UNIVERSITY OF BIRMINGHAM

Garfield++ Parallelisation using GPUs

RD51 Mini-Week, 18th February, 2021

Mark Slater, Konstantinos Nikolopoulos, Birmingham University

Overview

This talk will cover our work towards enabling GPU support in the Garfield codebase:

- **Motivation**
- **Brief Overview of GPU Architecture**
- **Performing the Conversion**
- **Consistency Checks**
- **Efficiency Gains**
- **Future Plans**

Disclaimer 1: We've approached this from the point of view of the code, NOT the physics. We've kept the code the same as the original where possible. We aspire to enhance Garfield++ capabilities for the benefit of the community.

Disclaimer 2: Our first aim is to enable GPU use in Garfield++ and show consistency with the known codebase. Subsequently, we will target specifically at further optimising performance.

Motivation

Garfield is one of the ‘industry standards’ in the Gaseous Detectors field

It has a number of applications including Ionisation generation, Electric Fields and Electron Transport and Avalanching. For this case study we are using Gas Electron Multiplier (GEM) detectors

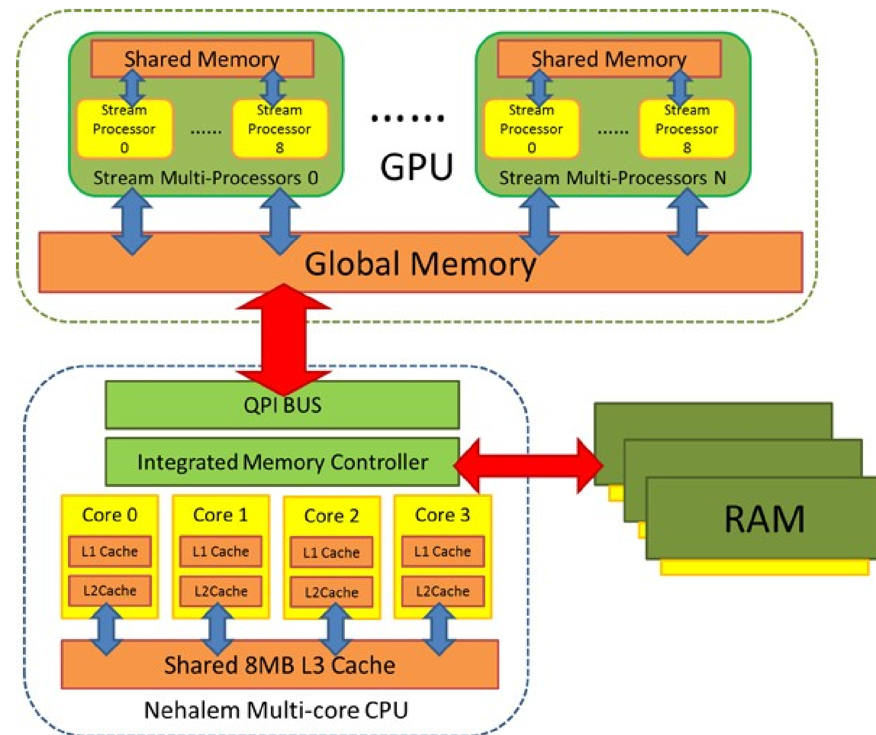
The main issue is that the generation and transport of large events within Garfield can take a long time (minutes)

In the past few months we have been looking at improving this time using paralellisation, specifically with GPUs

Basics on GPUs

Graphics Processing Units (GPUs) are ubiquitous in home computing and power the vast majority of **Machine Learning and AI applications**

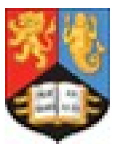
The general concept of them is **providing a vast number of cores** that individually are slower than the CPU but through, parallelisation provide **large efficiency gains**



For an application to take advantage of the gains offered by GPUs, it must:

- Just contain calculations
- Have paralellisable workflow
- Minimise data movement

