From raw data to space points and back again on a GPU

Stephen Nicholas Swatman
Introduction

I’ve been working on CUDA implementations of the algorithms that convert raw data to space points.

Goal of today’s talk:

- Present algorithms and design decisions
- Show results
- Look towards the future
Recap

CPU implementation performs three distinct steps:

- **Clusterization**
  - Which pixel activations belong to the same hit?

- **Measurement creation**
  - What is the weighted center of each hit?

- **Space point creation**
  - What are the global coordinates of each hit, coming from local coordinates?
Clusterization

Clusterization in *traccc* is relatively unique because it is a sparse connected component labeling problem (not very common)

Existing implementation follows the **SparseCCL** algorithm, designed specifically for sparse problems

SparseCCL enforces sequential execution within modules, allowing parallelism only between modules - potentially suboptimal on massively parallel devices
Measurement creation

Measurement creation is a reduction problem, taking the activations in a hit and computing the weighted center of that hit.

This is, in itself, not a particularly difficult problem.
Important insights

Clusterization is a **CCL** problem, measurement creation is a reduction problem, but (measurement creation \( \circ \) clusterization) is a **CCA** problem.

Providing a single algorithm that performs the composition of these algorithms has some distinct advantages:

- Allows us to use dedicated CCA algorithms instead of just CCL algorithms
- Avoids the extremely awkward jagged intermediate EDM

However, it also reduces the granularity at which we can compose algorithms. Need good algorithm semantics to make this work!
A category theory view
Raw to measurements

So, we have a connected component analysis problem (going from raw cells to measurements in one go) with the following requirements:

- The implementation must be performant for very sparse data
- The algorithm must map onto the massive parallelism of GPUs

I have been unable to find any literature on this particular problem, so I had to come up with something myself. Two options:

- Come up with an entirely novel algorithm for this
- Compose existing well-studied algorithms to make this work
Connected component problems come in three flavours: dense, sparse, and graph-based.

Dense and graph-based problems are far more commonly studied than sparse problems, in my exploration of the literature.

<table>
<thead>
<tr>
<th>Flavour</th>
<th>Positioning</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense</td>
<td>Implicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Sparse</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Graph</td>
<td>N/A</td>
<td>Explicit</td>
</tr>
</tbody>
</table>
Connected component flavours

Thankfully, all three flavours are (with some exceptions) reducible to each other!

Dense to sparse and vice versa are trivial, convert between explicit and implicit positioning

Converting a sparse problem consisting of points $P$ to a graph problem can be done by constructing the graph $(P, \{(p_1, p_2) \mid p_1 \in P, p_2 \in P, \text{neighbours}(p_1, p_2)\})$

Converting a graph problem $G$ to a sparse problem is the most complicated, can be done in two dimensions if $G$ is planar, can always be done in three-dimensions (because you can embed any graph in $\mathbb{R}^3$)

Missing links can be found through composition
Sparse to graph

Reducing a sparse problem to a graph problem would, in the general case, be a rather infeasible operation, would take $O(n^2)$ time to speed up an $O(n \log^* n)$ algorithm ($n$ is the number of activations in a module)

However, our data is conveniently sorted! Means we can ignore the vast majority of activations

Two-pass algorithm with short-circuiting, collecting up to four neighbours per pass

Allows us to reduce in $O(n^{\sqrt{n}})$, potentially $O(n)$ (I haven’t had time to develop a good argument for this claim)
FastSV

FastSV is an improvement on the classic Shiloach-Vishkin algorithm, with highly parallelizable properties

Classic Shiloach-Vishkin also available

Designed to work on graph problems, which we have now!

Output is a disjoint-set data structure, here implemented as an array with a few very nice properties

---

**Algorithm 2** The FastSV algorithm. **Input:** $G(V, E)$. **Output:** The parent vector $f$

1: procedure FastSV($V, E$)
2:     for every vertex $u \in V$ do
3:         $f[u], f_{next}[u] \leftarrow u$
4:     repeat
5:         $\triangleright$ Step 1: Stochastic hooking
6:         for every $(u, v) \in E$ do in parallel
7:             $f_{next}[f[u]] \leftarrow \min f[f[v]]$
8:         $\triangleright$ Step 2: Aggressive hooking
9:         for every $(u, v) \in E$ do in parallel
10:            $f_{next}[u] \leftarrow \min f[f[v]]$
11:        $\triangleright$ Step 3: Shortcutting
12:        for every $u \in V$ do in parallel
13:            $f_{next}[u] \leftarrow \min f[f[u]]$
14:        $f \leftarrow f_{next}$
15:        until $f[f]$ remains unchanged
FastSV

We know:

- $\forall k : k \geq f_k$
- $\forall k : f_{f_k} = f_k$
- $\forall k \in f, \exists ! i : k = i = f_k$

Each self-referential index is a parent, and each index reaches a parent in exactly one step.

