PORTABILITY FOR THE PATATRACK PIXEL TRACKS AND VERTICES RECONSTRUCTION

OCTOBER 2021

AUTHOR:

Abhinav Ramesh

SUPERVISORS:

Dr. Felice Pantaleo

Wahid Redjeb

PROJECT SPECIFICATION

As an Openlab Summer Student at CERN, my work involved the standalone Patatrack pixel track and vertex reconstruction repository (https://github.com/cms-patatrack/pixeltrack-standalone). My focus was on the Alpaka version of pixel tracking. Alpaka is a header-only C++ abstraction library for accelerator development supporting many accelerator back-ends.

In order to meet the goals of performance improvement and reduction of repeated memory allocations when processing many events as well as to mirror the CUDA version of pixel tracking, it was desired that a mechanism to reuse device and host memory blocks be incorporated.

This was accomplished by implementing a caching allocator each for the host and device in order to handle memory allocations and free-ups by managing memory blocks and caching them upon deallocation for future use. Smart pointers were provided to handle memory allocations in order to facilitate the usage of these allocators, which could also be disabled if necessary, at compile-time. The corresponding pull request can be found at https://github.com/cms-patatrack/pixeltrack-standalone/pull/248 .

ABSTRACT

The purpose of this report is to explain in detail the improvements to the Alpaka version of the Patatrack pixel track and vertex reconstruction repository in the form of caching allocators for allocating and reusing device and host memory as well as smart pointers as an interface for memory management. Furthermore, it’s aim is to also quantify these improvements.

The report begins with an introduction to the CMS silicon pixel detector within the LHC. It then discusses heterogeneous computing in the context of CMSSW briefly, before moving on to the Alpkaka library. After this, there is an overview of the pixel track and vertex reconstruction process. Finally, the caching allocators and the associated smart pointer interface is explained in detail along with the performance improvement results.

TABLE OF CONTENTS

CMS 05

HETEROGENEOUS COMPUTING 06

ALPAKA 06

PATATRACK PIXEL TRACK AND VERTEX RECONSTRUCTION 07

CACHING ALLOCATOR 09

- INTRODUCTION
- IMPLEMENTATION
- INTERFACE

RESULTS 13

CONCLUSION 13

REFERENCES 14

1. CMS

The Large Hadron Collider (LHC) is a circular collider, designed for proton-proton or heavy ion collisions. The LHC accelerates protons up to an energy of 7 TeV, producing an energy in the center of mass system of 13-14 TeV.

In the LHC, there are four points at which the beam collisions take place. At one of these points is placed the Compact Muon Solenoid (CMS) detector. It is responsible for observing new particles and Physics phenomena by “taking photos” of the collisions 40 million times a second. The particles produced from the collisions are detected, and then, their momenta and energies are measured. Figure 1 is an illustration of the CMS detector.