Though the Garfield code base isn't ideal for a GPU, the **transport of large numbers of electrons has the potential to be done in parallel**



UNIVERSITY OF BIRMINGHAM

Overview of Garfield 'Step'

Input initial Electron pos

Check pos is Valid and find Electric Fields

Add random time Steps until collision

x1-100

x1-x100

Process collision

Store updated electron info

Repeat for each electron in stack

Remove dead Electrons and Add new ones

UNIVERSITY OF BIRMINGHAM

Considerations For GPU Code

CUDA code must be compiled with the NVIDIA compiler

CUDA/device code must be compiled using the the NVIDIA compiler. This made things difficult when trying to extend general C++ classes.

Dynamic Allocation on the GPU isn't trivial

The typical STL classes are not available to the GPU and would be expensive to implement. This means extensive use of C-Style arrays.

Minimise memory copies

Ideally, all computation should be done in the GPU memory, not by transferring things back and forth to the CPU

Minimise Branching Code

'if' statements can be very costly in GPU coding so these have to be reduced if possible. This meant avoiding checks and only including things that were necessary

Geometry must be static

The code obviously needs to be thread-safe so the the geometry and associated setup cannot be changed during the processing

Hardware/Software Used

Development was done on a basic server with:

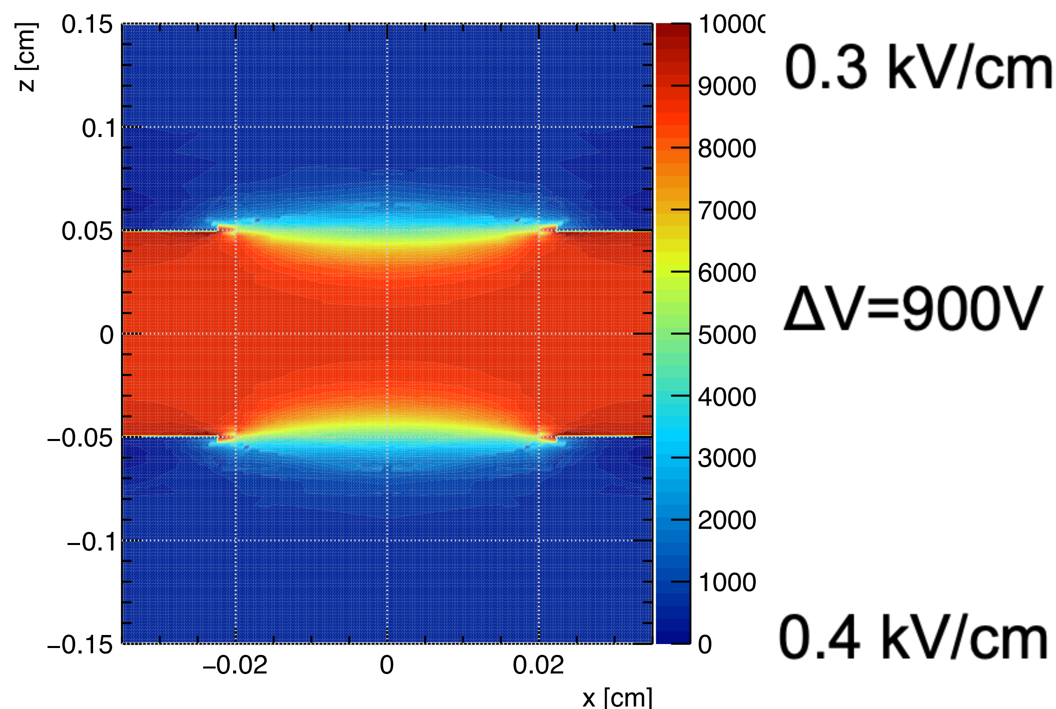
- Dual E5-2620 v4 (16 core total)
- 64GB RAM

This has two Tesla P100 cards (only 1 used at a time in study):

