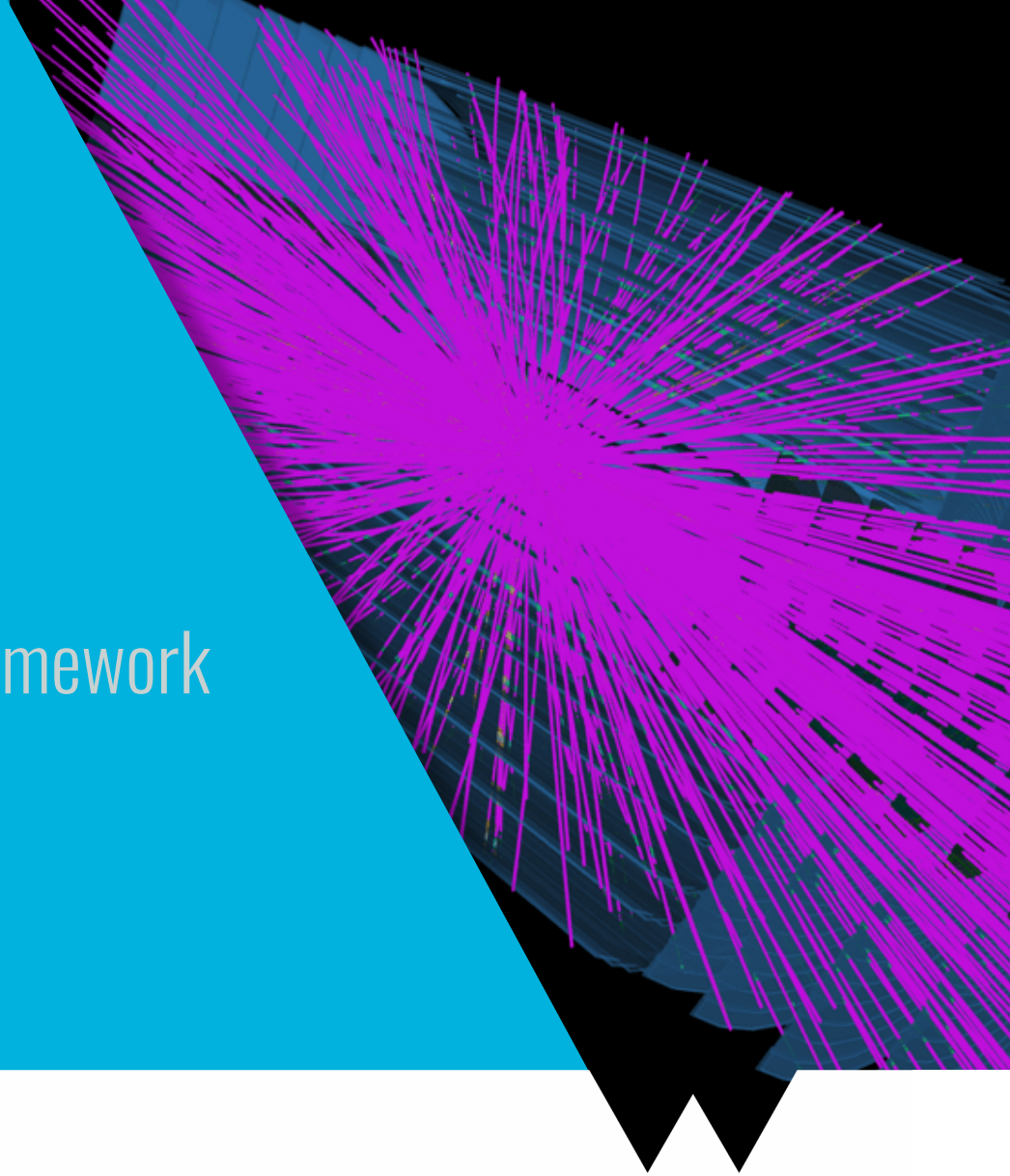
traccc

Integrating the Alpaka framework

Ryan Cross

2023/11/22



WARWICK
THE UNIVERSITY OF WARWICK

Overview

This talk will cover:

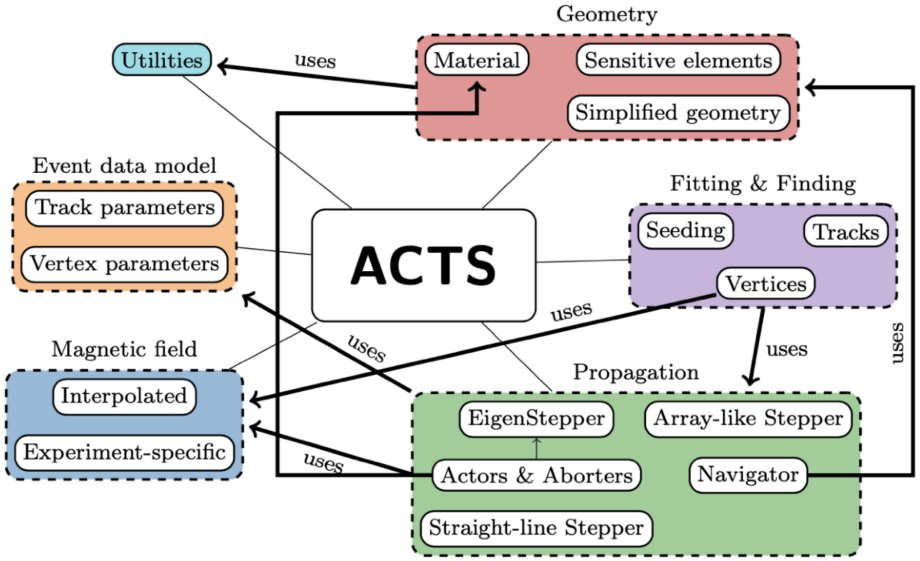
1. **A quick overview of traccc.**
2. **Cross-Platform Abstraction Libraries.**
3. **Recent Work**
4. **What comes next?**

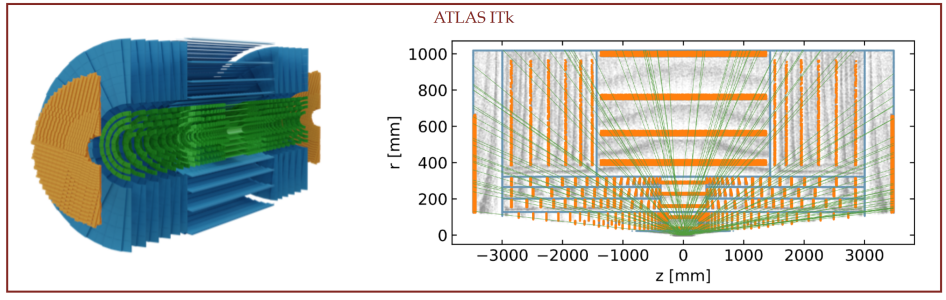
A Common Tracking Software

ACTS is a generic, experiment independent framework/software toolkit, written in C++. Through it, you can get algorithms for track reconstruction that can be used in any experiment, agnostic of any technical details (detector tech, design and event processing framework).

It has been designed in a thread-safe manner, with support for parallel code execution and optimised data structures for speeding up the many linear algebra operations used throughout the code base.

Wide set of use cases, with integrations/progress for Belle II, CEPC, sPHENIX, PANDA, FASER, ATLAS ID (current + ITk).



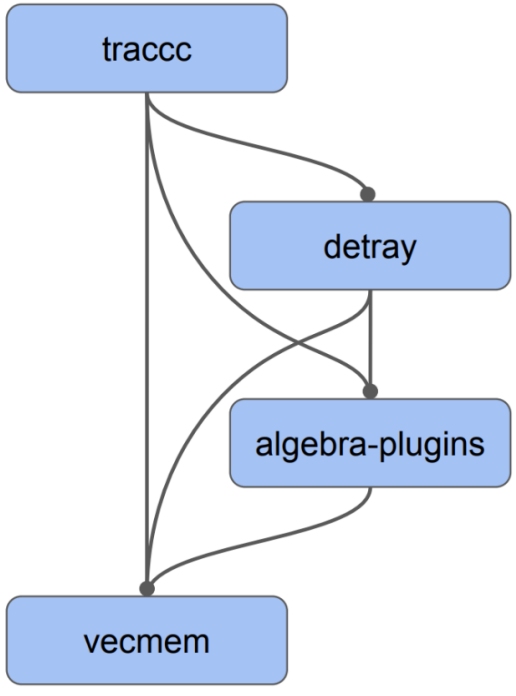


ACTS R&D Projects

Many of the core algorithms in ACTS have been ported to CUDA and SYCL, but there is a limit as to how far this can go. Full offloading is difficult, with some of the event data model and geometry not being the most GPU-friendly.

To tackle this, ACTS has launched several R&D projects:

- traccc - Tracking Algorithms on the GPU.
- detray - A GPU based Geometry Builder.
- algebra-plugin - Provides varying algebra plugins for the other projects.
- vecmem - A GPU Memory Management Tool for the other projects.

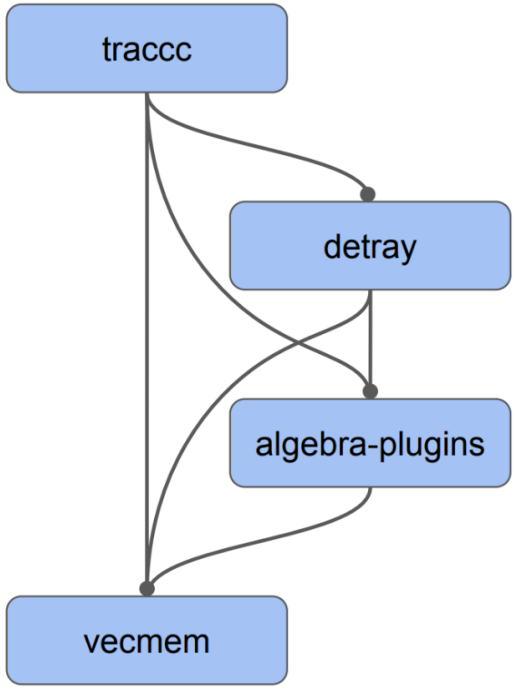


ACTS R&D Projects

Many of the core algorithms in ACTS have been ported to CUDA and SYCL, but there is a limit as to how far this can go. Full offloading is difficult, with some of the event data model and geometry not being the most GPU-friendly.

