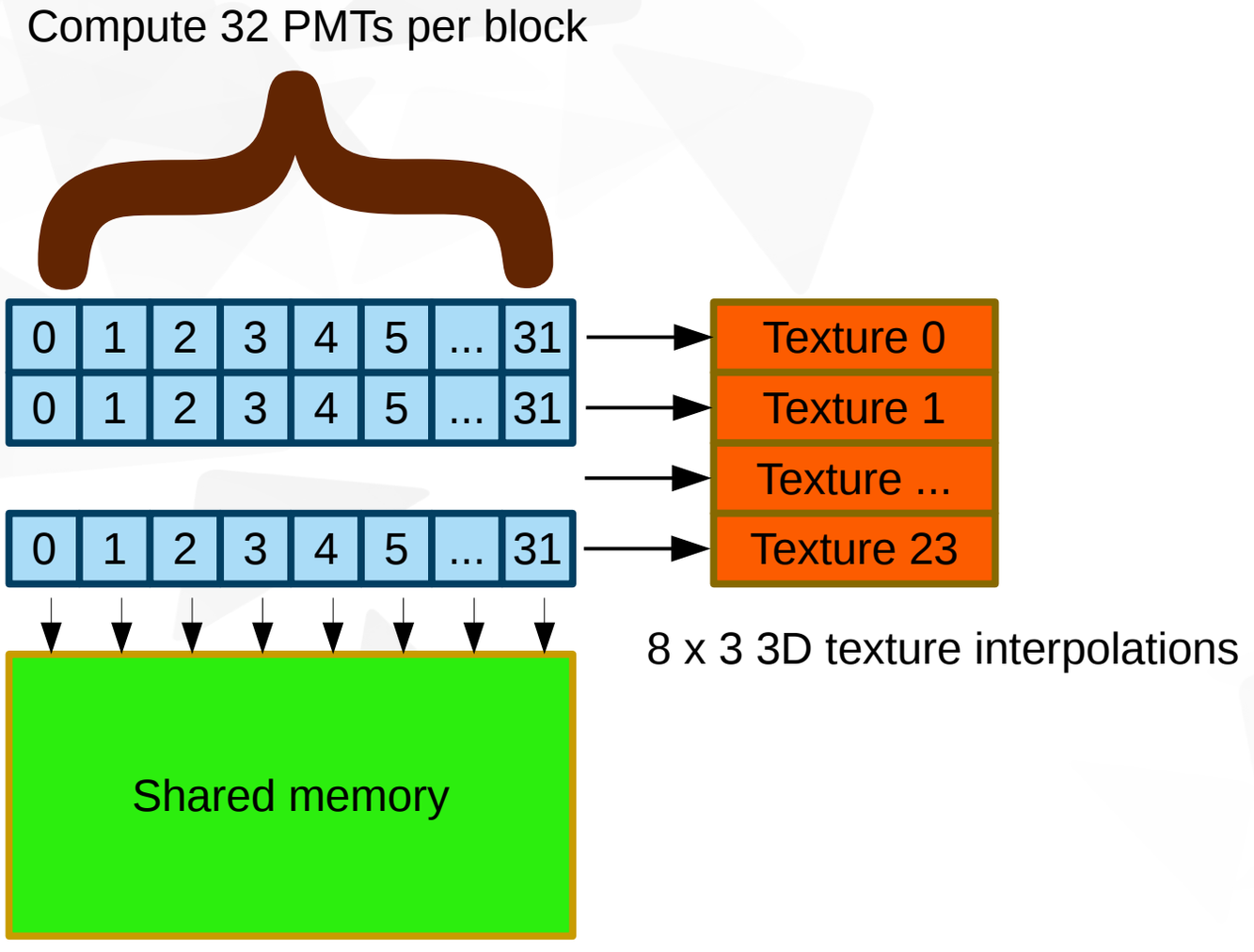


6D interpolation

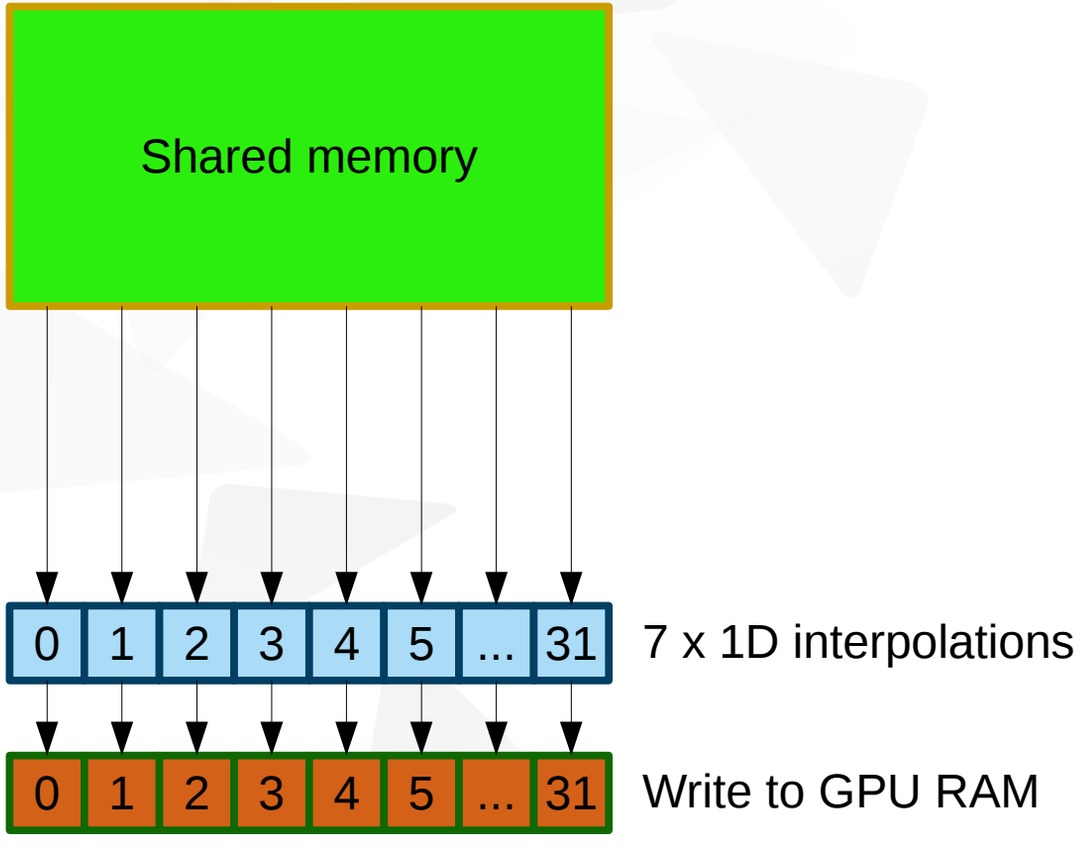
- The 6D interpolation can be performed using 8 3D textures, and combining the results using 7 1D interpolations
- The 3D interpolations are performed in hardware, and are fast
- The intermediate results are stored in shared memory cache, which is 100x faster than the VRAM