- Released Jun 20th, 2016
- Pascal architecture
- 3584 cores each
- 16GB memory
- Memory bandwidth: 732.2 GB/s
- Base Clock: 1190 MHz

CUDA 11.0 was installed with NVIDIA driver 450.51.05. The base system was Fedora but using a chroot to a CentOS 7 image

GEM Model Used



For this study we used a standard THGEM:

- **1mm thick FR4 coated on both sides with a 17 µm layer of copper**
- **hexagonal pattern of cylindrical holes**
- **∅ = 400 µm**
- **pitch: 700 µm**

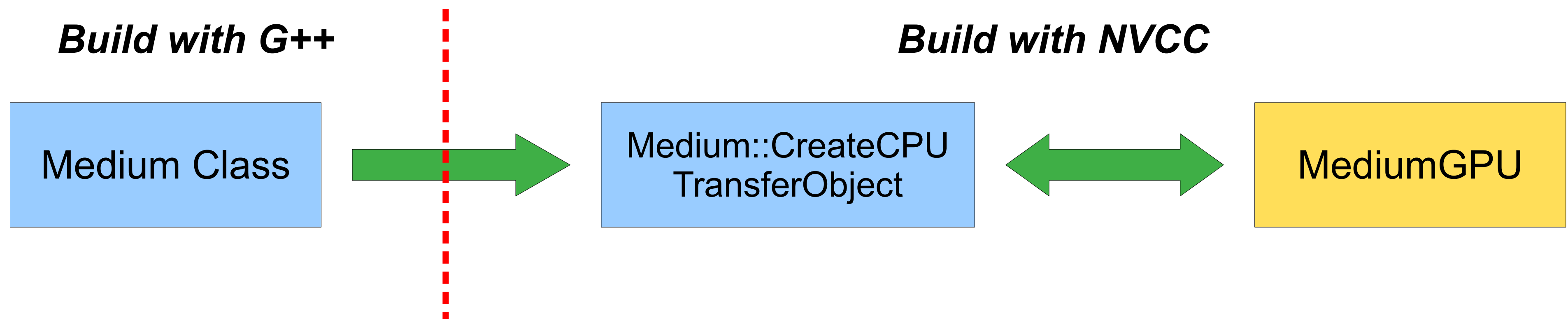
Using Ar:CF4 (80%:20%) at various pressures to have different avalanche sizes.

Converting - Initial Setup

The geometry and setup in Garfield is stored in several classes:

- Sensor
- MediumMagBoltz
- ComponentFieldMap

These couldn't just be run on the GPU due to extensive use of std::vector so copies of the classes were developed that were then filled from the originals



We ensured we did as much pre-processing on the CPU first before transferring all data to the GPU

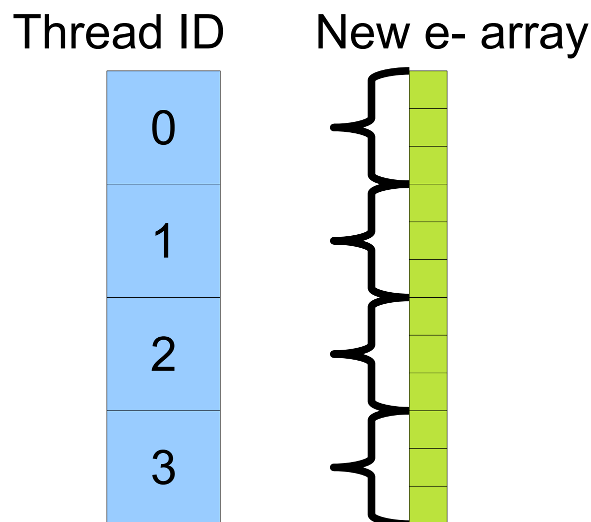
Converting - Main Loops

The main electron processing loop would be **performed in parallel across the GPU** with one **thread transporting one electron**

The code was used as is with the only changes being that to deal with:

- Shifting to c-style arrays
- Updating the info of an electron
- Storing any newly created particles

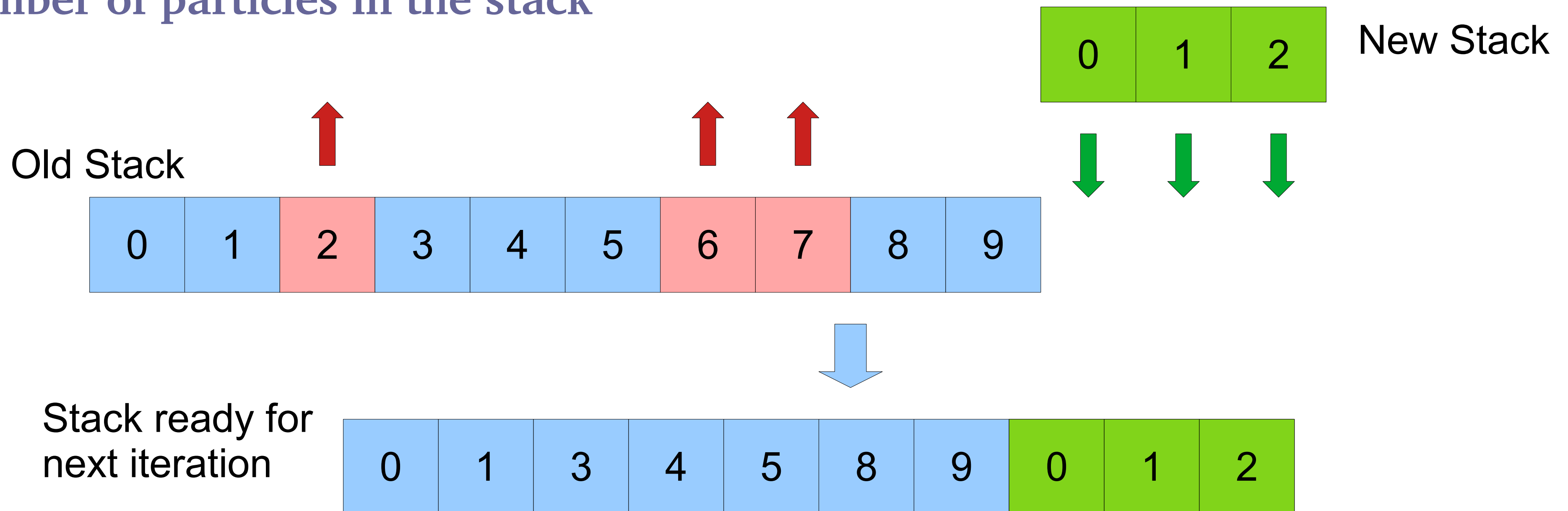
Due to not having thread-safe, dynamically allocated arrays, we created a large array that had enough space for each thread to store the particle info:



Converting - Stack Processing

At the end of each processing 'step' (i.e. each electron going through the main loop), the stack of electrons is processed. This involves **removing terminated electrons and adding in newly generated ones**

By necessity, this will involve 'random' memory access proportional to the number of particles in the stack



This was found to be a very significant cause of slow down for the GPU and much more efficient ways are possible

Consistency Checks - setup

The first goal was to make sure the GPU code **produced the same results** as the CPU code

To do this, both versions of the code were **run in parallel** – each step/iteration was processed by the GPU and CPU independently

We could then **compare electron numbers and positions** after each step as well as after the full generation