To tackle this, ACTS has launched several R&D projects:

- traccc - Tracking Algorithms on the GPU.
- detray - A GPU based Geometry Builder.
- algebra-plugin - Provides varying algebra plugins for the other projects.
- vecmem - A GPU Memory Management Tool for the other projects.



traccc specifically, is aiming to establish a sensible event data model and algorithms that are able to exploit parallelisation architecture, whilst relying heavily on the other projects.

Cross-Platform Abstraction - Why?

As part of traccc, the use of abstraction tools are interesting. There are many different ways to write code that can run on a GPU.

Writing CUDA directly can be a mixed experience. You are locked into a specific vendor for acceleration, but you don't have to deal with the additional layer of complexity an abstraction library brings.

Because of this, there exists many different abstraction paradigms, software that allows a single source code base that at compile time can target many architectures. Common targets include CUDA, AMD / Intel GPUs, and CPU parallelism via `thread`, OpenMP, TBB and more.

Cross-Platform Abstraction - What?

There is a few approaches worth talking about in the context of traccc. Whilst the broad goal of allowing a single code base to target many different accelerator backends is the same, the approach and technical details differ.

Cross-Platform Abstraction - What?

There is a few approaches worth talking about in the context of traccc. Whilst the broad goal of allowing a single code base to target many different accelerator backends is the same, the approach and technical details differ.

- **SYCL** is a higher level programming model, developed by the Khronos group (OpenCL/OpenGL/Vulkan and more). It defines an abstraction layer that enables code for heterogeneous processors via a 'single-source' style in standard C++. Supports many backends: CUDA, AMD GPUs, Intel GPUs, OpenMP, MPI, Vulkan, `std::thread`, OpenCL and more.

Cross-Platform Abstraction - What?

There is a few approaches worth talking about in the context of traccc. Whilst the broad goal of allowing a single code base to target many different accelerator backends is the same, the approach and technical details differ.

- **SYCL** is a higher level programming model, developed by the Khronos group (OpenCL/OpenGL/Vulkan and more). It defines an abstraction layer that enables code for heterogeneous processors via a 'single-source' style in standard C++. Supports many backends: CUDA, AMD GPUs, Intel GPUs, OpenMP, MPI, Vulkan, `std::thread`, OpenCL and more.

- **Kokkos** is C++ based programming model, which provides methods that abstract away details of parallel execution and memory management, such that code can be written for many shared-memory programming models in a unifed way. Supports CUDA, HIP, SYCL, HPX, OpenMP and `std::thread`.

Cross-Platform Abstraction - What?

There is a few approaches worth talking about in the context of traccc. Whilst the broad goal of allowing a single code base to target many different accelerator backends is the same, the approach and technical details differ.

- **SYCL** is a higher level programming model, developed by the Khronos group (OpenCL/OpenGL/Vulkan and more). It defines an abstraction layer that enables code for heterogeneous processors via a 'single-source' style in standard C++. Supports many backends: CUDA, AMD GPUs, Intel GPUs, OpenMP, MPI, Vulkan, `std::thread`, OpenCL and more.

- **Kokkos** is C++ based programming model, which provides methods that abstract away details of parallel execution and memory management, such that code can be written for many shared-memory programming models in a unifed way. Supports CUDA, HIP, SYCL, HPX, OpenMP and `std::thread`.

- **alpaka** is a header-only C++ 17 abstraction library for accelerator development. It aims to provide performance portability across a range of accelerators through the abstraction of the underlying levels of parallelism. Support CUDA, OpenMP, `std::thread`, TBB, HIP and OpenAcc.

Cross-Platform Abstraction - How?

Despite having differing ways of interacting with them, advertising themselves differently and more...they all have the same objective: **Write your code once**, and through the libraries abstraction methods, end up with a code base that supports a variety of accelerator backends.

The specific interface to achieve this differs between each of the options, but some broad steps are the same.

Cross-Platform Abstraction - How?

Despite having differing ways of interacting with them, advertising themselves differently and more...they all have the same objective: **Write your code once**, and through the libraries abstraction methods, end up with a code base that supports a variety of accelerator backends.

The specific interface to achieve this differs between each of the options, but some broad steps are the same.

Get an accelerator device:

```
accelerator = getAcceleratorDevice();
queue = getDeviceQueue(accelerator);
```

Cross-Platform Abstraction - How?

Despite having differing ways of interacting with them, advertising themselves differently and more...they all have the same objective: **Write your code once**, and through the libraries abstraction methods, end up with a code base that supports a variety of accelerator backends.

The specific interface to achieve this differs between each of the options, but some broad steps are the same.

Get an accelerator device:

```
accelerator = getAcceleratorDevice();
queue = getDeviceQueue(accelerator);
```

Define an operation for the device to perform:

```
job = [](auto accelerator, auto config, auto items) {
    auto item = items[getThreadIndex()];
    ...
};
```

Cross-Platform Abstraction - How?

Despite having differing ways of interacting with them, advertising themselves differently and more...they all have the same objective: **Write your code once**, and through the libraries abstraction methods, end up with a code base that supports a variety of accelerator backends.

The specific interface to achieve this differs between each of the options, but some broad steps are the same.

Get an accelerator device:

```
accelerator = getAcceleratorDevice();
queue = getDeviceQueue(accelerator);
```

Define an operation for the device to perform:

```
job = [](auto accelerator, auto config, auto items) {
    auto item = items[getThreadIndex()];
    ...
};
```

Run the jobs in parallel:

```
queue.submit(job, configuration, items);
queue.wait();
```

Why alpaka?

I've just outlined three projects that support the "write once, support many" paradigm, and both SYCL and Kokkos are already implemented in traccc, with differing levels of functionality. So why a third?

alpaka was chosen as a possible candidate for a few reasons:

- **Simplicity**: alpaka is a lightweight, header-only library, which makes integration into traccc very easy, as well as it being written in the same modern C++17 as traccc/acts.
- **Familiarity**: The alpaka abstraction model is very similar to the CUDA grid-blocks-thread model, making writing code for alpaka simple, and familiar for those with CUDA experience, whilst also providing a CPU and non-CUDA based implementation.
- **Community Support**: alpaka has been used extensively at CMS, including in `cms-sw` and their HLT achieving performance close to that of the native CUDA codebase, from a single source code that can be utilised on many devices.

Current Progress

The first steps around integration of alpaka in traccc were performed by Stewart Martin-Haugh, as part of a PR in January: PR #300.

I then built upon this base to add the first tracking code, to add a spacepoint binning algorithm. This algorithm is a reasonable starting point, fairly self-contained and easy to implement. This was added in PR #431.

Current Progress

The first steps around integration of alpaka in traccc were performed by Stewart Martin-Haugh, as part of a PR in January: PR #300.

I then built upon this base to add the first tracking code, to add a spacepoint binning algorithm. This algorithm is a reasonable starting point, fairly self-contained and easy to implement. This was added in PR #431.

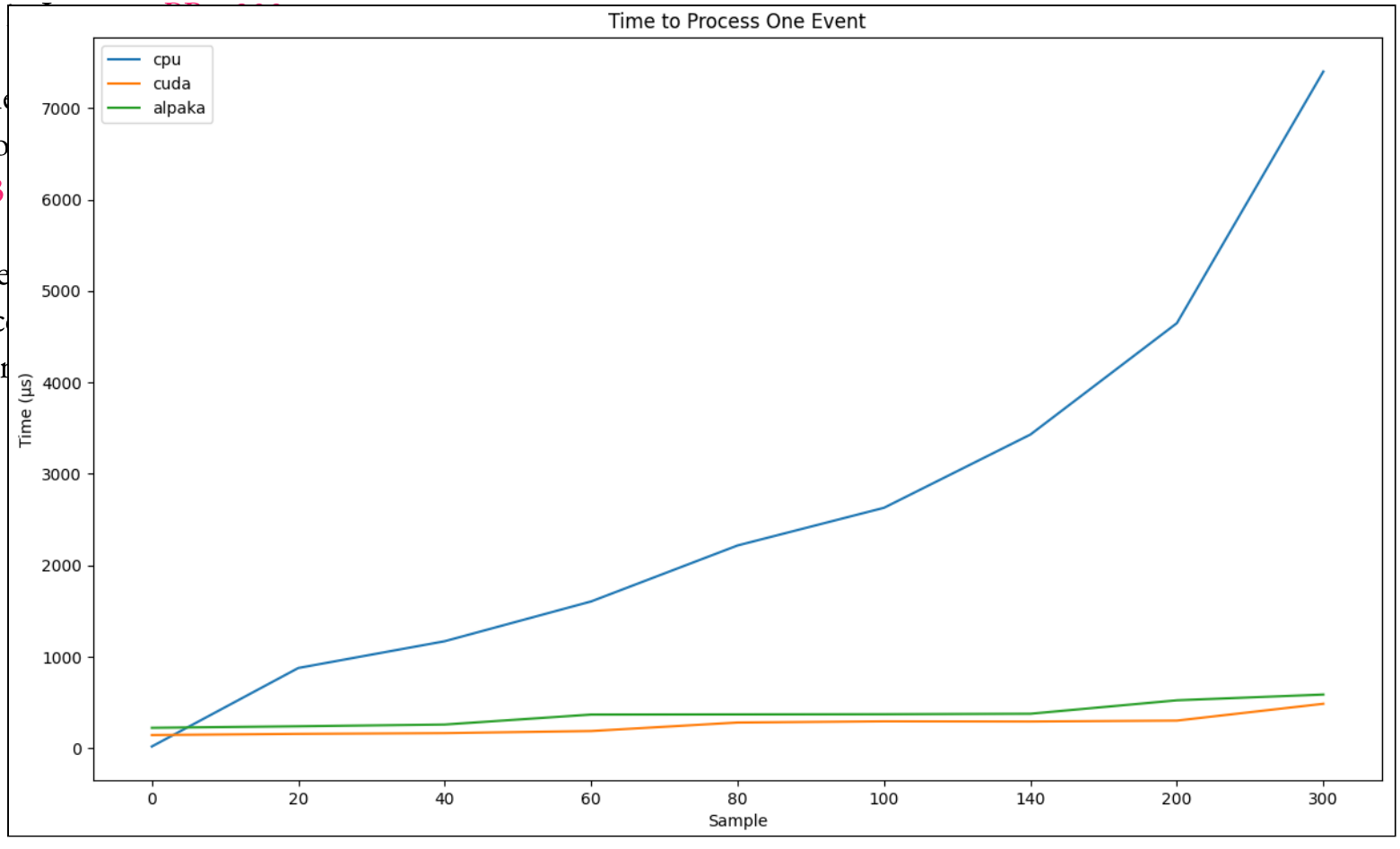
The spacepoint binning gave me a first look at development with Alpaka, as well as developing inside of traccc/ACTS. My previous slides, given at a UK SWIFT-HEP / GRIDPP meeting, give a bit of a better overview of that work, as well as some more basic comparisons of Alpaka vs CUDA.