One block

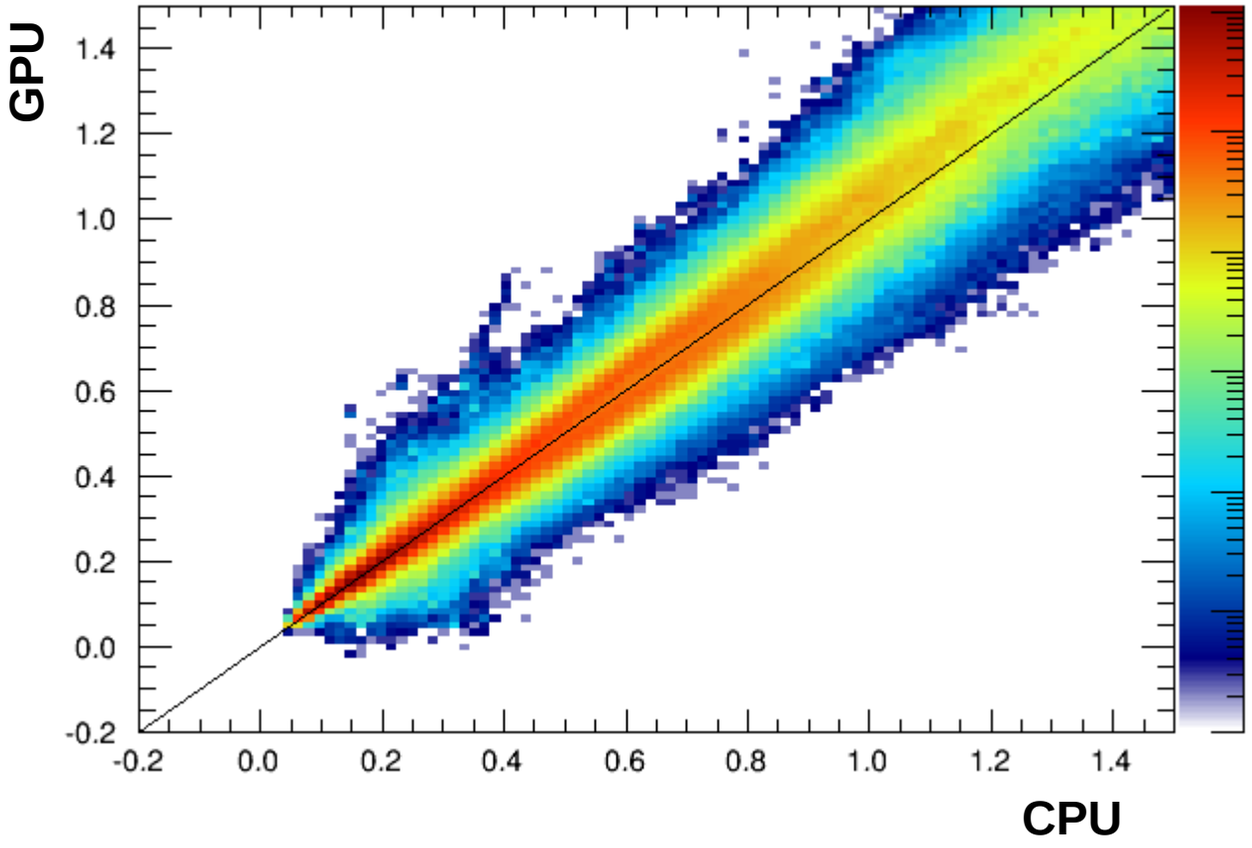


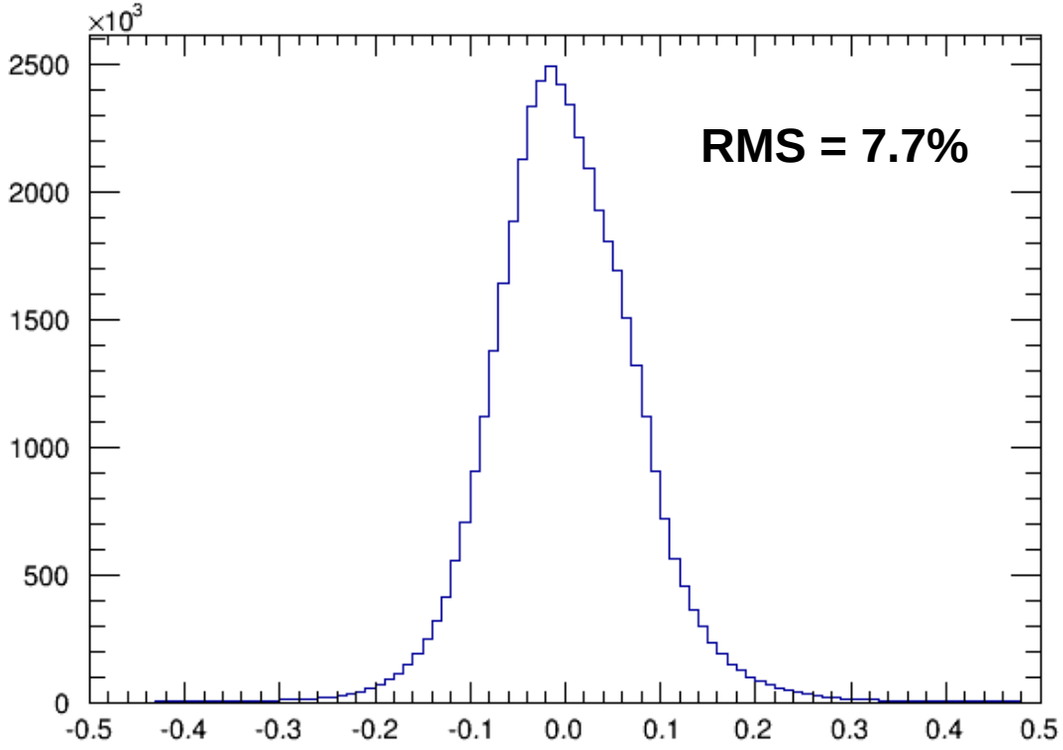
- A block is a unit of work that is performed on a multiprocessor
- Each block calculates 32 PMTs, performing 24 3D interpolations in parallel
  - 24*32 = 768 threads
- 24 / 3 = 8 textures per step
- The results of the 3D interpolations are stored in shared memory cache