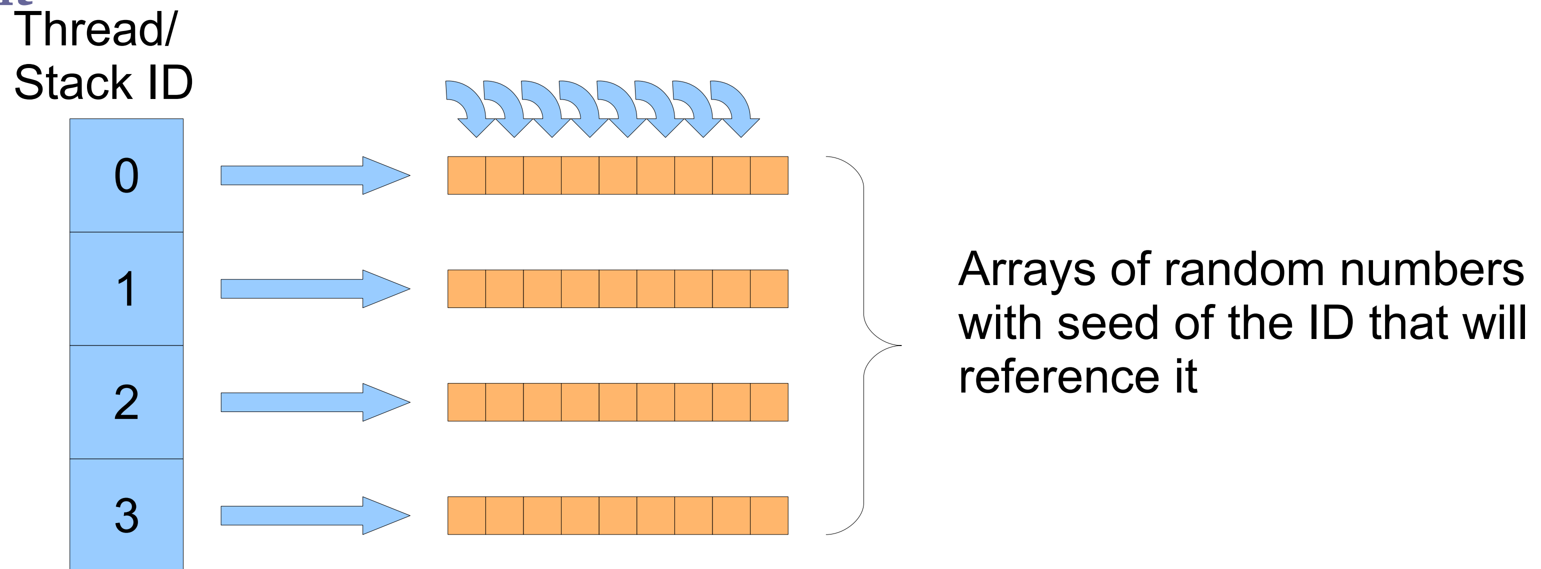
Particular cases that **generated large showers** were used as the test cases. The starting position and seeds were fixed to ensure **the same shower was created each time**

University of Birmingham

Consistency Checks - RNG

The biggest problem with comparing CPU and GPU code was **making sure the RNG was the same for both**. By default, Garfield uses ROOT for it's RNG source which wouldn't work on the GPU

To get around this, we **pre-generated a large number of random numbers** with seeds equal to the electron position in the stack which also corresponded to the thread that tracked it



This accounted for 14.5GB of GPU memory but would not be needed in normal running as CUDA provides RNG generation itself

Consistency Checks - Results

We get very good agreement in the stack sizes and end points for the events we looked at

For this event, there was a maximum difference of 247(~1%) in stack size during transport

After transport, the difference in the end points was 48 (0.02%)

UNIVERSITY OF BIRMINGHAM

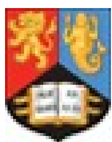
Inconsistencies

After doing a lot of tracking of individual particles and positions, the GPU code was found to **match the CPU code very closely**

However, investigations showed **occasional small differences at the level of 10^{-6}** that over subsequent iterations, sometimes **created larger differences** in the results due to the exponential nature of the avalanche process