Current Progress

The first steps around integration of alpaka in traccc were performed by Stewart Martin-Haugh, as part of a PR

I th
algo
#43

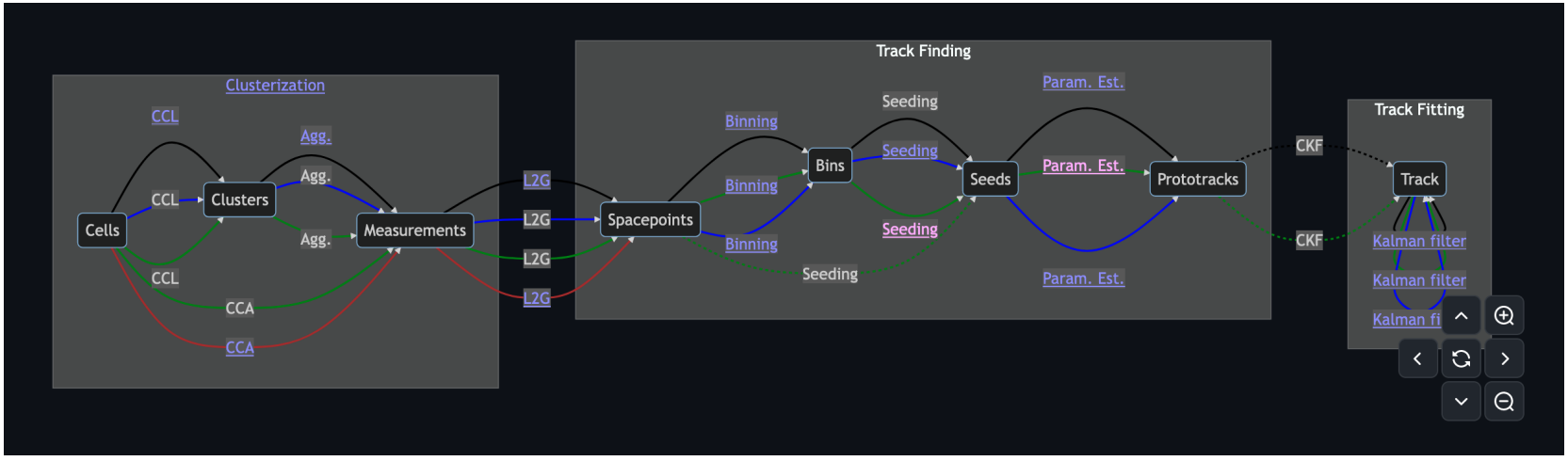
The
trac
over



Recent Work - Seeding Algorithms

Following the Spacepoint Binning work, the next logical step is to fill out the rest of the seeding process.

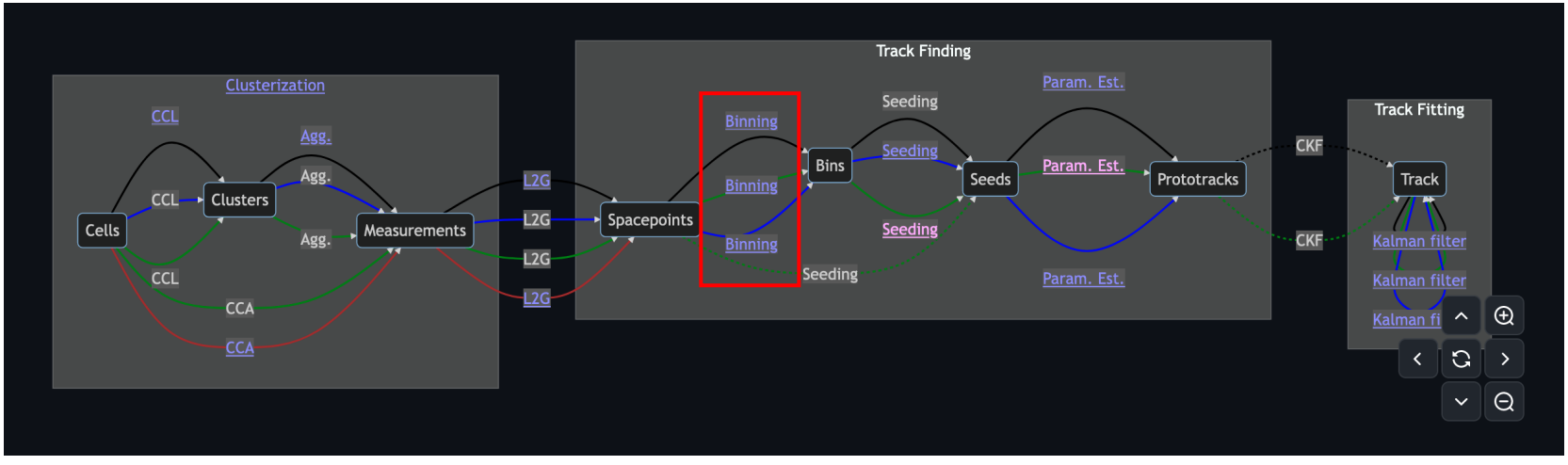
If we look at the diagram given in the `traccc` Git repo:



Recent Work - Seeding Algorithms

Following the Spacepoint Binning work, the next logical step is to fill out the rest of the seeding process.

If we look at the diagram given in the `traccc` Git repo:

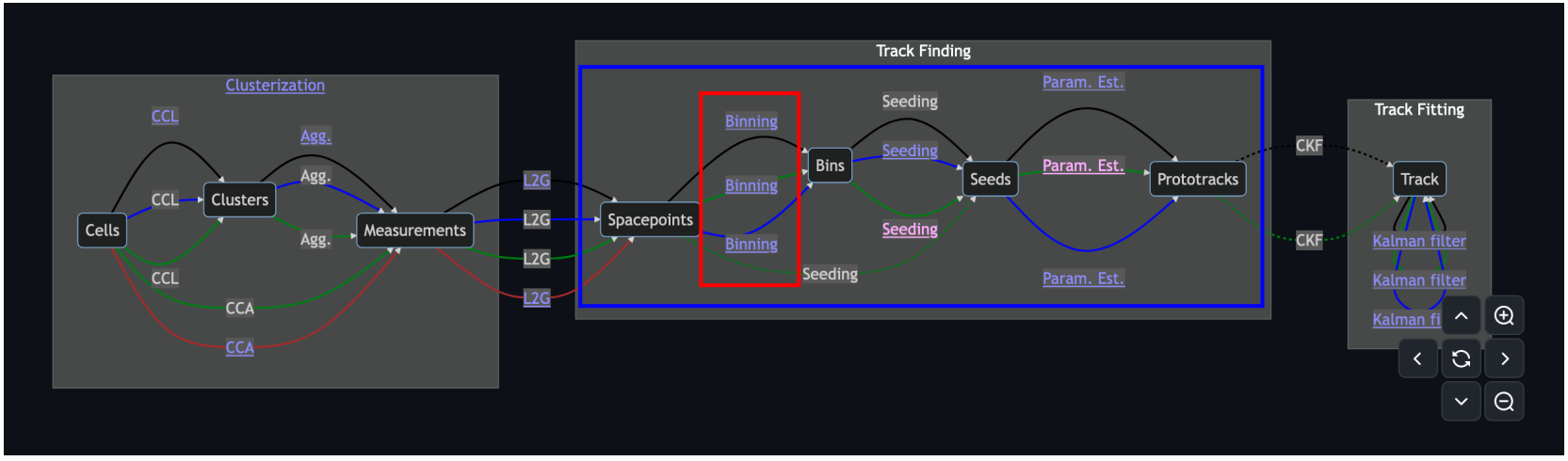


We can see that the previously added spacepoint binning, makes up a part of a larger track finding step, that relies on the production of track seeds.

Recent Work - Seeding Algorithms

Following the Spacepoint Binning work, the next logical step is to fill out the rest of the seeding process.

If we look at the diagram given in the `traccc` Git repo:



We can see that the previously added spacepoint binning, makes up a part of a larger track finding step, that relies on the production of track seeds.

This process is implemented across a few different algorithms, with a seed finding algorithm consuming the previously implemented binned spacepoints, before performing some track parameter estimation using the result of the seeding.

Seeding Algorithms

The actual implementation of these new algorithms mostly proceeds the same as the implementation of the spacepoint binning, which in turn mimics CUDA development. Device code is written in to kernels, with those kernels being called with some form of launch parameters later in the code.

Seeding Algorithms

The actual implementation of these new algorithms mostly proceeds the same as the implementation of the spacepoint binning, which in turn mimics CUDA development. Device code is written in to kernels, with those kernels being called with some form of launch parameters later in the code.

A couple of new (to me in Alpaka at least) development points did pop up with the latest bit of work:

- Shared memory is supported in Alpaka, and is used as part of the seed finding. (I believe it uses system memory for CPU-based accelerator targets, and then the expected `__shared__` memory on GPUs).
- It is a bit of a pain/verbose, at least as a new developer to Alpaka, but once added, works well.

Outside of that though, most of the complication in adding these algorithms is just ensuring that everything is ported correctly to use the correct Alpaka handlers, and is copied to/from the devices as expected.

Seeding Algorithms - Results

With it implemented however, we can now start to look at something a touch more representative, and also turn on the CPU-comparisons, to compare the produced track parameters from the Alpaka implementation to the reference CPU implementation.

Seeding Algorithms - Results

With it implemented however, we can now start to look at something a touch more representative, and also turn on the CPU-comparisons, to compare the produced track parameters from the Alpaka implementation to the reference CPU implementation.

