UNIVERSITY OF BIRMINGHAM

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement no 101026519 (GaGARin).

Garfield++ Parallelisation using GPUs

RD51 Meeting, 19th June, 2023

Mark Slater, **Tom Neep**, Konstantinos Nikolopoulos, Birmingham University

UNIVERSITY OF BIRMINGHAM

Overview

This talk provides an update to the our work on investigating the use of GPUs within Garfield++. It will cover:

- Motivation and Setup
- Consistency Checks
- Code Changes
- Performance Gains
- Details on Code Conversion
- Future Plans

Disclaimer: As before, we've approached this from the point of view of the code, NOT the physics, i.e. the GPU code is running the same algorithms (and where possible, the same code) as the non-modified CPU version

Motivation

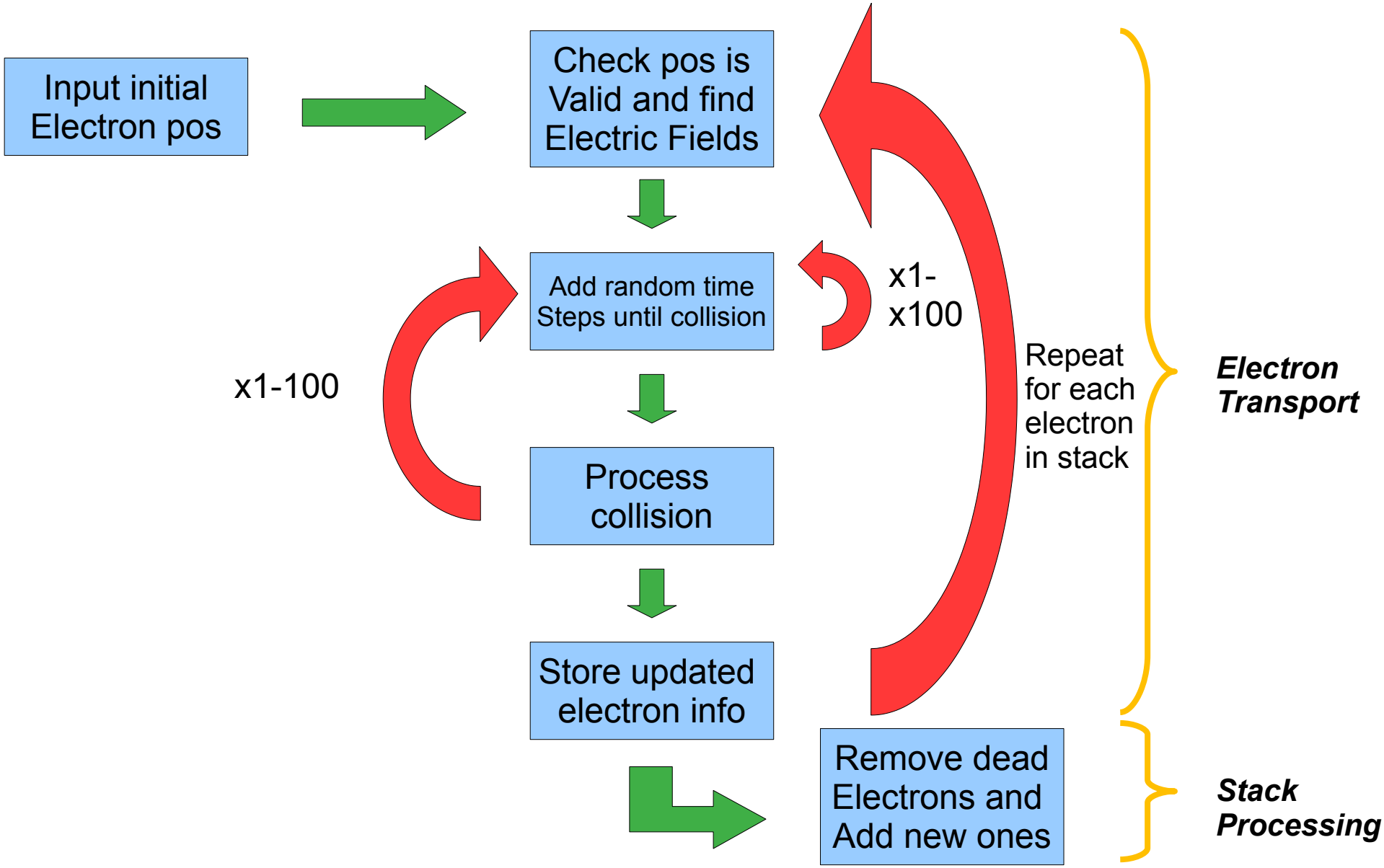
Garfield is one of the 'industry standards' in the Gaseous Detectors field