This is a nice way to represent a jagged clusterization result in GPU memory!

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f[i]$</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f[i]$</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>12</td>
<td>8</td>
<td>12</td>
<td>12</td>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>

{0, 1, 3, 4, 10, 17},
{2, 5, 6, 7, 11},
{8, 9, 13},
{12, 14, 15},
{16}
Aggregation

I mentioned earlier that we could use a CCA algorithm instead of CCL, this is true, but FastSV is actually just a CCL algorithm (it may be possible to extend it to a CCA algorithm, but this comes with some additional implementation details)

For now, use one of two aggregation strategies to reduce the disjoint-set data into measurements
Aggregation 1 - Parent-driven

In this strategy, each parent node collects the data from its children by iterating over all nodes, and adding together the necessary values.

This is an $O(n^2)$ algorithm, but it is actually fairly efficient for the small modules I have been working on in the TrackML dataset.

The ‘quick and hacky’ implementation, but may be feasible for real-world applications despite concerns from ivory-tower complexity theorists.
Aggregation 2 - Child-driven

In the more theoretically sound strategy, children “walk home from school by themselves”, adding their own values to their parent using atomic operations.

This makes for a nice $O(n)$ algorithm, but that is only if you make the generous assumption that atomic operations are a negligible implementation detail.

Improved strategy: have children add their values to one of several values which are then reduced by the parents, may significantly reduce atomic operation overhead, but requires more memory.
Implementation details

I’ve chosen to implement a one-event-one-kernel, one-module-one-block strategy, so we can leverage the GPU’s built-in block scheduling to solve the load imbalance between modules.

Within each block, I use a many-cell-one-thread model, where each thread processes at least $\lfloor n / t \rfloor$ and at most $\lceil n / t \rceil$, where $n$ is the number of activations in the module, and $t$ is the threads per block.

Load imbalance at most one cell, worst case is $n = t + 1$, giving utilization $\frac{1}{2}$, but utilization improves as $n$ increases.
Implementation details

Significant use of shared memory for speed, primarily for the disjoint-set structure

Shared memory usage per block is equal to $2n$ 16-bit integers, so shared memory cap is reached around 8192 activations in a module, giving us a few options:

- Dynamically switch to global memory when necessary, should be a relatively small number of cases, but has significant overhead (balanced out over time due to block scheduling)
- Throw out very large modules (may anger physicists)
- Wait until newer CUDA devices come out with more shared memory
Performance

Directly comparing these algorithms has been a little bit difficult because as mentioned we don’t have strongly defined algorithm semantics yet, so I have been generous to the CPU implementation: comparison between raw to clusters on the CPU and raw to measurements on the GPU

Average over 128 runs on the TrackML pixel data:

- 43.1 ms on the CPU
- 3.8 ms on the GPU

Speed-up factor of 11 times, although that is not considering threading on the CPU
## Performance

### Performance breakdown in thousands of cycles

<table>
<thead>
<tr>
<th></th>
<th>Reduction</th>
<th>Clusterization</th>
<th>Aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contr</td>
<td>23.8%</td>
<td>39.6%</td>
<td>36.6%</td>
</tr>
<tr>
<td>Mean</td>
<td>12.95</td>
<td>6.03</td>
<td>10.9</td>
</tr>
<tr>
<td>Min</td>
<td>0.50</td>
<td>0.69</td>
<td>1.31</td>
</tr>
<tr>
<td>Max</td>
<td>2039.3</td>
<td>1833.9</td>
<td>2058.2</td>
</tr>
<tr>
<td>25%</td>
<td>1.56</td>
<td>3.05</td>
<td>2.05</td>
</tr>
<tr>
<td>50%</td>
<td>1.85</td>
<td>3.77</td>
<td>2.81</td>
</tr>
<tr>
<td>75%</td>
<td>2.05</td>
<td>4.68</td>
<td>4.94</td>
</tr>
</tbody>
</table>
Scaling up

Question is: is this data not too small for the GPU to really shine?