- The intermediate results in shared memory are combined by 1D interpolations using a thread for each PMT
- This is wasteful! Already defined 768 threads for a block, so 732 threads are idle
- The results are written to VRAM
- Shared memory takes ~4 GPU clock cycles, VRAM is ~400 cycles!

Validation





More validations and tests will be shown later

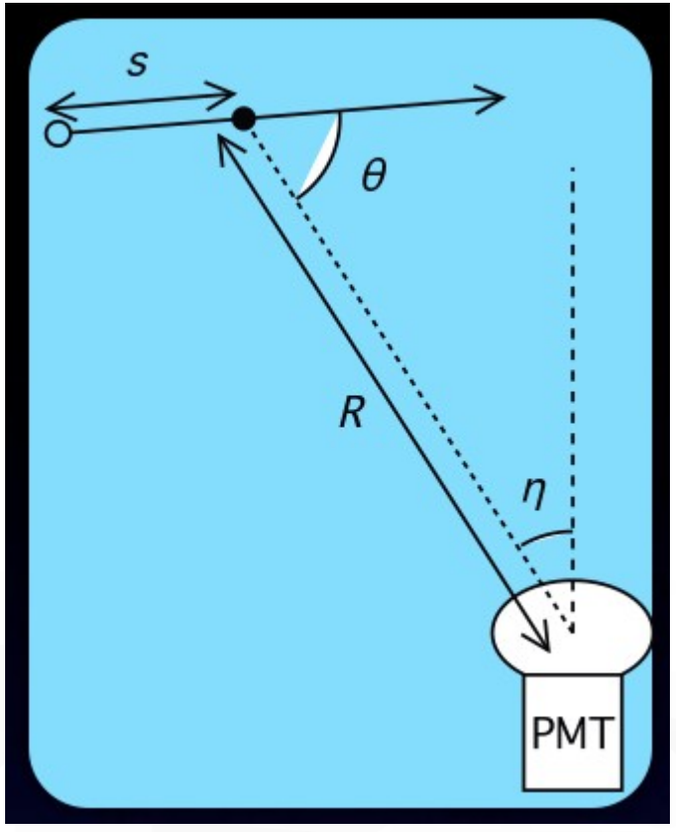
How does it perform?