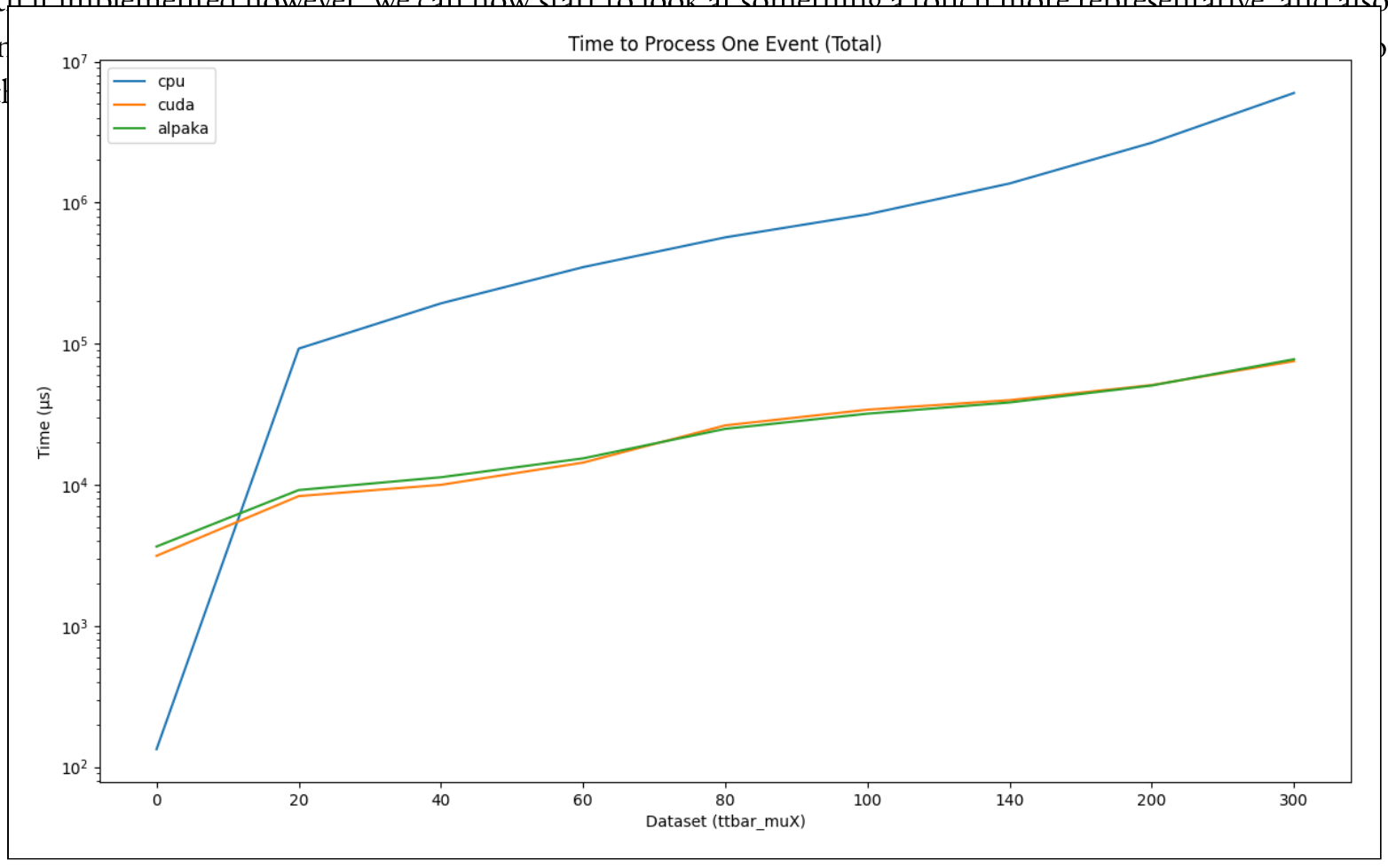
Time to Process One Event (μs)

Sample	CPU	CUDA	alpaka
single_muon	133.53	3136.5	3650.28
ttbar_mu20	92296.1	8311.2	9174.8
ttbar_mu40	193081.8	9989.3	11305.9
ttbar_mu60	348806.3	14360.4	15385
ttbar_mu80	567138.6	26362.5	24915.4
ttbar_mu100	825637.3	34054.8	31915.5
ttbar_mu140	1367165	39802.2	38342.8
ttbar_mu200	2659153.9	50907.1	50540.8
ttbar_mu300	5986834.4	75209.7	77584.5

Performed on a i9-10980XE and RTX A5000, average over 10 runs.

Seeding Algorithms - Results

With it implemented however, we can now start to look at something a touch more representative, and also turn
to th



Performed on a i9-10980XE and RTX A5000, average over 10 runs.

Result Breakdown

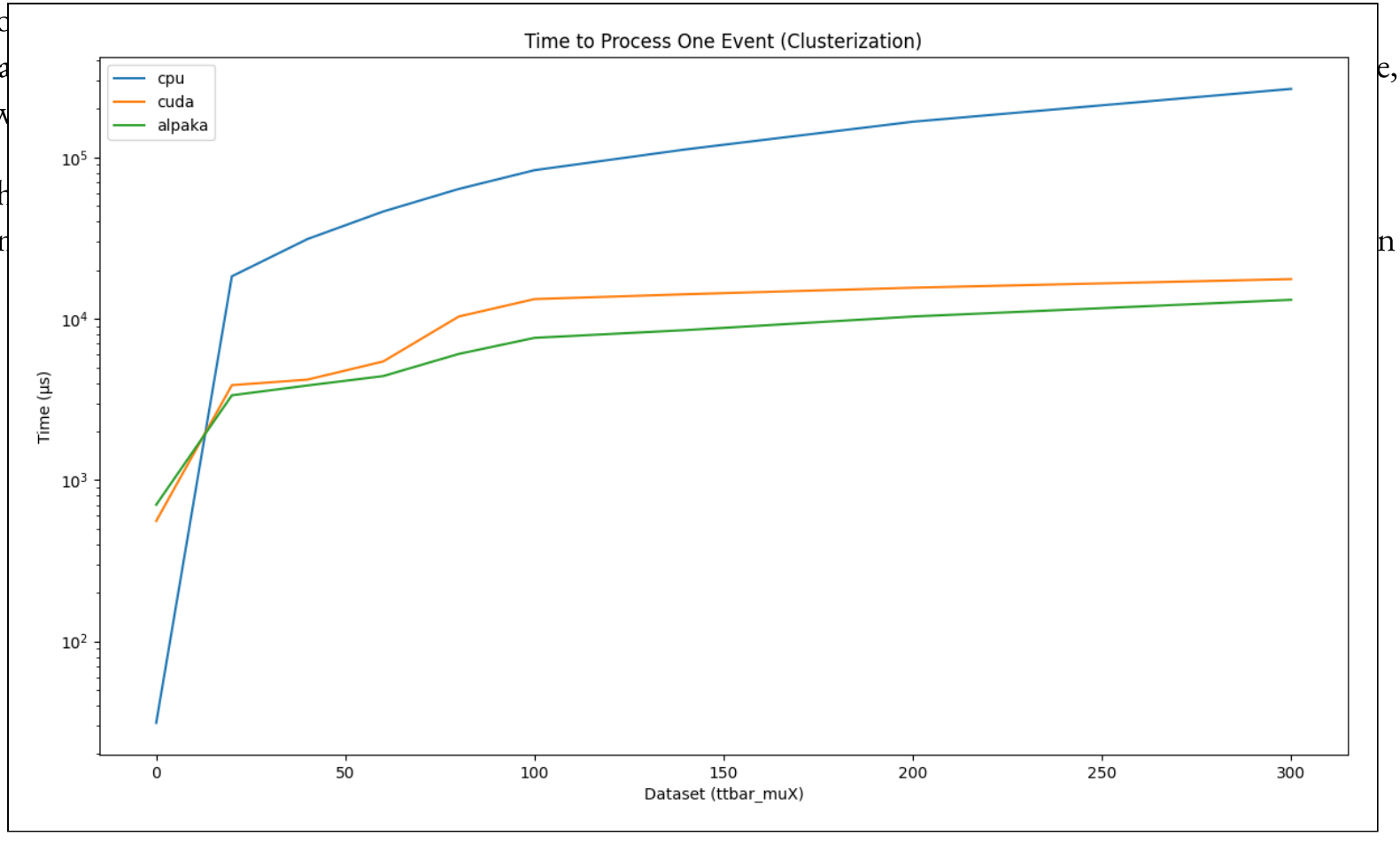
One interesting thing we can look at, is that these numbers are the average total time to process one event through the `seq_example`, which runs the clusterization + spacepoint creation, then seeding, then track parameter estimation. Because of this, we actually get three numbers out, which make up the total run time, so we can go into a little bit more detail to see how each of the sub-components take to run.

It should be noted that these numbers are interesting from a development point-of-view, but these smaller examples aren't completely representative of the "real" performance of the algorithms, as they are running in a less realistic way.

Result Breakdown

One interesting thing we can look at, is that these numbers are the average total time to process one event thro
para
so w