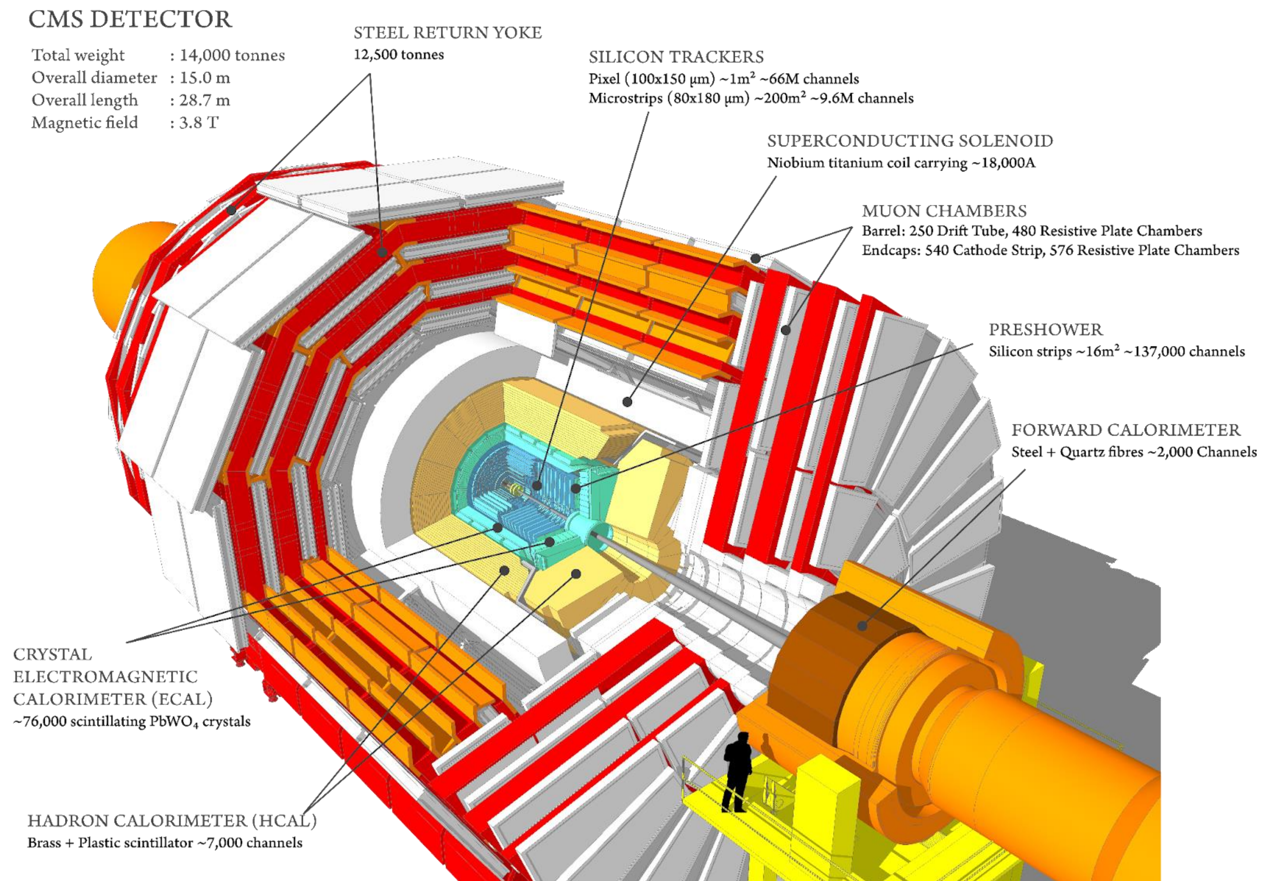

Figure 1: An illustration of the CMS detector

In order to process the data collected, the CMS has developed a software framework, called CMSSW. This allows us to perform simulations, High Level Trigger (HLT) reconstruction, offline reconstruction, and analysis, among other things. The independent chunks of data collected after each collision are called events. Data from each event is processed by so called CMSSW Modules. Each Module works on a specific reconstruction (pixel tracks reconstruction, calorimeter reconstruction, particle flow, etc). Modules can produce objects and put them in the event, making them available to be used by other modules.

2. HETEROGENEOUS COMPUTING

As part of the CMS phase-2 upgrade, there will be a lot of data being produced as well as an increase in luminosity (5x). The output of the L1 trigger will be increasing from 100 kHz to 750 kHz (7.5x). The High Level Trigger (HLT)'s output will also be increasing by 5-7x. There will be more complex detectors involved. There will also be an increase in pile-up of 4x (50 to 200). Moreover, reconstruction algorithms do not scale linearly with pile-up. Importantly, we wish to run better Physics algorithms that are more complex at the same cost. This motivates the need to look into heterogeneous computing and hardware accelerators.

Heterogeneous systems consist of different types of computational units such as multicore CPUs, GPUs and FPGAs. Hardware accelerators can perform specific functions more efficiently compared to software running on a general-purpose CPU. They have always been used for 3D graphics acceleration, compression, encryption and pattern recognition. Each of CPUs and GPUs have their strengths and weaknesses and the key lies in offloading the right kind of computationally intensive tasks to GPUs.

3. ALPAKA

There are many architectures and vendors today - NVIDIA GPUs (CUDA), ARM GPUs (HIP), OpenMP and TBB, to name some. Writing code mundanely for each back-end that's used would lead to code duplication. Changes will have to be made in multiple places, and this could lead to mistakes cropping up in several areas. Alpaka was created in order to address this issue, allowing users to write source code once, that can be executed on different architecture by compiling for each one of them, with performance close to the native ones.