- In a standalone benchmark, the GPU algorithm is ~3.2x faster at performing the interpolation for all PMTs
- This is far from the limit, profiling the code shows that it is being limited by “execution dependency stalls”, i.e. the threads are waiting for resources to become available
- This is fixable, however I haven't got round to it yet…
- The algorithm has plenty of room for improvement e.g. the idle threads when doing the 1D linear interpolations
- The difference between the CPU and GPU versions is larger than expected, but as we will see later, this doesn't seem to have a big impact

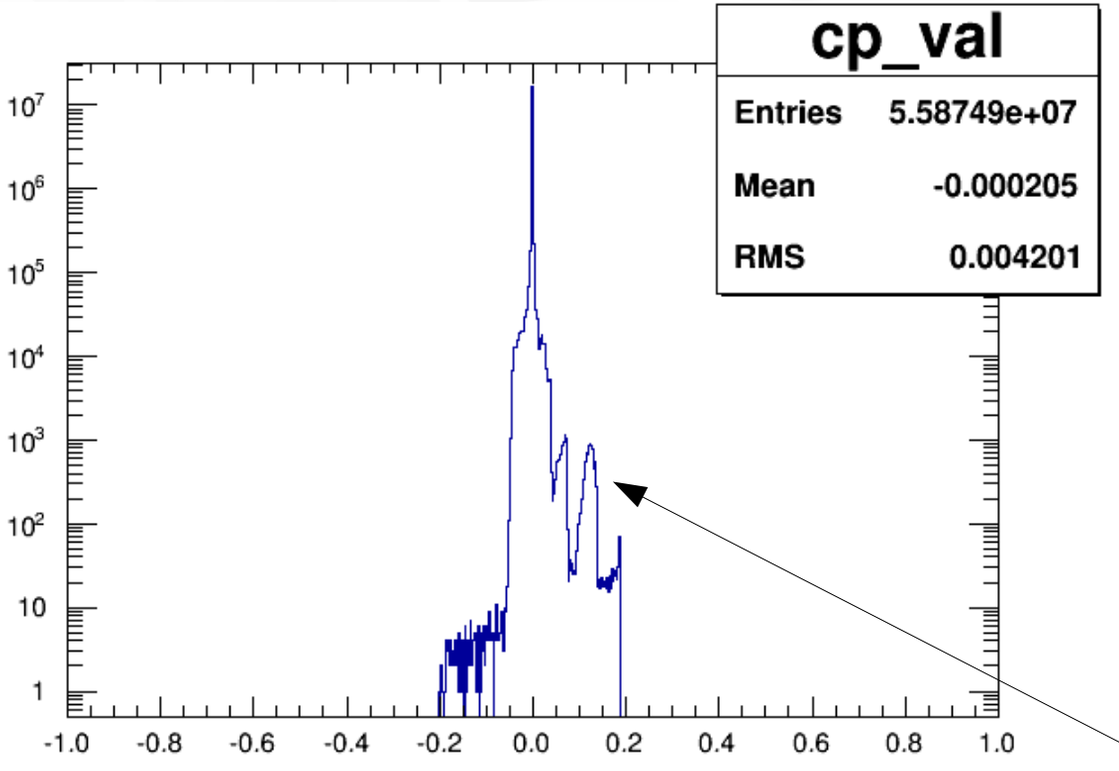