It sh
exar
less



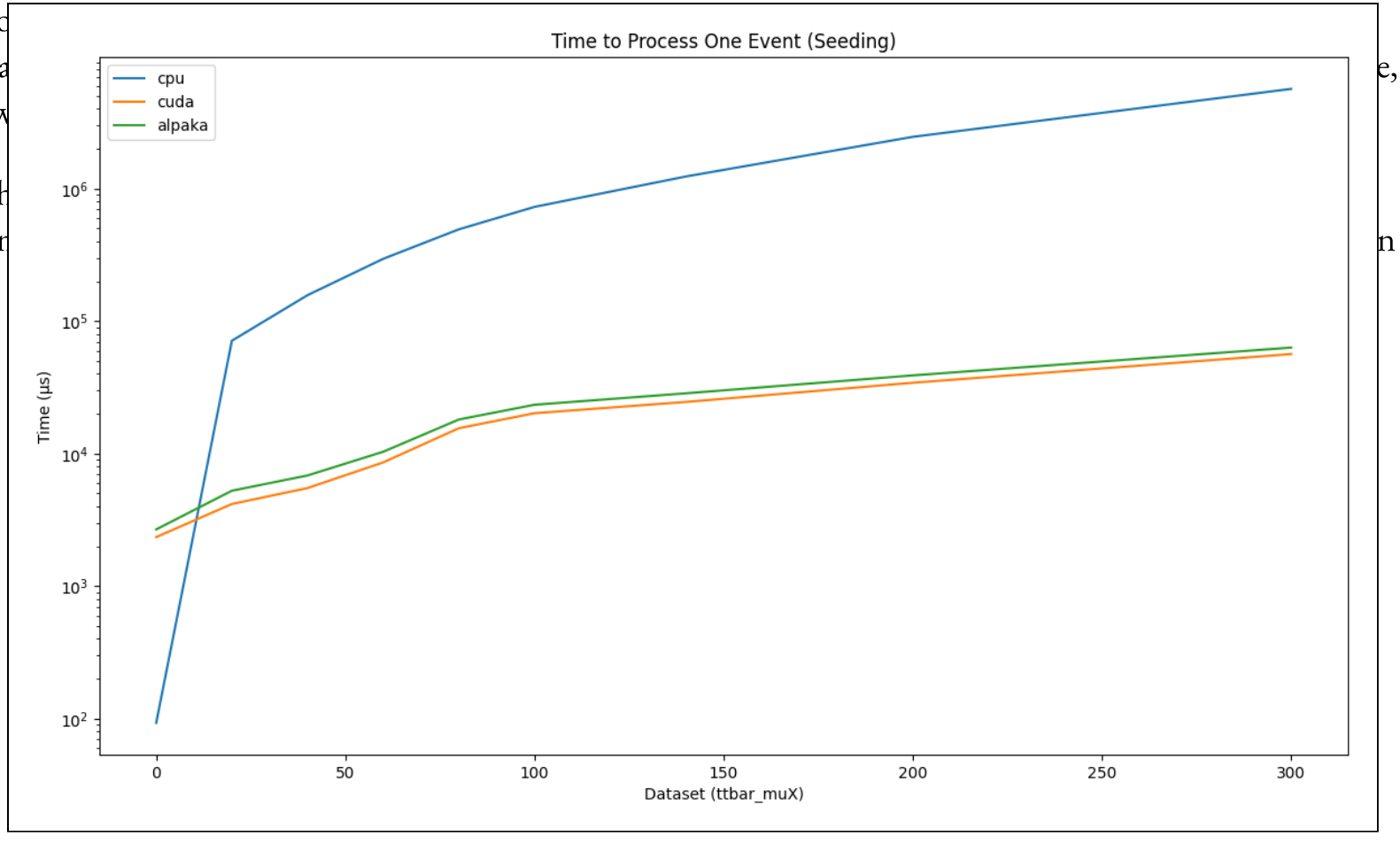
e,

n a

Result Breakdown

One interesting thing we can look at, is that these numbers are the average total time to process one event thro
para
so w

It sh
exar
less



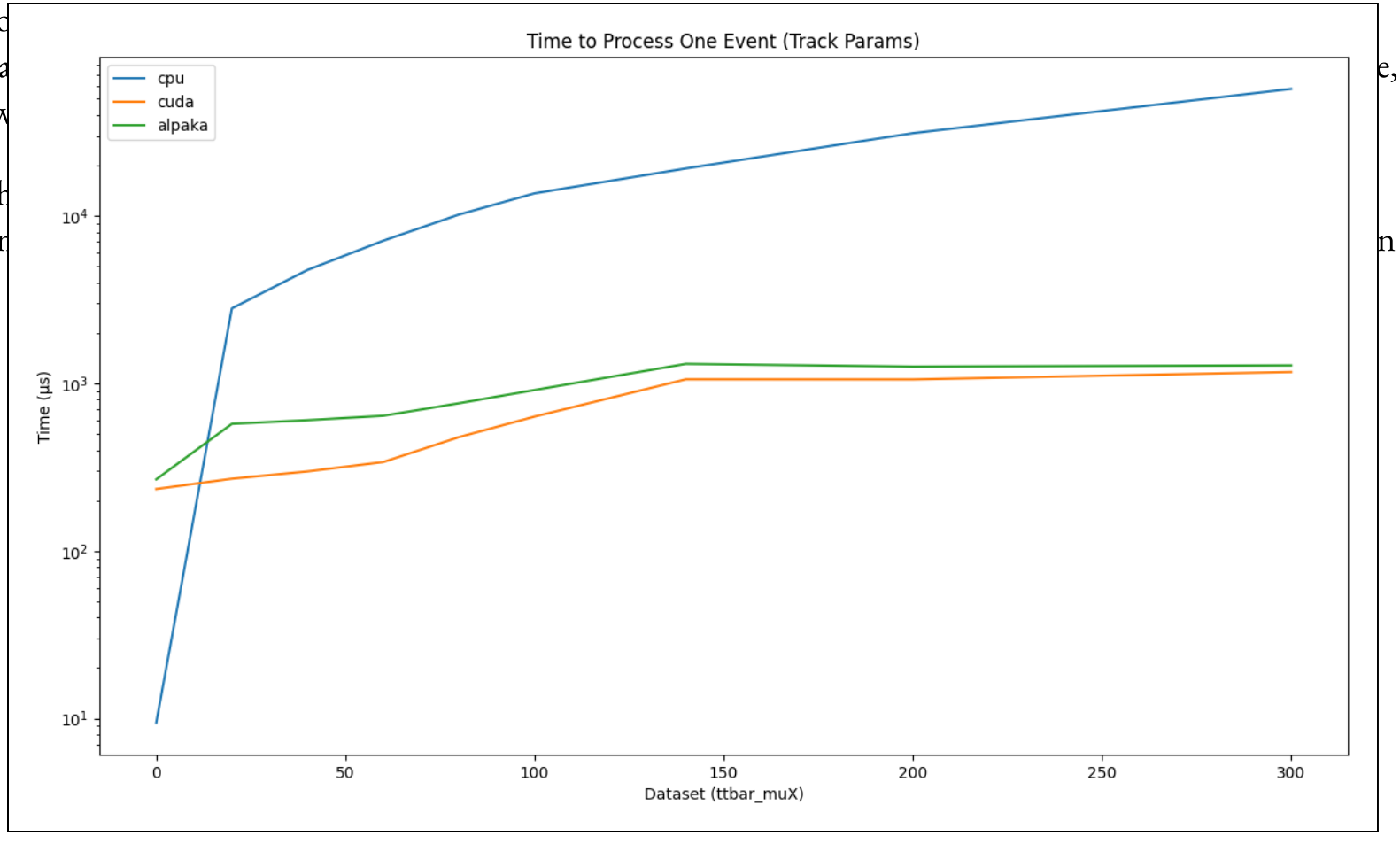
e,

n a

Result Breakdown

One interesting thing we can look at, is that these numbers are the average total time to process one event thro
para
so w

It sh
exar
less



Result Breakdown

One interesting thing we can look at, is that these numbers are the average total time to process one event through the `seq_example`, which runs the clusterization + spacepoint creation, then seeding, then track parameter estimation. Because of this, we actually get three numbers out, which make up the total run time, so we can go into a little bit more detail to see how each of the sub-components take to run.

It should be noted that these numbers are interesting from a development point-of-view, but these smaller examples aren't completely representative of the "real" performance of the algorithms, as they are running in a less realistic way.

There is some interesting features there!

- The clusterization seems to be consistently actually faster inside of Alpaka!
  - It is entirely possible that I'm not doing a 100% fair comparison...the CUDA code uses a decent amount of hard-coded, but sane, parameters, so if I have utilised something more dynamic such that the powerful GPU here is utilised more, that makes sense.

Result Breakdown

One interesting thing we can look at, is that these numbers are the average total time to process one event through the `seq_example`, which runs the clusterization + spacepoint creation, then seeding, then track parameter estimation. Because of this, we actually get three numbers out, which make up the total run time, so we can go into a little bit more detail to see how each of the sub-components take to run.

It should be noted that these numbers are interesting from a development point-of-view, but these smaller examples aren't completely representative of the "real" performance of the algorithms, as they are running in a less realistic way.

There is some interesting features there!

- The clusterization seems to be consistently actually faster inside of Alpaka!
  - It is entirely possible that I'm not doing a 100% fair comparison...the CUDA code uses a decent amount of hard-coded, but sane, parameters, so if I have utilised something more dynamic such that the powerful GPU here is utilised more, that makes sense.

- In the remaining plots, there is a fairly consistent offset between the Alpaka and CUDA implementations. There is still 1 or 2 missing bits from my Alpaka implementation, with async being the most notable difference. Potential that some of the work done at CMS could make its way back into Alpaka proper, to help here (i.e. caching allocator).

More Recent Work

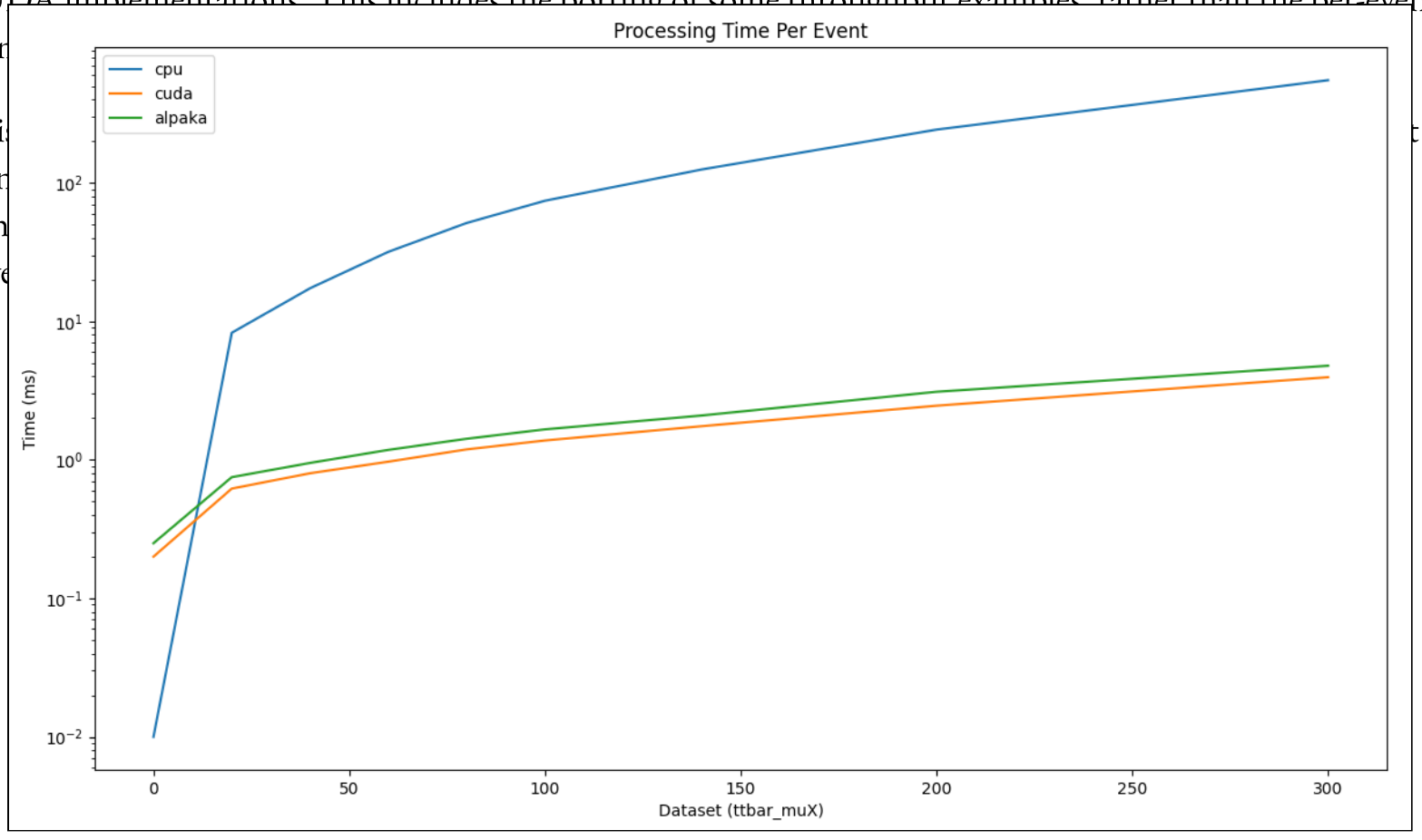
More recently, I've been filling out even more of the algorithms into Alpaka, starting from the existing CUDA implementations. This includes the porting of some throughput examples, rather than the per-event examples shown so far.