It has a number of applications including Ionisation generation, Electric Fields and Electron Transport and avalanching. For this case study we have focused on **Gas Electron Multiplier (GEM) detectors** using the **AvalancheMicroscopic** class

In this work, it was found that that the generation and transport of typical events within Garfield **can take a long time (5-10 minutes per event) where there are several hundred thousand electrons in the avalanche**

As the transport of each electron in the avalanche is independent, we have been exploring the benefits of using parallelisation, **specifically porting the code to run on GPUs**

An AvalancheMicroscopic 'Iteration'



UNIVERSITY OF BIRMINGHAM

Hardware/Software Used

Development was done on a basic server with:

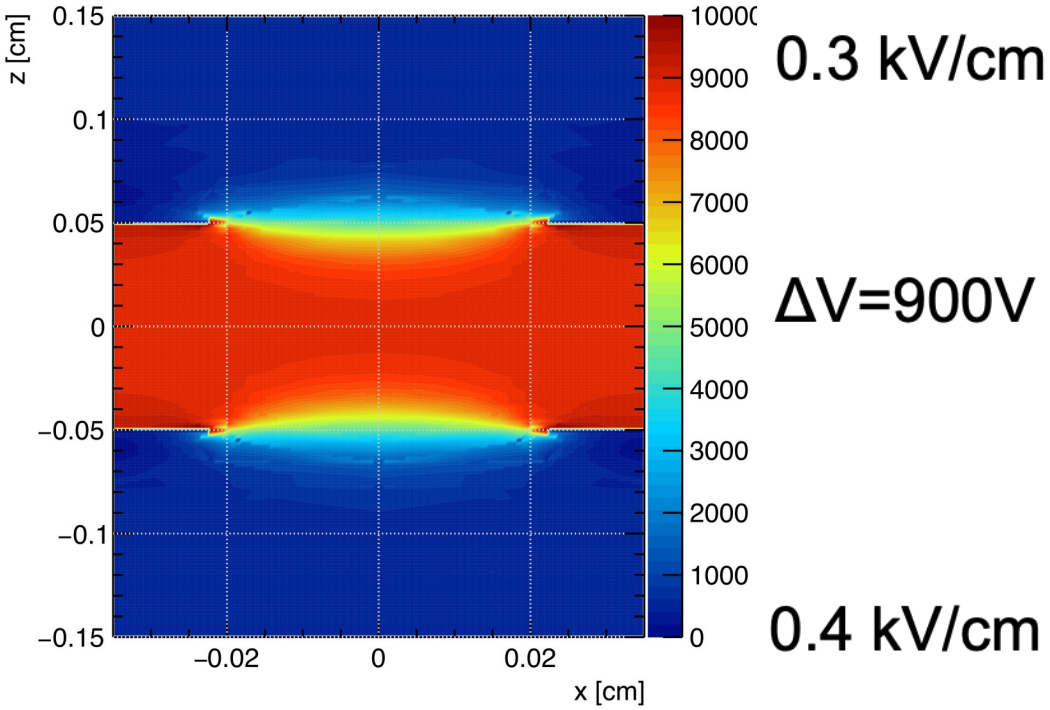
- Dual E5-2620 v4 (16 core total)
- 64GB RAM

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

- Pascal architecture
- 3584 cores each
- 16GB memory
- Memory bandwidth: 732.2 GB/s
- Base Clock: 1190 MHz

CUDA 11.5 was installed with NVIDIA driver 495.29.05 running CentOS 7

GEM Model Used



For this study we used a standard THGEM:

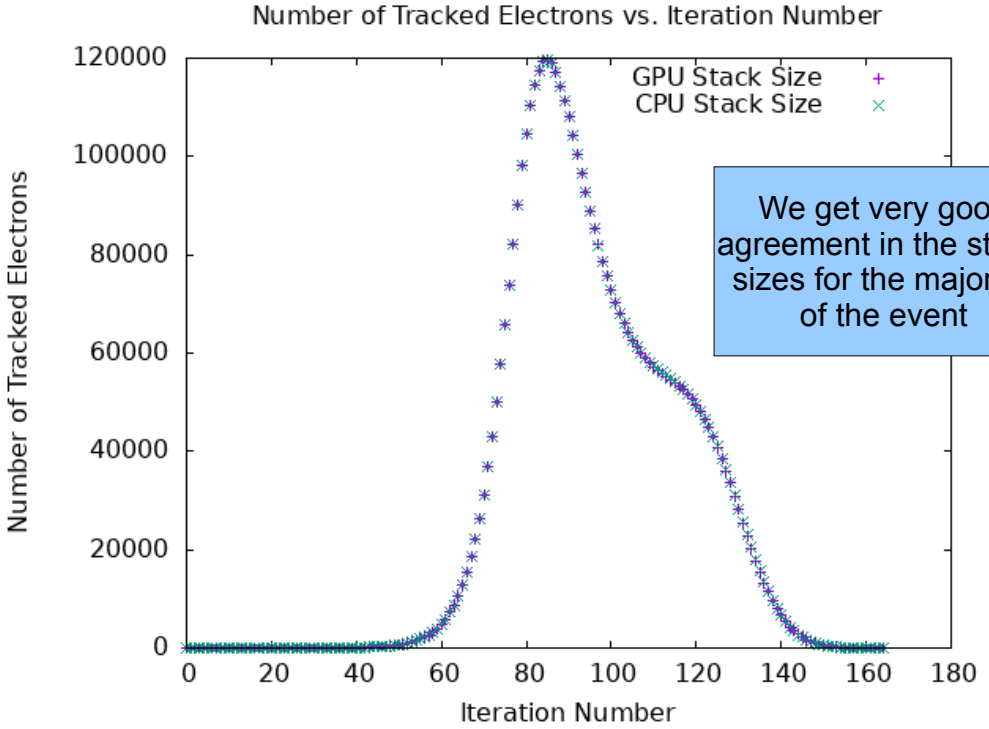
- 1mm thick FR4 coated on both sides with a 17 µm layer of copper
- hexagonal pattern of cylindrical holes
- ∅ = 400 µm
- pitch: 700 µm

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

Consistency Checks - Results

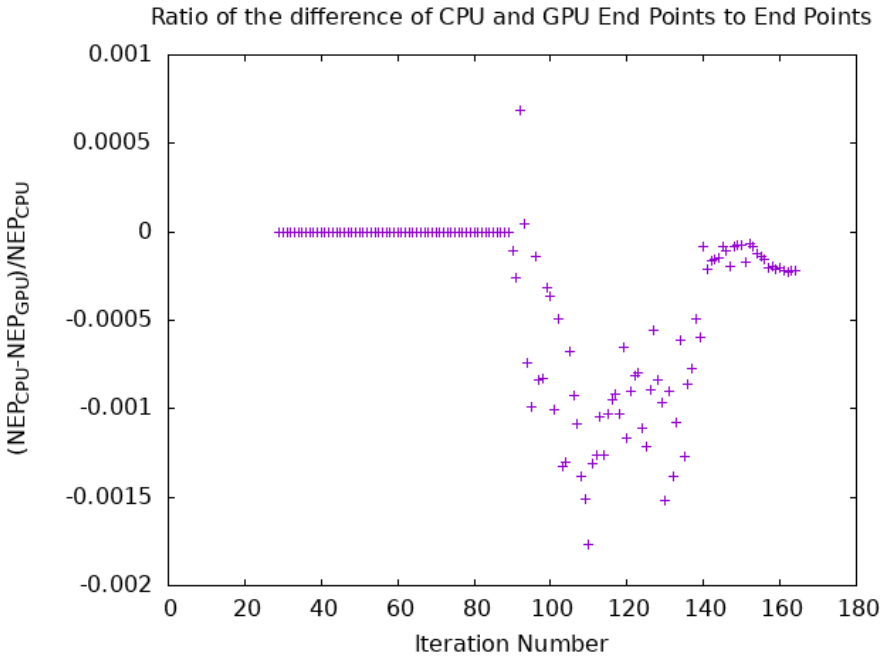
The initial work was designed to be a **feasibility study** to make sure this was worth pursuing, i.e. results were **consistent** and **significant performance benefits** were seen