Cherenkov profile

Cherenkov profile

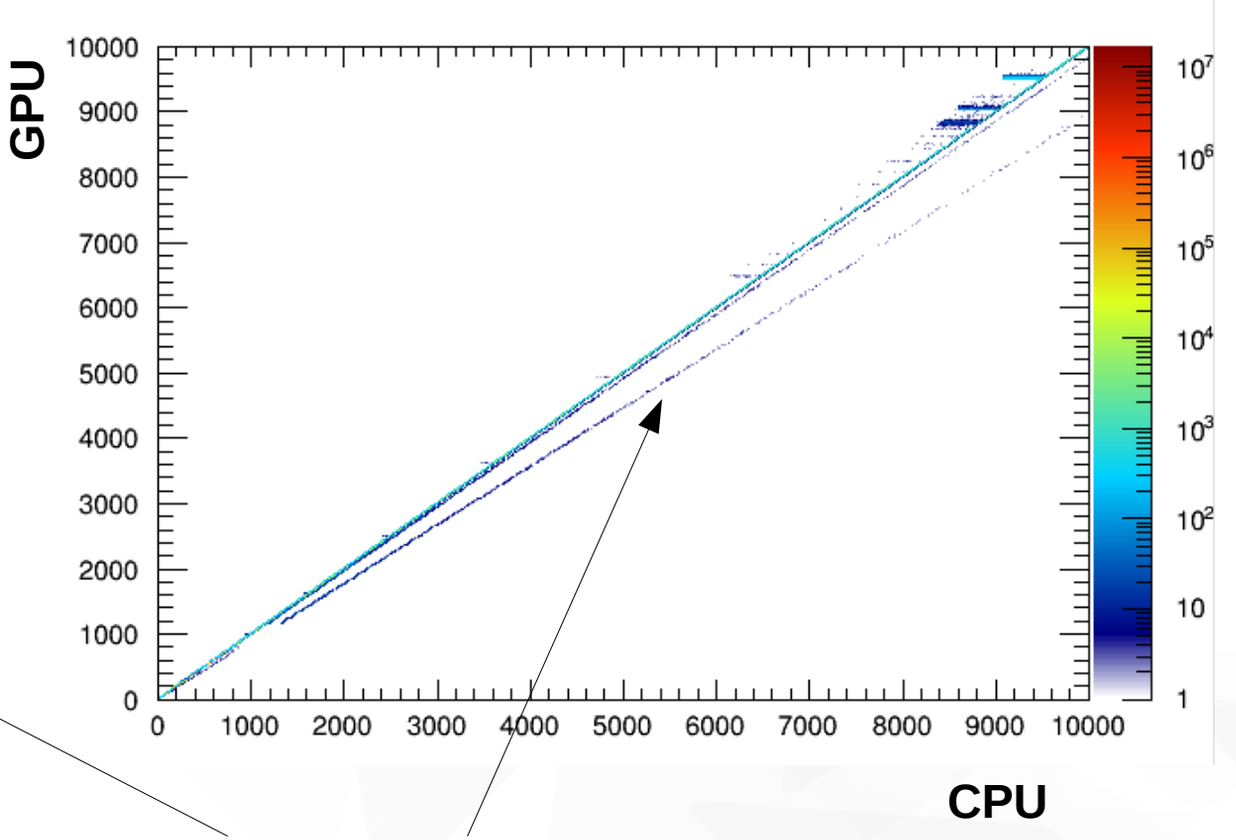
- Current scheme is organized into bins of **costh** and **R**, with each bin containing what are essentially 5th order splines to map the momentum space
- Momentum polynomials are evaluated in 4 neighbouring bins, and the results are linearly interpolated
- GPU attempt: Simply sample the distribution described above, into a (400,200,200) 3D grid
- 3D linear interpolation using GPU texture memory



Cherenkov profile validation



RMS < 0.5%



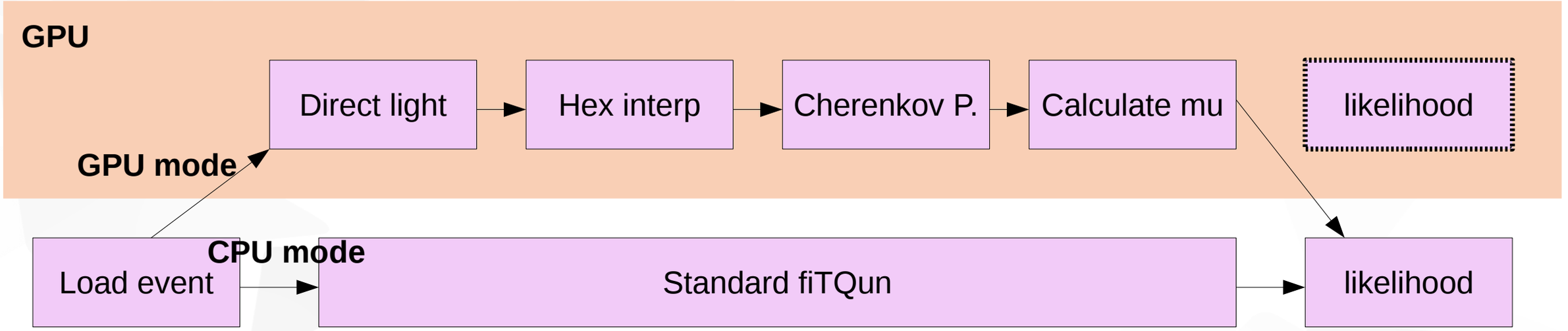
Strange features, though negligible.