This represents a more fair comparison, with the appropriate amount of setup and cold-runs, as well as just being more realistic in the setup/tear-down process for if this code was being used to process many events, rather than the previous examples which are much more designed around looking at a single event from a development point-of-view.

More Recent Work

More recently, I've been filling out even more of the algorithms into Alpaka, starting from the existing CUDA implementations. This includes the porting of some throughput examples, rather than the per-event exam

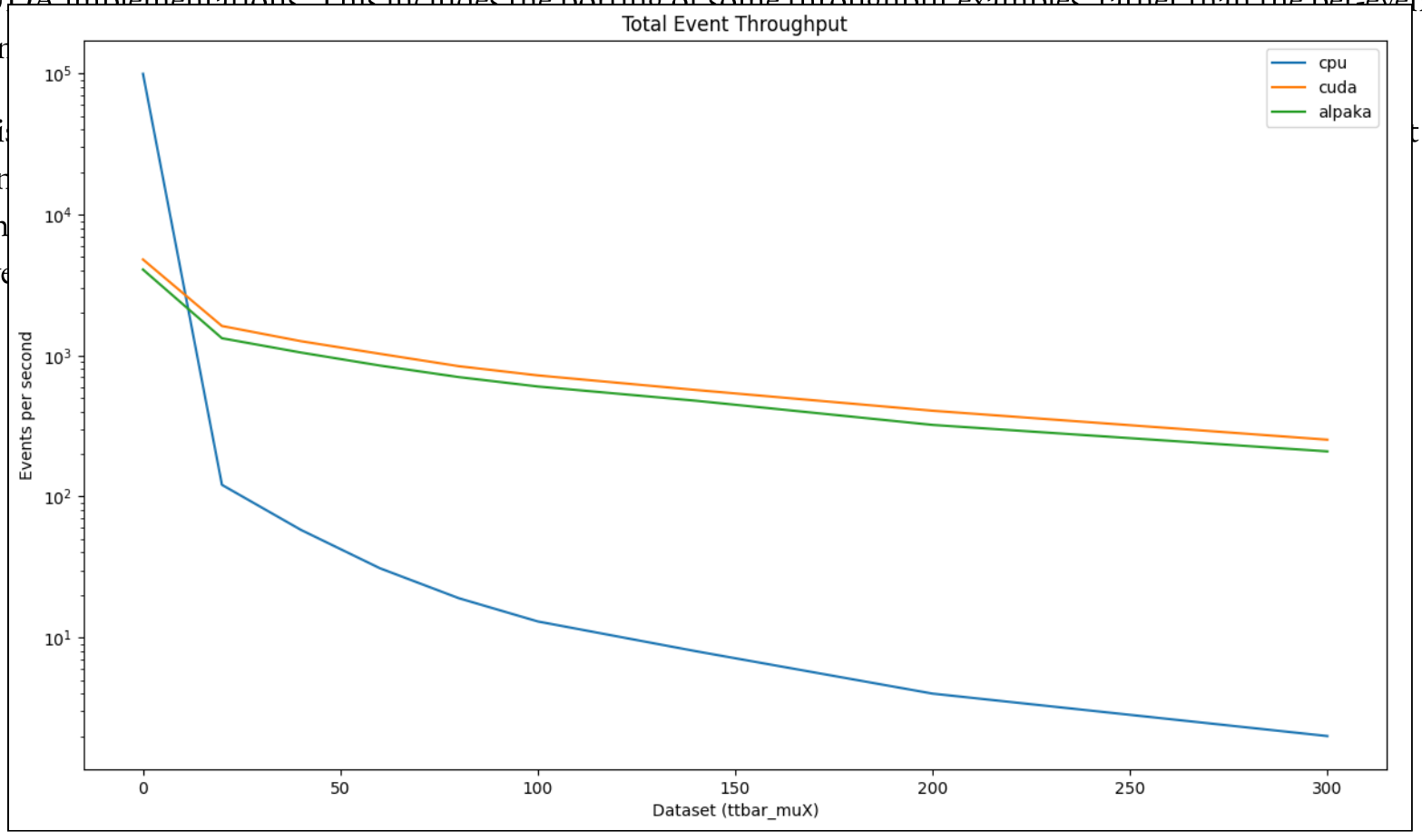
This
beir
rath
deve



More Recent Work

More recently, I've been filling out even more of the algorithms into Alpaka, starting from the existing CUDA implementations. This includes the porting of some throughput examples, rather than the per-event exam

This
beir
rath
deve



Developer Experience

Now that I'm slightly more familiar with alpaka, I thought perhaps it would be useful to give my *opinions* on how it is to develop with it, at least from the context of porting existing CUDA code to benefit from Alpaka:

Developer Experience

Now that I'm slightly more familiar with alpaka, I thought perhaps it would be useful to give my *opinions* on how it is to develop with it, at least from the context of porting existing CUDA code to benefit from Alpaka:

- Mostly, very positive! The parallelisation paradigm being the same as CUDA does make the process much easier, and should also be fairly intuitive to people who know CUDA (at least when considering GPU-based backends...).

Developer Experience

Now that I'm slightly more familiar with alpaka, I thought perhaps it would be useful to give my *opinions* on how it is to develop with it, at least from the context of porting existing CUDA code to benefit from Alpaka:

- Mostly, very positive! The parallelisation paradigm being the same as CUDA does make the process much easier, and should also be fairly intuitive to people who know CUDA (at least when considering GPU-based backends...).
- Documentation is mostly good. There is a few gaps, which may assume a touch too much C++ knowledge, but its broadly good. The few parts I struggled with were either out-of-date docs, or they missed some additional context.
  - As is common in lots of projects, the examples are mostly quite basic, so there is a bit of a jump between say the Alpaka examples, and use in real code.

Developer Experience

Now that I'm slightly more familiar with alpaka, I thought perhaps it would be useful to give my *opinions* on how it is to develop with it, at least from the context of porting existing CUDA code to benefit from Alpaka:

- Mostly, very positive! The parallelisation paradigm being the same as CUDA does make the process much easier, and should also be fairly intuitive to people who know CUDA (at least when considering GPU-based backends...).
- Documentation is mostly good. There is a few gaps, which may assume a touch too much C++ knowledge, but its broadly good. The few parts I struggled with were either out-of-date docs, or they missed some additional context.
  - As is common in lots of projects, the examples are mostly quite basic, so there is a bit of a jump between say the Alpaka examples, and use in real code.
- Perhaps an obvious pain-point...Alpaka uses lots of templates, so you can easily end up with 1000s of lines of errors for a tiny typo.

Developer Experience

Now that I'm slightly more familiar with alpaka, I thought perhaps it would be useful to give my *opinions* on how it is to develop with it, at least from the context of porting existing CUDA code to benefit from Alpaka:

- Mostly, very positive! The parallelisation paradigm being the same as CUDA does make the process much easier, and should also be fairly intuitive to people who know CUDA (at least when considering GPU-based backends...).
- Documentation is mostly good. There is a few gaps, which may assume a touch too much C++ knowledge, but its broadly good. The few parts I struggled with were either out-of-date docs, or they missed some additional context.
  - As is common in lots of projects, the examples are mostly quite basic, so there is a bit of a jump between say the Alpaka examples, and use in real code.
- Perhaps an obvious pain-point...Alpaka uses lots of templates, so you can easily end up with 1000s of lines of errors for a tiny typo.
- Debugging with Nsight and similar works fine, just as well as normal if given the correct compile time flags, though can have slightly awkward to read auto-generated names.

Developer Experience

Now that I'm slightly more familiar with alpaka, I thought perhaps it would be useful to give my *opinions* on how it is to develop with it, at least from the context of porting existing CUDA code to benefit from Alpaka:

- Mostly, very positive! The parallelisation paradigm being the same as CUDA does make the process much easier, and should also be fairly intuitive to people who know CUDA (at least when considering GPU-based backends...).
- Documentation is mostly good. There is a few gaps, which may assume a touch too much C++ knowledge, but its broadly good. The few parts I struggled with were either out-of-date docs, or they missed some additional context.
  - As is common in lots of projects, the examples are mostly quite basic, so there is a bit of a jump between say the Alpaka examples, and use in real code.
- Perhaps an obvious pain-point...Alpaka uses lots of templates, so you can easily end up with 1000s of lines of errors for a tiny typo.
- Debugging with Nsight and similar works fine, just as well as normal if given the correct compile time flags, though can have slightly awkward to read auto-generated names.
- Perhaps obvious, perhaps not, but just because your code base now supports multiple backends, doesn't mean it can support them effectively. For example, compiling to utilise a single CPU thread, compared to the single thread CPU implementation already in traccc....results in much worse performance.

Future Work

The PR for this set of work is open (#451), and I've gotten a bit further in my development branch, such that I can run (almost) the full set of algorithms now, excluding some that were added very recently.

Along side that, there is a few remaining bugs and things to look at:

- I haven't tested across a wide-range of devices yet. I've done a quick CPU test, but not thoroughly checked yet. I also would like to get access to somewhere I can test the HIP side of things!
- Similarly, since the CUDA implementation is 1-element-per-thread (for a lot of kernels at least), I'd need to expand the parallelisation a touch to work on threaded CPUs, to process more than 1 thing at once.
- There is still a performance delta between the CUDA and alpaka implementation, which is to be expected, but it would be interesting to look into it a bit more.