As far as I know, we’re doing all of this so we can prepare our compute for HL-LHC and other future settings with high data throughput

How can we test the performance of the software at larger scales? Synthesize input data!
Synthesizing input

An event consists of a number of modules of a certain size, each with a number of hits, and each hit having a number of activations

The last two are seemingly distributed in a log-normal fashion

Obviously these things are not independent in the real world, but they are as far as our input generating model is concerned (considering only factors which might affect performance)

These synthesized inputs will have zero physical meaning! Only meant to stress the algorithms at a very fine-tunable level
I’ve modeled the events according to 6 parameters:

- The number of modules $N$
- The size of each module $S \times S$
- The distribution of hits per module $M \sim \text{Lognormal}(\mu_M, \sigma_M^2)$
- The distribution of activations per hit $H \sim \text{Lognormal}(\mu_H, \sigma_H^2)$

This allows us to estimate how many activations (our predicted main performance-determining feature) there will be in the ‘event’:

$$|A| = N \times \exp(\mu_M + 0.5 \times \sigma_M^2) \times \exp(\mu_H + 0.5 \times \sigma_H^2)$$
Synthesizing input

For the TrackML pixels I find:

- $N = 2500$
- $S = 655$
- $\mu_M = 1.65$
- $\sigma_M = 0.95$
- $\mu_H = 1.80$
- $\sigma_H = 0.89$

Seems to match the existing outputs very well in terms of structure
Synthesizing input

To test the scaling properties, double the complexity of our event at every step through one of five ways:

- \( N_{n+1} = 2N_n \)  
  double the number of modules (larger experiment)

- \( \mu_{M,n+1} = \mu_{M,n} + \ln 2 \)  
  double the mean hits per module (higher pile-up?)

- \( \sigma_{M,n+1} = \sqrt{2 \ln 2 + \sigma_{M,n}^2} \)  
  increase the variance in hits per module

- \( \mu_{H,n+1} = \mu_{H,n} + \ln 2 \)  
  double the mean activations per hit (higher energy?)

- \( \sigma_{H,n+1} = \sqrt{2 \ln 2 + \sigma_{H,n}^2} \)  
  increase the variance in activations (wider energy range)

Can be combined in arbitrary ways, but this sadly creates a five-dimensional experimental design space, not really feasible to exhaustively test.
Scaling up

To test performance on bigger inputs, I generated 23 new inputs: scaled up by factors 2, 4, 8, 16, (32, and 64) in all five directions from the TrackML starting point.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>2500</td>
<td>5000</td>
<td>10000</td>
<td>20000</td>
<td>40000</td>
<td>80000</td>
<td>160000</td>
</tr>
<tr>
<td>$\mu_M$</td>
<td>1.65</td>
<td>2.34</td>
<td>3.04</td>
<td>3.73</td>
<td>4.42</td>
<td>Not applicable (too many modules over activation cap)</td>
<td></td>
</tr>
<tr>
<td>$\sigma_M$</td>
<td>0.95</td>
<td>1.51</td>
<td>1.92</td>
<td>2.25</td>
<td>2.54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mu_H$</td>
<td>1.80</td>
<td>2.49</td>
<td>3.19</td>
<td>3.88</td>
<td>4.57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\sigma_H$</td>
<td>0.89</td>
<td>1.48</td>
<td>1.89</td>
<td>2.23</td>
<td>2.52</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Disclaimer: the following results were not taken in a sufficiently clean environment and are noisy.
## Scaling up - N

<table>
<thead>
<tr>
<th>Scale</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>2500</td>
<td>5000</td>
<td>10000</td>
<td>20000</td>
<td>40000</td>
<td>80000</td>
<td>160000</td>
</tr>
<tr>
<td>CPU time</td>
<td>44.9</td>
<td>84.4</td>
<td>170.9</td>
<td>340.6</td>
<td>691.8</td>
<td>1353.5</td>
<td>2755.0</td>
</tr>
<tr>
<td>GPU time</td>
<td>3.8</td>
<td>8.0</td>
<td>14.8</td>
<td>29.8</td>
<td>54.9</td>
<td>109.9</td>
<td>221.3</td>
</tr>
<tr>
<td>CPU to base</td>
<td>1.00</td>
<td>1.88</td>
<td>3.81</td>
<td>7.59</td>
<td>15.4</td>
<td>30.1</td>
<td>61.4</td>
</tr>
<tr>
<td>GPU to base</td>
<td>1.00</td>
<td>2.11</td>
<td>3.89</td>
<td>7.84</td>
<td>14.4</td>
<td>28.9</td>
<td>58.2</td>
</tr>
<tr>
<td>GPU speedup</td>
<td>11.8</td>
<td>10.55</td>
<td>11.5</td>
<td>11.4</td>
<td>12.6</td>
<td>12.3</td>
<td>12.4</td>
</tr>
</tbody>
</table>
## Scaling up - $\mu_M$