Comments

- The GPU cherenkov profile calculation is ~**20x** faster than the CPU counterpart
- The error is less than 1%
- Would consider this a speed improvement with no detrimental effect to the result
- There are some strange features (previous slide), possibly related to the “bin finding”
  - These effects are negligible
  - Nevertheless, I will investigate their cause

Program flow

Flow chart



Still missing from GPU:
Time likelihood calculation (small overhead)
Negative track subtraction (uses the same code)
Likelihood (ties everything together)

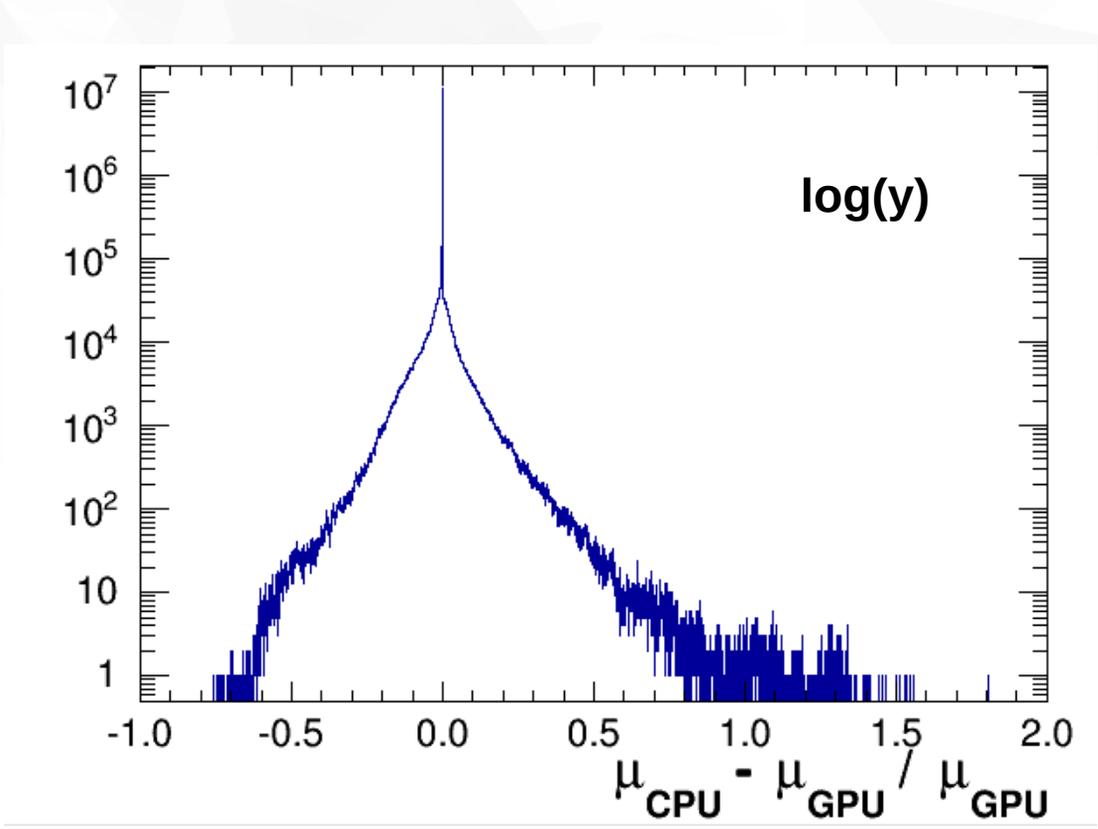
Copying back **“mu”** and **“muscat”** means copying 2*11146 elements, when likelihood is calculated on GPU, only need to copy 1 element