Conclusion

In Conclusion:

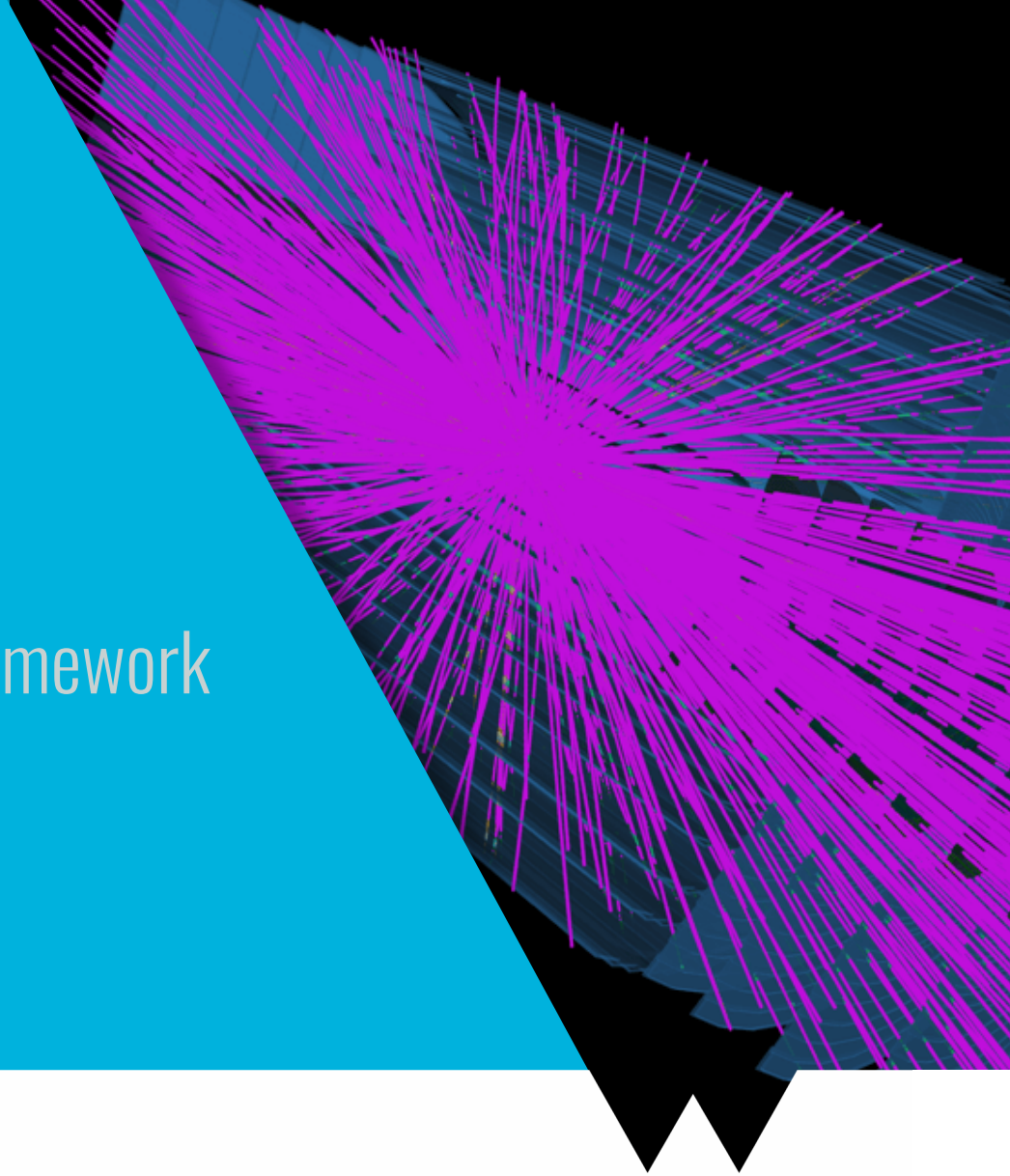
- traccc is a R&D effort as part of the ACTS project, working on exploiting GPUs and other accelerators to speed up tracking across a range of experiments.
- As part of that, many different acceleration abstraction libraries have been implemented, with alpaka being the newest.
- alpaka has good support already in HEP, and its parallelisation model make it a strong candidate for being the general purpose abstraction library.
- To that end, this talk outlines the next steps, porting more algorithms to alpaka.
- Decent performance has been achieved, even without extensive tuning or testing of the different parallelisation parameters or threading model.
- Finally, there is a clear path forward to how to more extensively test alpaka, to understand how it performs and how easy it is to implement the sort of operations we need in it.