As has already been reported, running the **same avalanche generation code** through the GPU led to almost **exactly the same numerical results** with discrepancies due to unavoidable floating point differences at the level of 10^{-8}



We get very good agreement in the stack sizes for the majority of the event

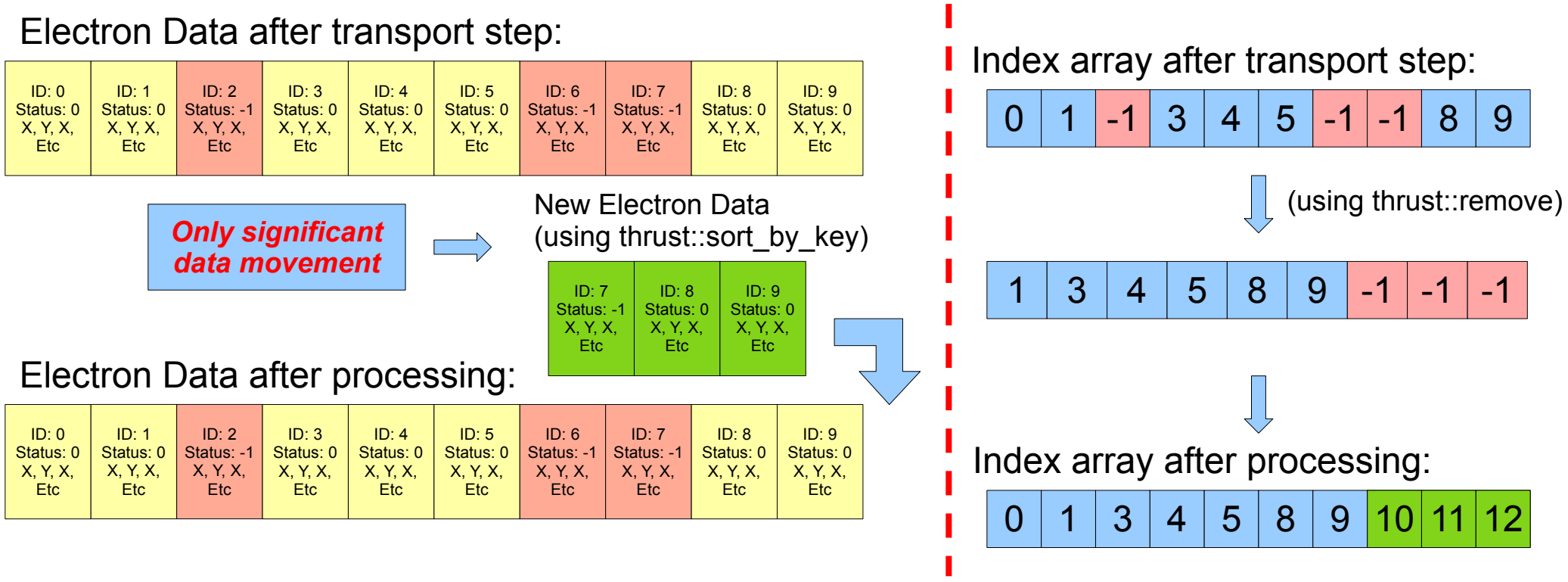
The number of end points at each iteration is in very good agreement with the difference being <0.15%



Stack Processing

Originally, a **20x speed up was found** but a lot of time was spent **processing the electron stack**, i.e. removing terminated electrons and adding newly produced ones to the stack

This slowdown was due to the **data reordering** being done. To avoid this, an **index array was processed** instead and each thread used this as a lookup into the electron data stack



After this change, the GPU stack processing time was **found to be negligible (<2%)**

Random Engine

To do the consistency checks, we set up a new Random Engine that **pre-calculated a large stack (14.5GB)** of random numbers using the default RootEngine

These were then copied to the GPU and the avalanche code for **both CPU and GPU drew from these** based on Electron ID in the stack instead of calculating them on the fly