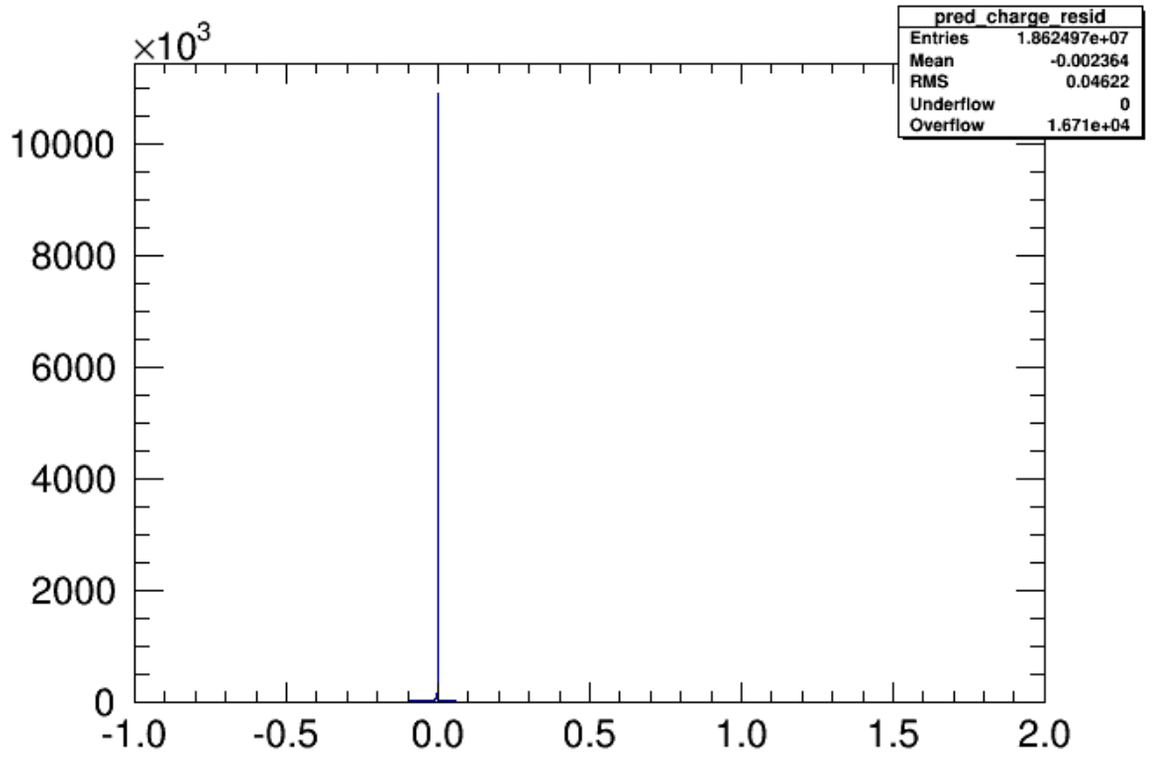
“boilerplate” code

- There are other computations done on GPU, specifically the J() function which computes the *solid angle*, *water attenuation* and *angular response*
- These don't contribute significantly to runtime, however performing them on GPU allows the code to compute the predicted charge, rather than copy the results of each “heavyweight” algorithm back to the CPU

Validation

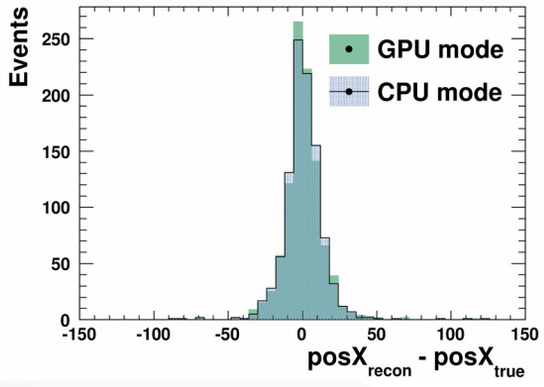
Comparison of predicted charge per PMT

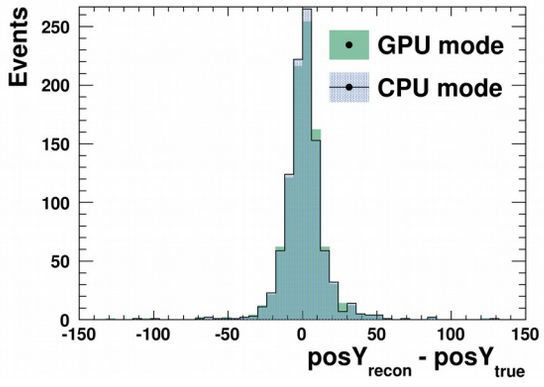


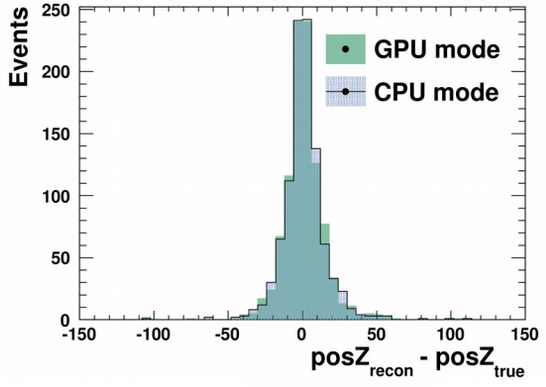


Predicted charge for every PMT compared for CPU and GPU algorithms, plotted as a fraction.
This was generated by running a single ring fit and comparing each PMT at every iteration. **RMS is 4.6%**