traccc

Integrating the Alpaka framework

Ryan Cross

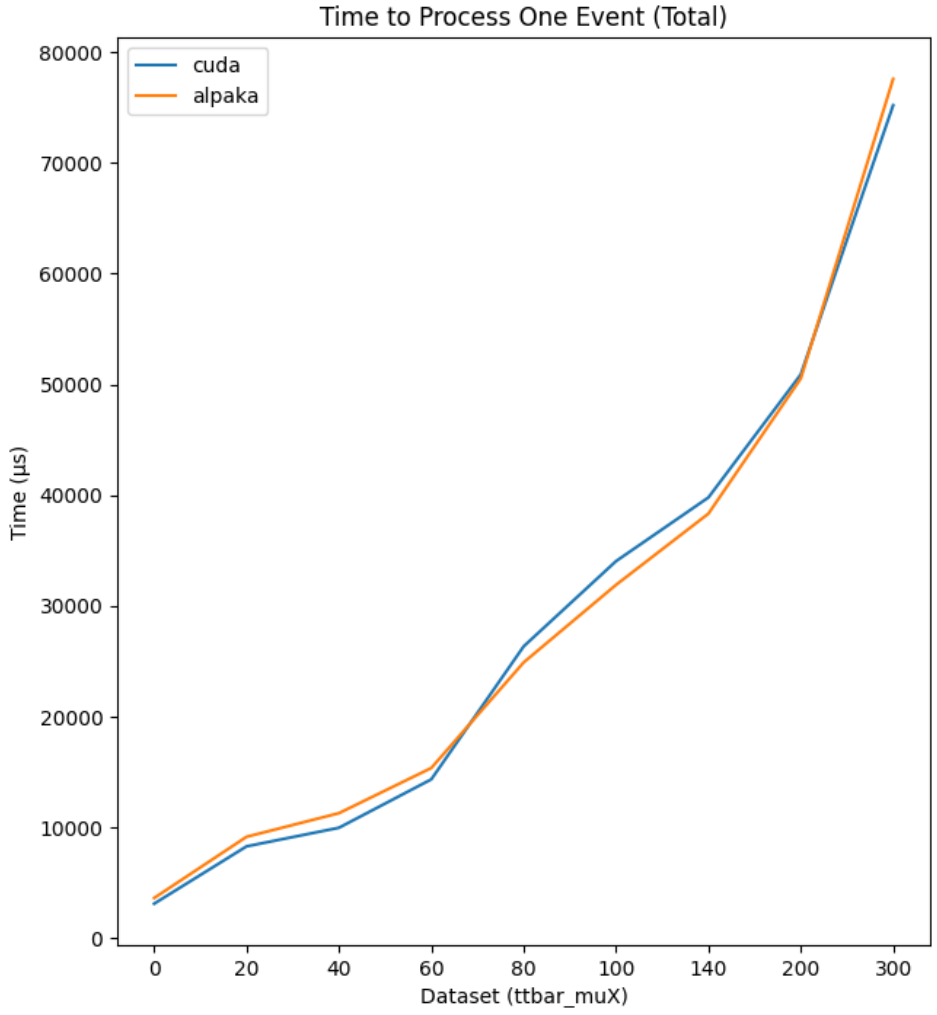
2023/11/22



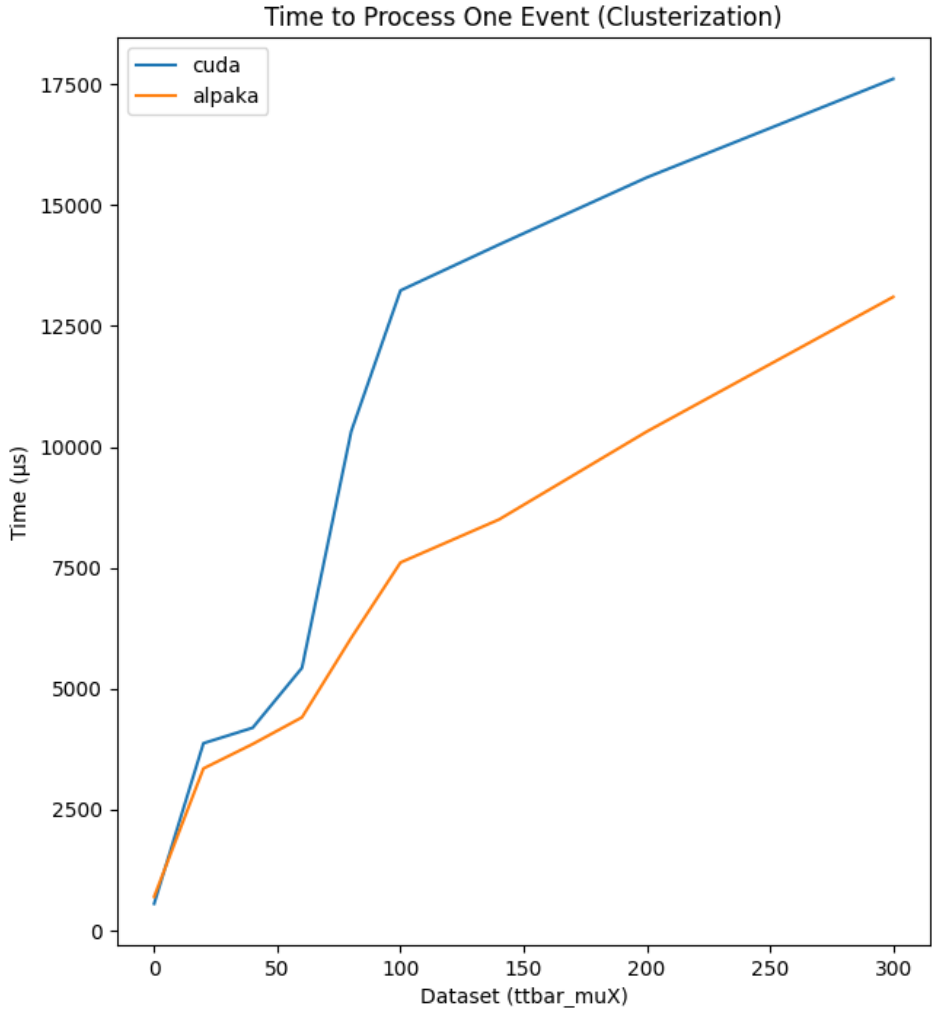
WARWICK
THE UNIVERSITY OF WARWICK

Backup Slides

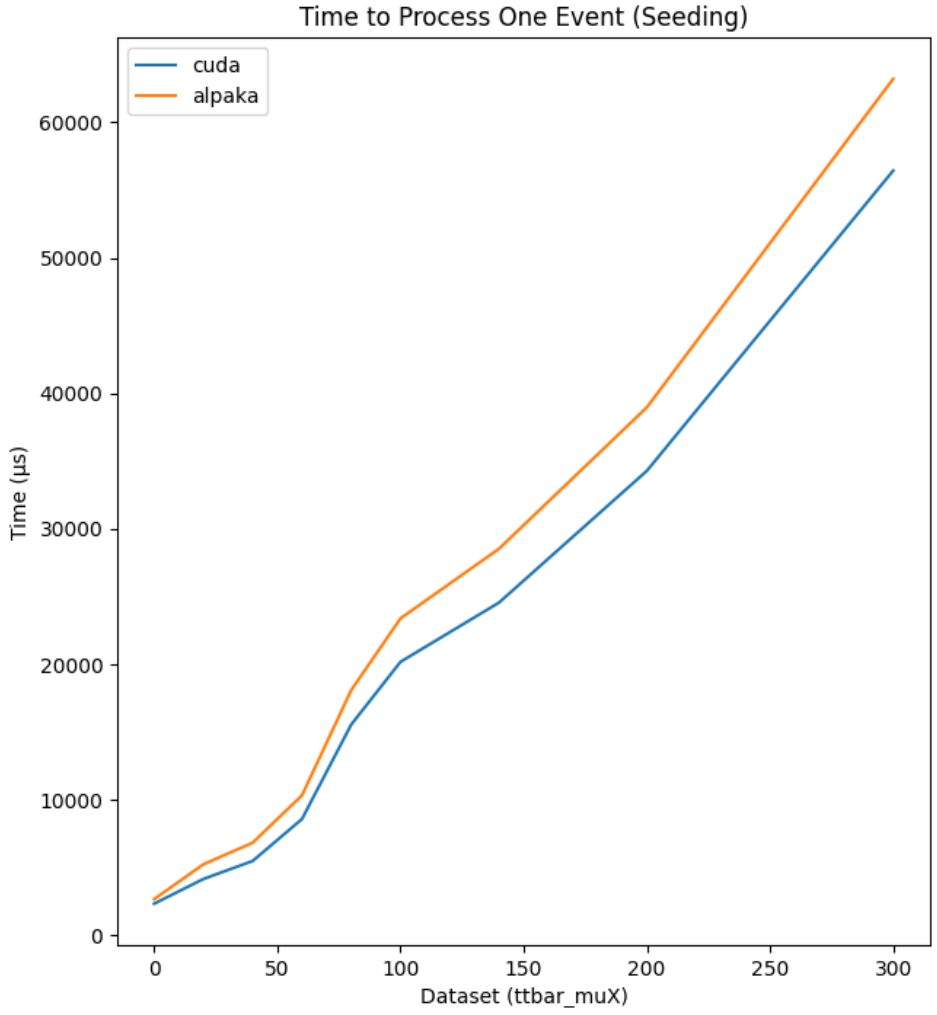
CUDA vs alpaka only



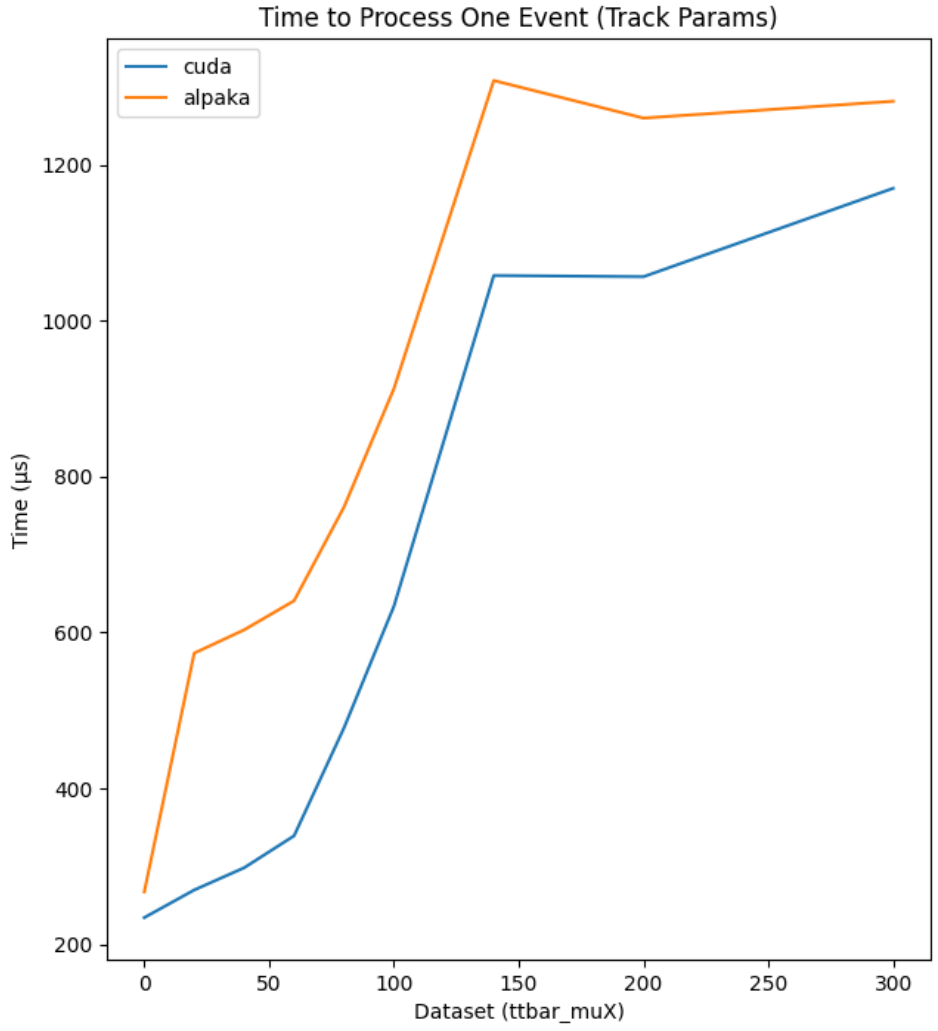
CUDA vs alpaka only



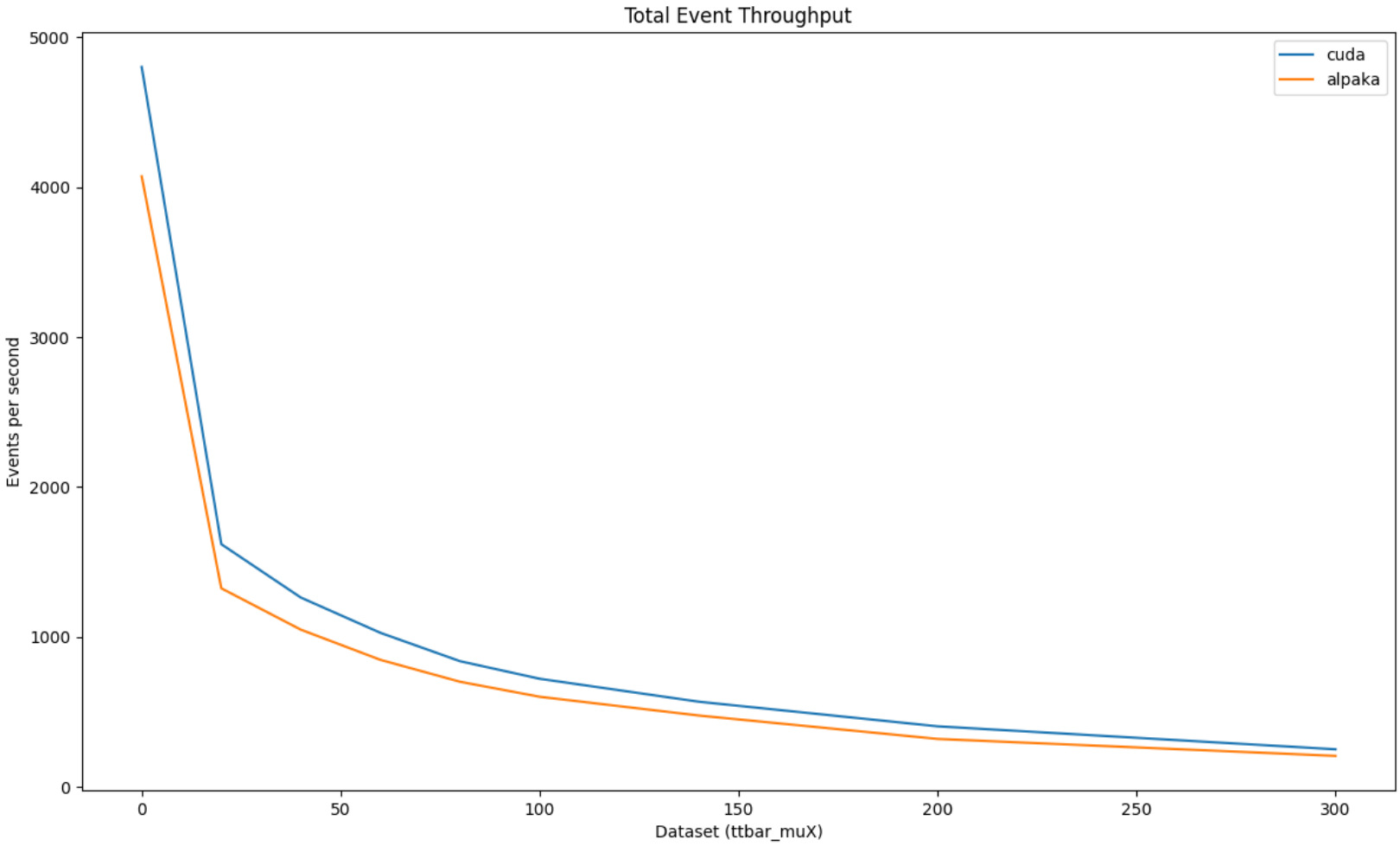
CUDA vs alpaka only



CUDA vs alpaka only



CUDA vs alpaka only



CUDA vs alpaka only