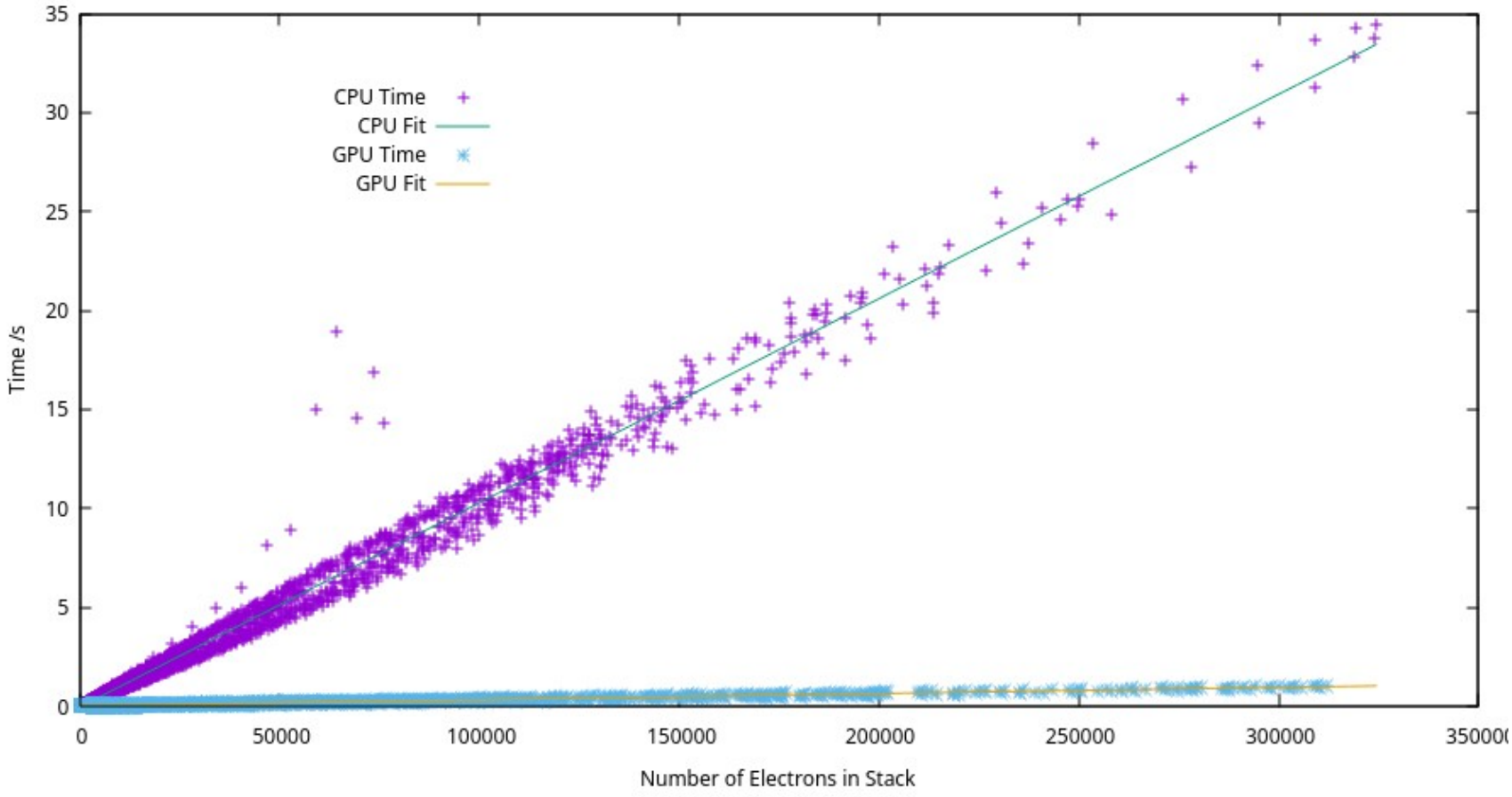
For performance comparison in a realistic environment however, we switched the CPU back to using the default RootEngine and coded a GPU RNG Engine that used the **cuRAND library supplied by CUDA**

This library can accept a **single seed** as expected but ensures that each thread of the GPU will use a **different sequence of random numbers**

This change resulted in a **speed up of ~50% in transport processing times** in both the CPU and GPU code. Therefore, the overall change in relative performance between GPU and CPU was **mostly negligible**

Relative GPU Performance

Generating **50 events** on both CPU (~164mins) and GPU (~8mins) and comparing the transport times for each iteration over all of these against number of particles shows an **average speed up of 30-35x in transport processing** on average