<table>
<thead>
<tr>
<th>Scale</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>CPU to base</th>
<th>GPU to base</th>
<th>GPU speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu_M$</td>
<td>1.65</td>
<td>2.34</td>
<td>3.04</td>
<td>3.73</td>
<td>4.42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time</td>
<td>44.9</td>
<td>83.8</td>
<td>174.9</td>
<td>331.8</td>
<td>654.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU time</td>
<td>3.8</td>
<td>6.6</td>
<td>12.3</td>
<td>27.1</td>
<td>60.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU to base</td>
<td>1.00</td>
<td>1.87</td>
<td>3.90</td>
<td>7.40</td>
<td>14.57</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU to base</td>
<td>1.00</td>
<td>1.74</td>
<td>3.24</td>
<td>7.13</td>
<td>15.84</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU speedup</td>
<td>11.8</td>
<td>12.7</td>
<td>14.2</td>
<td>12.2</td>
<td>10.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not applicable
Scaling up - $\sigma_M$

<table>
<thead>
<tr>
<th>Scale</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_M$</td>
<td>0.95</td>
<td>1.51</td>
<td>1.92</td>
<td>2.25</td>
<td>2.54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time</td>
<td>44.9</td>
<td>81.6</td>
<td>156.0</td>
<td>390.5</td>
<td>759.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU time</td>
<td>3.8</td>
<td>7.5</td>
<td>24.3</td>
<td>34.8</td>
<td>50.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU to base</td>
<td>1.00</td>
<td>1.82</td>
<td>3.47</td>
<td>8.70</td>
<td>16.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU to base</td>
<td>1.00</td>
<td>1.97</td>
<td>6.39</td>
<td>9.16</td>
<td>13.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU speedup</td>
<td>11.8</td>
<td>10.9</td>
<td>6.4</td>
<td>11.2</td>
<td>15.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Scaling up - $\mu_H$

<table>
<thead>
<tr>
<th>Scale</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu_H$</td>
<td>1.80</td>
<td>2.49</td>
<td>3.19</td>
<td>3.88</td>
<td>4.57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time</td>
<td>44.9</td>
<td>83.6</td>
<td>165.2</td>
<td>386.3</td>
<td>945.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU time</td>
<td>3.8</td>
<td>6.0</td>
<td>11.2</td>
<td>20.9</td>
<td>42.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU to base</td>
<td>1.00</td>
<td>1.86</td>
<td>3.68</td>
<td>8.60</td>
<td>21.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU to base</td>
<td>1.00</td>
<td>1.58</td>
<td>2.95</td>
<td>5.50</td>
<td>11.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU speedup</td>
<td>11.8</td>
<td>13.9</td>
<td>14.8</td>
<td>18.5</td>
<td>22.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Scaling up - $\sigma_H$

<table>
<thead>
<tr>
<th>Scale</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_H$</td>
<td>0.89</td>
<td>1.48</td>
<td>1.89</td>
<td>2.23</td>
<td>2.52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time</td>
<td>44.9</td>
<td>99.0</td>
<td>300.4</td>
<td>1518.4</td>
<td>3841.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU time</td>
<td>3.8</td>
<td>8.2</td>
<td>14.7</td>
<td>25.4</td>
<td>42.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU to base</td>
<td>1.00</td>
<td>2.20</td>
<td>6.69</td>
<td>33.82</td>
<td>85.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU to base</td>
<td>1.00</td>
<td>2.16</td>
<td>3.87</td>
<td>6.68</td>
<td>11.16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU speedup</td>
<td>11.8</td>
<td>12.1</td>
<td>20.4</td>
<td>59.8</td>
<td>90.6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not applicable
Additional results

On top of these benchmarks being synthetic by themselves, the scaling is also synthetic: the values would likely not scale so independently of each other.

As a test, generate another file, $2^5$ times larger, by doubling in every direction, average over 16 runs:

- 2619.8 ms on the CPU
- 95.3 ms on the GPU

Speed-up of 30 times here!
Space point formation

Relatively boring problem, to be completely honest

Maps very well onto GPUs, in either a one-hit-one-block or a one-hit-one-thread model

One-hit-one-block doesn’t perform as well, as matrix multiplication of homogeneous three-dimensional coordinates isn’t really enough work to fill up a block

Also, little to no load imbalance in a one-hit-one-thread model

Possible future work: use tensor cores for these matrix multiplications
Conclusions

Working GPU-only implementations of all the code required to convert raw input to space points

When you can’t find a solution to your problem, just convert your problem to a different problem

Scaling strongly depends on which parameters of the input are scaled, for some the speedup of GPU code is a consistent factor 10, for other parameters the scaling of the GPU code is much stronger

Can’t find any cases where the GPU implementation performs or scales worse than the CPU implementation