Alpaka is a header-only C++ library that provides performance portability across accelerators by abstracting the underlying parallelism. It is platform independent, and can support the concurrent and cooperative use of multiple devices. A good number of accelerator back-ends are supported and there is no need to write any specialized code for any particular back-end. The abstraction used is similar to the CUDA grid-blocks-threads division strategy.

Memory allocations are managed using buffers. Buffers may be allocated on the host or device. Moreover, we can also execute copy operations from one buffer to another, irrespective of where they reside. Alpaka operations are synchronized using a queue. The following code snippet (Figure 2) illustrates buffer allocation:

```
using BufAcc = mem::buf::Buf<Acc, float, Dim, std::size_t>;   // Accelerator buffer type

auto const devAcc = pltf::getDevByIdx<Acc>(0u);              // create accelerator dev.

BufAcc accBuffer = mem::buf::alloc<float, std::size_t>(devAcc, extents);
```

Figure 2: Alpaka code snippet on buffer allocation

The code below (Figure 3) illustrates how to define a kernel and launch it. This is similar to how it's done in CUDA. In this case, we launch a number of threads equal to the number of points. For each point, we check whether it falls within a certain circle. We have to pass the accelerator as an argument to the kernel in order to give the kernel access to accelerator information like device, work division, etc.

```
struct PixelFinderKernel
{
    template <typename Acc>
    ALPAKA_FN_ACC void operator()(Acc const & acc, Points points, float r) const {

        uint32_t gridThreadIdx = idx::getIdx<Grid, Threads>(acc)[0];

        float x = points.x[gridThreadIdx];
        float y = points.y[gridThreadIdx];
        float d = math::sqrt(acc, x * x + y * y);

        bool isInside = (d <= r);

        points.inside[gridThreadIdx] = isInside;
    }
};
PixelFinderKernel pixelFinderKernel;
auto taskRunKernel = kernel::createTaskKernel<Acc>(workDiv, pixelFinderKernel,
                                                    pointsAcc, r);
queue::enqueue(queue, taskRunKernel);
```

Figure 3: Alpaka code snippets on kernel definition and launch

4. PATATRACK PIXEL TRACK AND VERTEX RECONSTRUCTION

The CMS silicon detector is designed for particle tracking. Its 65 million silicon pixels, which are arranged in layers of two-dimensional tiles, allow us to reconstruct the trajectories of particles. This is done using the three-dimensional picture formed using the pixels touched by the particles. These “hits” are caused by charges accumulated on the pixel surface due to the charged particles that pass through that pixel. An illustration of the silicon pixel detector is given below (Figure 4):

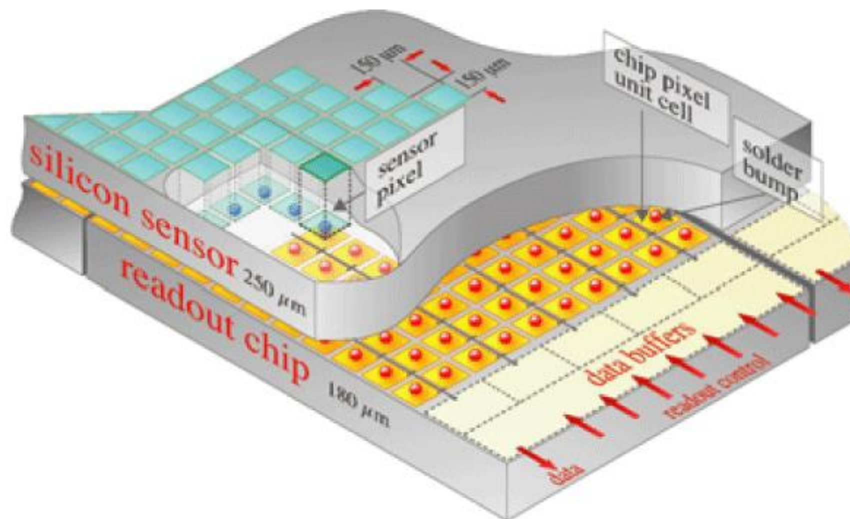

Figure 4: An illustration of the CMS silicon pixel detector

The steps involved in the reconstruction are as follows:

1. Digis, which represent a single pixel, and contain information about the collected charge as well as grid position, are created. Then, neighboring digis are grouped together to form clusters.
2. Hits belonging to adjacent pairs of layers are connected to form doublets. The doublets that share a common hit are tested for compatibility to form a triplet. These compatible doublets form a directed acyclic graph. Some of these doublets are marked as root doublets, from each of which a depth-first search is executed. This Cellular Automaton pattern recognition algorithm is used to form n-tuplets.
3. The Fishbone mechanism is used to resolve ambiguities due to several n-tuplets that could correspond to the same particle.
4. Broken Line Fit(a multiple scattering-aware fit) are performed over all n-tuplets, to produce the final pixel tracks. The final pixel tracks are eventually clusterized to produce the vertices.

Parallel algorithms were developed to perform the track reconstruction on GPUs, starting from the raw data from the CMS pixel detector. The structure of arrays(SoA) data structures used by the parallel algorithms are optimized for coalesced memory access on the GPU. Since data transfer between CPU and GPU is time consuming, the chain of modules that produce the final tracks and vertices from the raw data all run on the GPU. The flow-chart below (Figure 5) describes the reconstruction process:

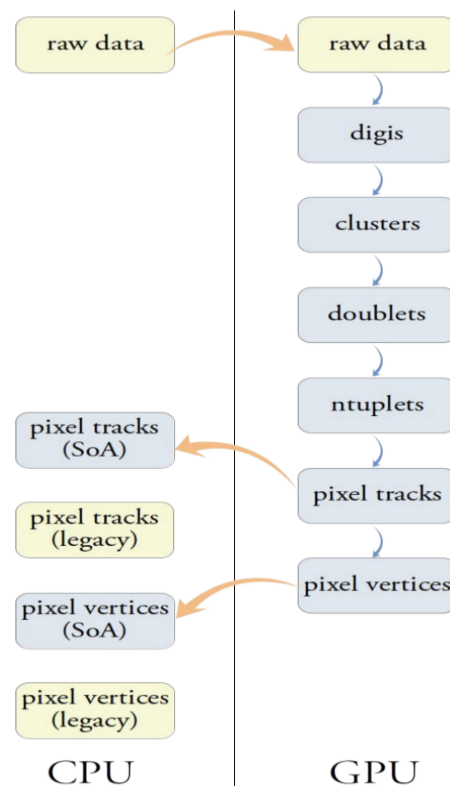

Figure 5: Pixel tracks and vertices reconstruction flow-chart

There is a standalone repository for the pixel track and vertex reconstruction (https://github.com/cms-patatrack/pixeltrack-standalone). Various performance portability solutions are explored using this repository, and Alpaka is being tried as well.

5. CACHING ALLOCATOR

a. INTRODUCTION

A major part of the reconstruction has been ported to Alpaka, but performance improvement is needed. It's not desirable to keep allocating memory repeatedly as we process the events. So, we wish to reuse memory. Moreover, the CUDA version makes use of caching allocators whose job is to cache blocks of memory allocations and keep track of them using data structures. Memory that's freed is reclaimed by these allocators for future allocations. My work involved porting these allocators to Alpaka (https://github.com/cms-patatrack/pixeltrack-standalone/pull/248).

There is a caching allocator for the host and one allocator for managing allocations on all devices corresponding to that particular accelerator back-end. Both the allocators are thread-safe. In order to make reuse of memory blocks better, it's better to quantize the size of the memory blocks. If we allocate exactly for every request, we would have to wait for a request of the same or a smaller size in order to reuse that block once it becomes available, and there would also be a lot of wastage. This is why we have a certain number of bins, and every memory block belongs to a certain bin and all blocks belonging to a particular bin have the same size.