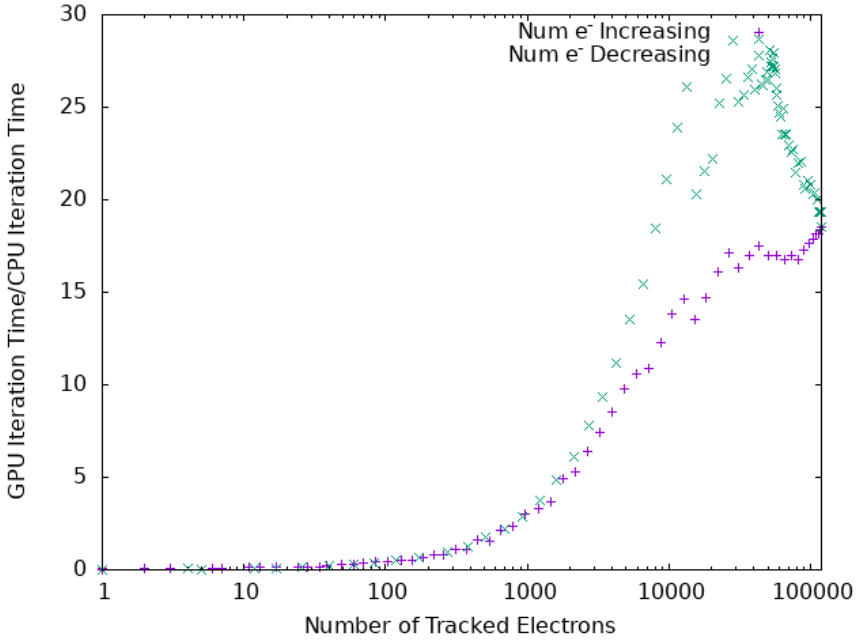


Variations in Performance

As was noted previously in the feasibility study, the **time taken for an iteration** seemed to be greater if the number of electrons was **increasing** (i.e. during the start of the avalanche) for both CPU and GPU

This was found to be due to electrons that were **drifting for the full 100 loops** in the iteration rather than creating a new particle and **breaking out of the loop** – a consequence of the drift region underneath the avalanche region in the GEM

Interestingly, these iterations also showed an **increase in relative GPU performance**, most likely due to increased occupancy of the GPU – there would be decreased divergence due to fewer breaks in the loops

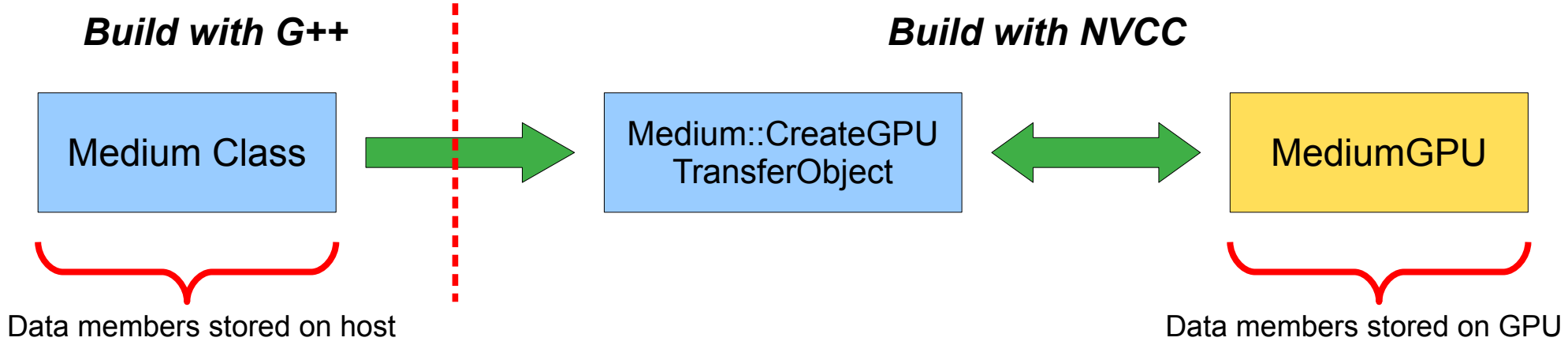


Converting Data for the GPU

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. Note the volume of data here is relatively small (~100MB)