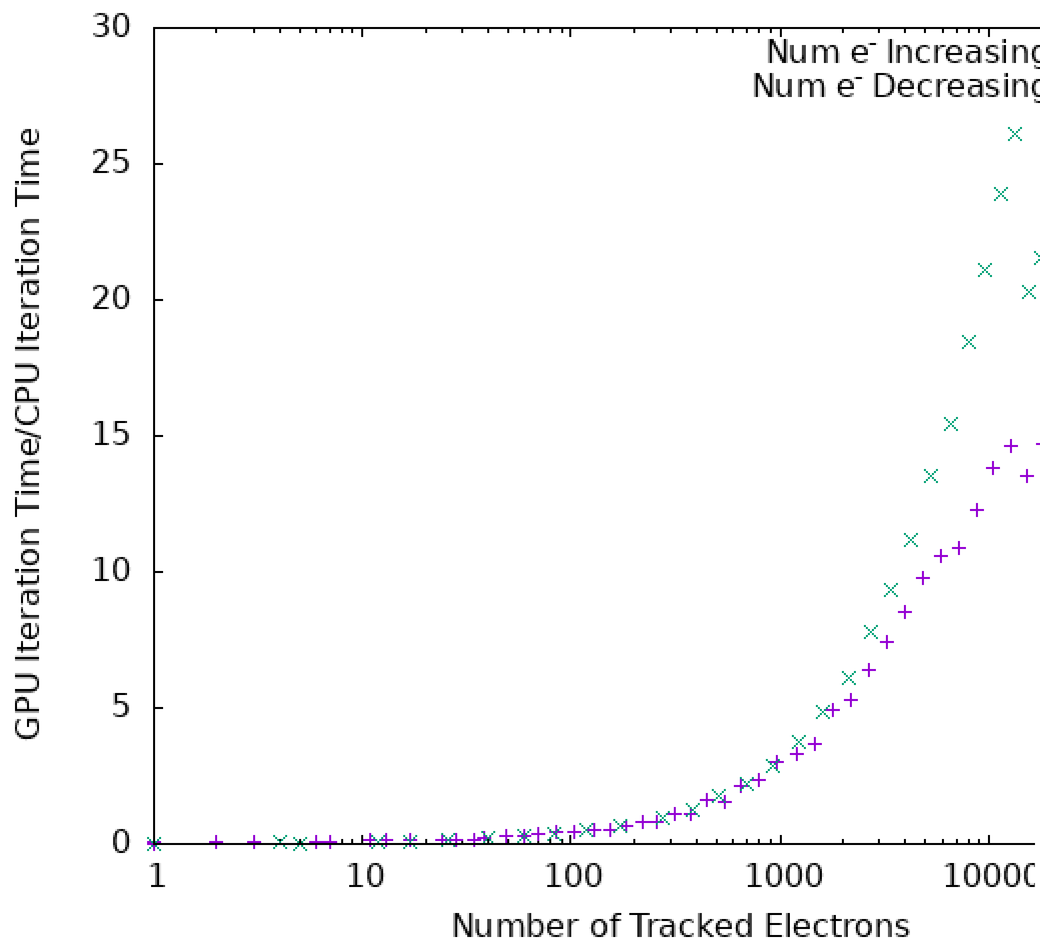
Tracing the source of these showed there were **unavoidable differences at the 10^{-8} level** due to the different architectures and compilers. These errors were compounded due to the many hundreds of FP calcs being done in sequence

To try to remove this, we added the option to **round off to a number of decimal places** at various points in the code. This showed significant improvement but also indicated some very small differences were irreducible