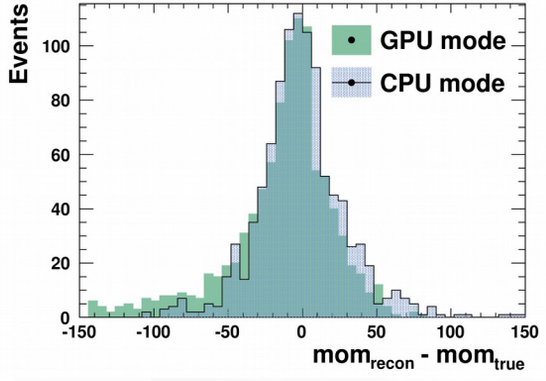
1000 Electron Particle Gun



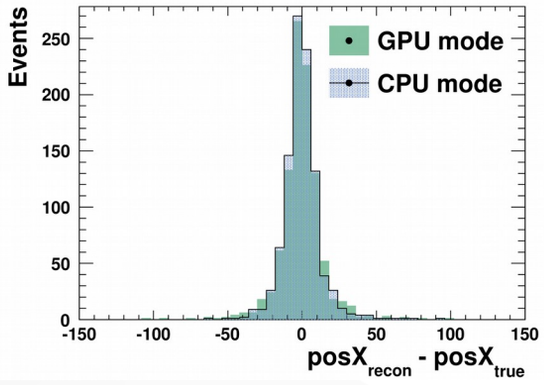


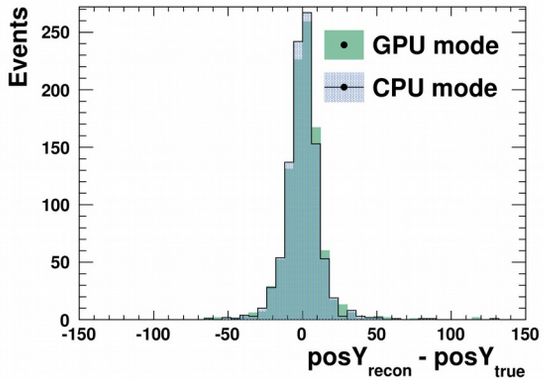


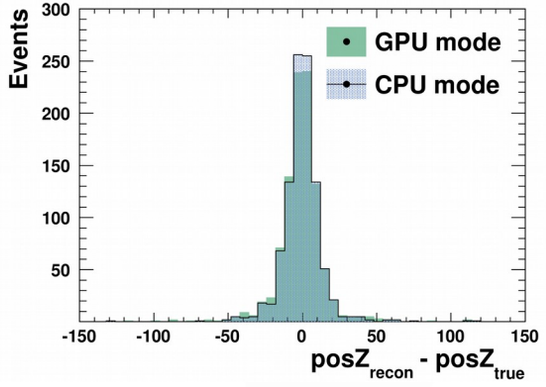
- Evaluate effect on fit result
- Fit 1000 electron particle gun events in CPU and GPU mode
- Results show there is negligible effect on the fitting power, with statistically different distributions



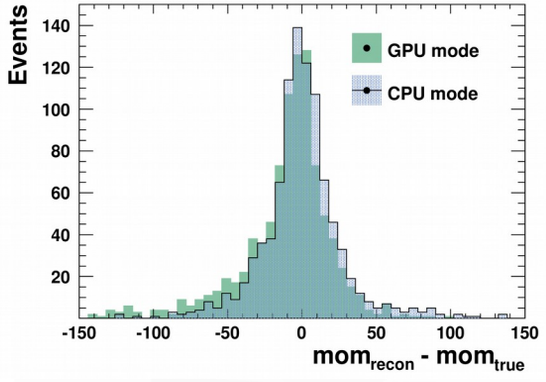
1000 Muon Particle Gun (no decay)







- Same as previous slide, but with muon particle gun. Decay electrons were turned off



Benchmarks

Execution time comparison

This is creating the predicted charge “PDF”, not the total time per step

	6D lerp	Cherenkov profile	J calc	Time per single track	Speed up
CPU	3	0.42	N/A	7	1
GPU	0.88	0.02	0.03	1	7

Time in milliseconds

Discussion on timing

- There remains some outstanding things to remove from the CPU code
  - A sqrt() per PMT is still performed, to feed to the time likelihood
  - Moving the time likelihood to GPU will shave off some 1/10th of a ms
- Not copying back the scattered light will reduce this further
  - Rather calculating the likelihood on GPU and passing a single floating point value back
  - Likelihood also contains log() and exp() functions, will further reduce runtime
- These factors I expect will push the time beyond 10x faster

Conclusions

- FiTQun now runs almost 10x faster with negligible effect on the fit result
  - Caveat: upstream track-subtraction is not implemented, however it uses all the same tools, so it is a matter of implementing it
- Next step, push to 50-100x speed-up
- This is feasible, but need to get time per single track to ~0.1 ms
  - All algorithms are within this time budget, besides the scattering table, which is 0.88ms
  - Significant work is needed here, but there is also many optimization opportunities

How do I use it?

- In my forked version of fiTQun, I have committed all the code
- GPU code is abstracted into an overloaded *fiTQun* class
- Compile GPU code using a Makefile flag
- Run GPU code by swapping
  - **fiTQun *thefit = new fiTQun()** with
  - **fiTQunGPU *thefit = new fiTQunGPU()**
- Enjoy ~10x speed improvement!

Backup

I want a GPU

- NVIDIA GPUs only (sorry for the vendor-locking)
- GPU memory footprint is 1.2Gb, so almost any NVIDIA card will run
- Required *compute capability* is 3.5 (kepler architecture)
  - This is due to the bindless texture feature used
- Recommended:
  - GTX 960 (CAD$ 260) with 2Gb RAM
- I want to spend a lot of money!:
  - GTX Titan X or Tesla production card