Converting Code for the GPU

Inheritance trees in Sensor and Medium classes

In the original code, derived classes of parent abstract classes were used. At first, similar inheritance trees were created for the GPU classes but this was found to be 1.5x slower than the 'flat' structure due to the overhead of vtable lookups on the GPU

In order to still allow the required polymorphism, the 'CreateGPUTransferObject' function was overloaded to set an enumerated flag to the required concrete class

When calling virtual functions in the GPU class, this flag is then checked in a 'switch' statement to decide which actual code to call

Ensuring no Repeated code

As the majority of the code in these 'Geometry' classes was exactly the same for both CPU and GPU (though the members referred to were often different types), a form of cross compilation was used

By adding macros around the appropriate code, the same function code and declarations could be used for both GPU and CPU compilation

This prevented duplication of code in the codebase and kept the differences clear and together

Converting the Code

Sensor.hh

```
#ifdef GPUCOMPILE
#define __SENSORCLASS__ SensorGPU
#else
#define __SENSORCLASS__ Sensor
#endif

class __SENSORCLASS__
{
    ...
}
```

Defines the name depending on if you're compiling the GPU or CPU code

Sensor.cc

```
#include "Sensor.hh"

...

#ifdef GPUCOMPILE
...
#endif

...
```

Functions used by the GPU are surrounded by **#defines** as well as any slightly different code required

SensorGPU.h

```
#include "Sensor.hh"
```

Essentially just a stub file that includes the main header file

SensorGPU.cu

```
#define GPUCOMPILE

#include "SensorGPU.h"
#include "Sensor.cc"

Sensor::CreateGPUTransferObject()
{

}
```

Includes the ***c++ source file*** after setting the flag to only include the GPU parts

UNIVERSITY OF BIRMINGHAM

Debug and Profiling Options

During development of the GPU code, a number of profiling and debug options were added to check consistency and performance

These have been kept in the AvalancheMicroscopic class to aid with further development

They include:

SetShowProgress

Print out a brief summary for each iteration showing time taken and stack size

SetMaxNumShowerLoops

Stop the event simulation after a maximum number of iterations

SetDebugShowerIterationAndElectronID

Print debug info (positions and energies) for a particular electron ID in a particular iteration. Useful for checking consistency

PrintComparisonStats

Print out summaries of each iteration (stack size, processing times and transport times). Note this info is also directly available from the class after event generation

Generic Multi-threading

A significant effort was also made to use this work to allow **CPU multi-threading** in the AvalancheMicroscopic class

This was done at a low level by **starting a pre-determined number of threads** and **dividing the electrons** to process in the stack between them

It was assumed that the improvements would **scale roughly linearly** with the number of threads used (assuming num threads < num cores), however the **same performance as single thread was found**, irrespective of the number of threads running

After significant investigations into the reason, it was found to be due to the **default compiler optimisations** (-O3) that are used when compiling Garfield (It is a known issue/feature that compiler optimisations can **affect multi-threaded code**)

Turning these off showed the expected performance increase over single thread but that single thread performance was **drastically reduced**.

More investigations are needed to try **switching off optimisations** on particular parts of the code or maybe looking at using **OpenMP** in some areas of the code

UNIVERSITY OF BIRMINGHAM

Future improvements

Increase GPU efficiency

Profiling of the GPU code currently shows very low GPU occupancy (~14%). To try to improve this we plan to look at:

- Ways to avoid branching in the code
- Breaking the algorithm down into smaller elements
- Code simplifications to reduce register use

Switch between GPU/CPU use automatically

Due to overheads for setup/copying, you only get significant benefit from the GPU with larger avalanches. We will provide an option to switch the generation to the GPU at these points.

Allow multiple GPU use

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

Summary

We have got a **working GPU version of the AvalancheMicroscopic class** from Garfield that could be useful for those doing investigations into larger avalanches

Consistency between the GPU and CPU versions has been shown to **a very high level** and a performance increase of **x30-35 has been shown**

Differences between the GPU and CPU as well as factors affecting performance have been **investigated and understood** and there appears to be scope for increasing this performance gain even further

We're happy to start discussions to look at getting **this work integrated into the Garfield++ codebase** (initially as a branch) to allow people to try it and provide feedback!