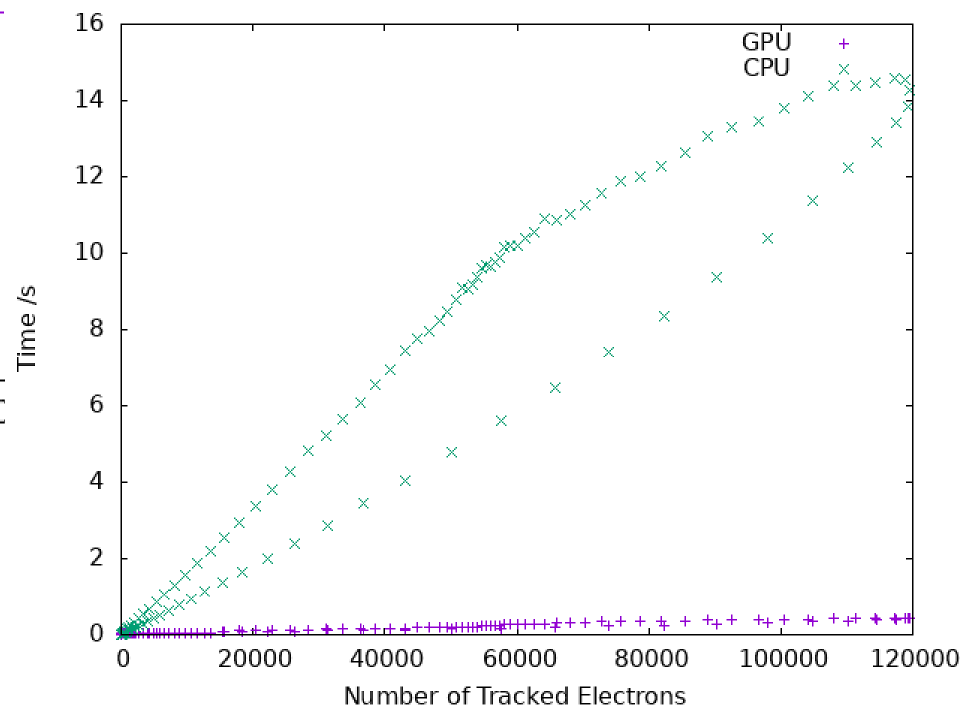
UNIVERSITY OF BIRMINGHAM

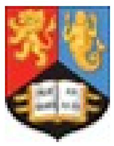
Efficiency Improvements



On average, there is a x20 speed up using the GPU even without optimisations

Both CPU and GPU take longer to transport when electrons are decreasing rather than increasing – more investigation is needed to find the cause





UNIVERSITY OF BIRMINGHAM

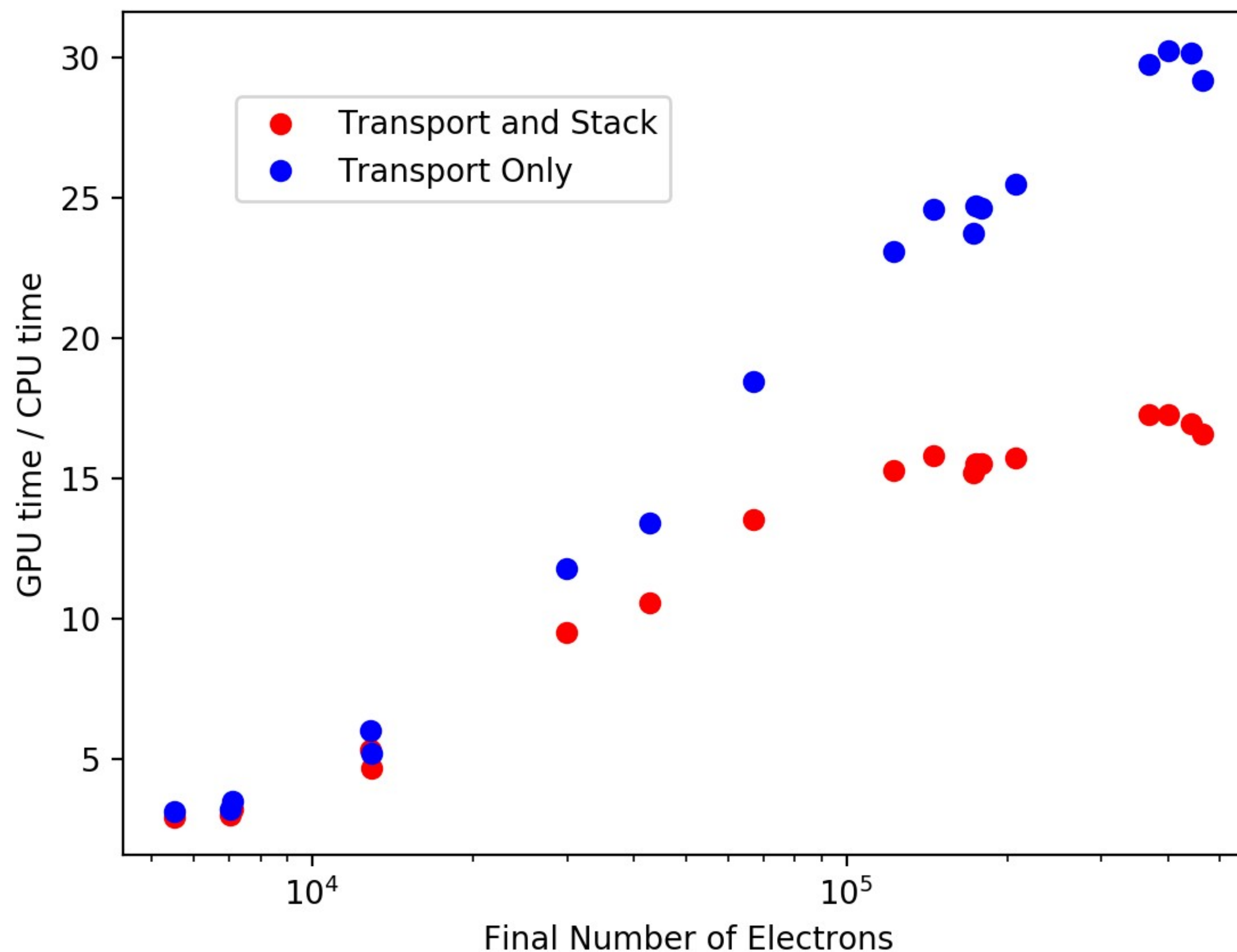
Efficiency Improvements

UNIVERSITY OF BIRMINGHAM

Efficiency Improvements

Studies were also done over a number of other large avalanche events

NOTE: this plot shows the GPU/CPU ratio against the Number of End Points which will be significantly larger than the per iteration stack size



Future improvements

Increase GPU efficiency

Now we have showed consistency between the GPU and CPU code we plan to increase the efficiency as much as possible, e.g. by removing conditionals and debug info, copying for consistency checks and switching to floats.

Switch between GPU/CPU use automatically

As you only get improvement for sufficiently large showers, the code should be able to switch to GPU use when it would give benefit

Allow multiple GPU use

There's no technical reason why multiple GPUs can't be used simultaneously for bigger efficiency gains

UNIVERSITY OF BIRMINGHAM

Future improvements

Allow processing of multiple showers at once

By tagging electrons with an 'event ID' you could run many showers simultaneously on the GPU using the same geometry/setup

Automatic Code Generation

Many of the steps in converting the code (i.e. create class, shift to c-style arrays, transfer data) could be automated as a separate build step ensuring there aren't separate GPU and CPU versions of the code

Expansion to generic multi-threading

The GPU version of the code is, by necessity, thread-safe. It should therefore be relatively easy to switch to multi-threaded generation which would allow the utilisation of multi-core machines

Summary

We have managed to get the **Electron Transport parts of Garfield++ running on a GPU**

The GPU and CPU transport has been found to be **almost identical on smaller electron populations** and **consistent on larger populations** to within the expected error due to different architectures, compilers, etc.

A significant **speed up of up to 20x** has already been found with many possible efficiency improvements left to be applied

This work could also fairly easily lead to **providing full multi-threaded support in Garfield++**