The caching allocators are configured with the minimum bin, maximum bin, bin growth as well as the maximum bytes cached. The maximum bytes cached is equal to the maximum fraction of bytes cached times the smallest amount of free memory present on a device, in case there are multiple devices existing. We restrict the number of bins possible to maximum bin minus minimum bin plus one. The minimum bin will correspond to bytes equal to bin growth to the power of minimum bin, and the maximum bin will correspond to bytes equal to bin growth to the power of maximum bin, and so on for every other bin in between. For example, the default constructed allocator has bin growth equal to 8, minimum bin equal to 3, and maximum bin equal to 7. This delineates 5 bin sizes: 512B, 4KB, 32KB, 256KB, and 2MB.

b. IMPLEMENTATION

There is a block descriptor data structure that corresponds to every memory block. There is a data structure each for managing live allocations (blocks that are in use) and cached allocations (cached blocks that can be reused for future allocations). These are implemented as unordered multisets in C++. The device allocator block descriptor definition is as follows (Figure 6):

```
/**
 * Descriptor for device memory allocations
 */
struct BlockDescriptor {
  ALPAKA_ACCELERATOR_NAMESPACE::AlpakaDeviceBuf<std::byte> buf;  // Device buffer
  size_t bytes;                                                   // Size of allocation in bytes
  size_t bytesRequested;  // CMS: requested allocation size (for monitoring only)
  unsigned int bin;       // Bin enumeration

  // Constructor (suitable for searching maps for a block, given a device and bytes)
  BlockDescriptor(unsigned int block_bin,
                  size_t block_bytes,
                  size_t bytes_requested,
                  const ALPAKA_ACCELERATOR_NAMESPACE::DevAcc1& device)
      : buf{alpaka::allocBuf<std::byte, alpaka_common::Idx>(device, 0u)},
        bytes{block_bytes},
        bytesRequested{bytes_requested},  // CMS
        bin{block_bin} {}

  // Constructor (suitable for searching maps for a specific block, given a device buffer)
  BlockDescriptor(ALPAKA_ACCELERATOR_NAMESPACE::AlpakaDeviceBuf<std::byte> buffer)
      : buf{std::move(buffer)},
        bytes{0},
        bytesRequested{0},  // CMS
        bin{INVALID_BIN} {}
};
```

Figure 6: Definition of the memory block descriptor

The one for the host is similar, with the difference being that the buffer involved is a host buffer.

The live device blocks are compared and hashed using the device identifier and native device pointers of the memory blocks involved, while the cached device blocks are compared and hashed using the device identifier and the size in bytes. The ones for the host are similar, except that there is no device identifier involved. The associated code is as follows (Figure 8):

```
struct BlockHashByBytes {
  size_t operator()(const BlockDescriptor& descriptor) const {
    size_t h1 = std::hash<int>{}(getIdxOfDev(alpaka::getDev(descriptor.buf)));
    size_t h2 = std::hash<size_t>{}(descriptor.bytes);
    return h1 ^ (h2 << 1);
  }
};

struct BlockEqualByBytes {
  bool operator()(const BlockDescriptor& a, const BlockDescriptor& b) const {
    return (getIdxOfDev(alpaka::getDev(a.buf)) == getIdxOfDev(alpaka::getDev(b.buf)) && a.bytes == b.bytes);
  }
};

struct BlockHashByPtr {
  size_t operator()(const BlockDescriptor& descriptor) const {
    size_t h1 = std::hash<int>{}(getIdxOfDev(alpaka::getDev(descriptor.buf)));
    size_t h2 = std::hash<const std::byte*>{}(alpaka::getPtrNative(descriptor.buf));
    return h1 ^ (h2 << 1);
  }
};

struct BlockEqualByPtr {
  bool operator()(const BlockDescriptor& a, const BlockDescriptor& b) const {
    return (getIdxOfDev(alpaka::getDev(a.buf)) == getIdxOfDev(alpaka::getDev(b.buf)) &&
            alpaka::getPtrNative(a.buf) == alpaka::getPtrNative(b.buf));
  }
};
```

Figure 7: Functors for hashing and comparing blocks

```
/// Set type for cached blocks (hashed by size)
using CachedBlocks = std::unordered_multiset<BlockDescriptor, BlockHashByBytes, BlockEqualByBytes>;

/// Set type for live blocks (hashed by ptr)
using BusyBlocks = std::unordered_multiset<BlockDescriptor, BlockHashByPtr, BlockEqualByPtr>;
```

Figure 8: Type definitions for the cached and live blocks data structures

The allocation method of the caching allocator takes the number of bytes as a parameter and returns a host or device buffer corresponding to the C++ standard byte (std::byte) data type. For the caching device allocator, the device on which the block is to be allocated is also taken as a parameter. The allocation proceeds as follows:

1. First of all, the nearest power of bin growth that is greater than or equal to the bytes requested is found out. We create a new block descriptor and set its bin to this and the descriptor's bytes to the bytes corresponding to this bin.
2. If this bin is greater than the maximum bin, we mark the descriptor's bin as invalid and set the bytes of the block descriptor to the bytes requested, and we go to step 5. Else, we go to the next step.
3. First, we check if the bin is smaller than the minimum bin, in which case we set the bin and bytes of the block descriptor to the minimum bin and the corresponding bytes.
4. Then, we try to find a cached block on the same device corresponding to the same bin. If this is found, we remove this block from the cached blocks and insert it into the live blocks, and we return the corresponding buffer. Else, we go to the next step.

5. We allocate a new buffer with bytes equal to the block descriptor's bytes. We insert the block into the live blocks, and return this buffer.

The method for freeing up a block takes in the corresponding buffer as a parameter, and the freeing proceeds as follows:

1. First, we try to search for a live block that contains the same buffer as the parameter. If it's found, we remove it from the live blocks and proceed to the next step.
2. If the corresponding bin is not invalid and we won't exceed the maximum cached threshold if we keep this allocation, we insert this into the cached blocks.

The illustration below (Figure 9) depicts allocation and free-up. First, an allocation request comes by for 20 KB. This is assigned to the 32 KB bin. Then, a 32 KB block is allocated and marked as a live block and then returned. Then, it is marked as a cached block during deallocation/free-up. After this, another allocation request for 24 KB comes by. This is also assigned to the 32 KB bin by finding the nearest power. SInce the earlier 32 KB block is present as a cached block, it's reused and returned after being marked as a live block.

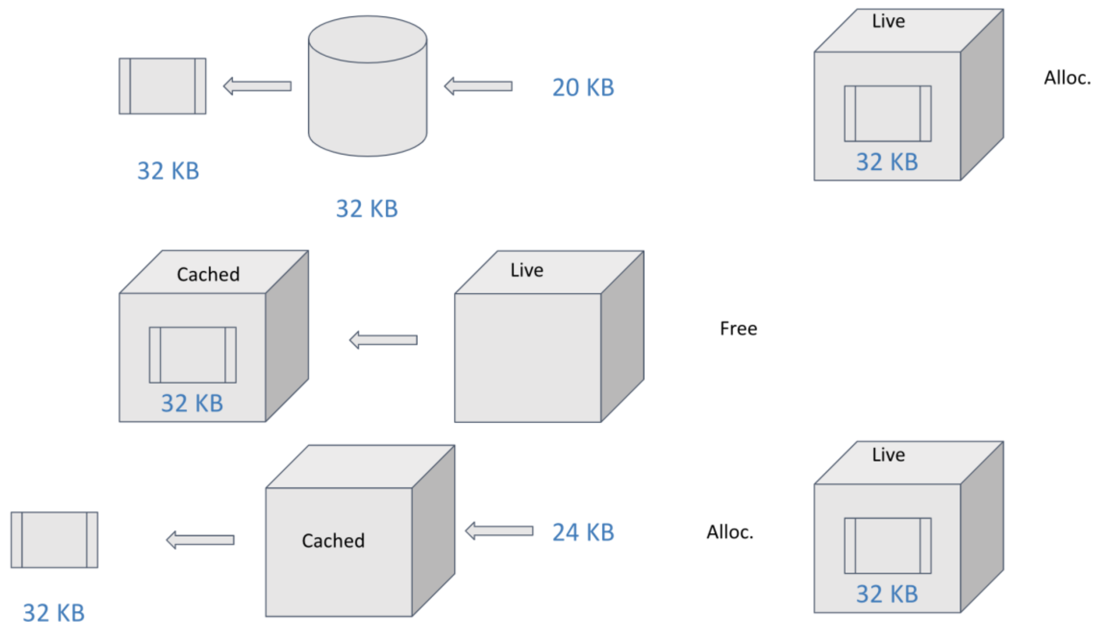

Figure 9: An illustration showcasing memory allocation, free-up, caching, and subsequent allocation

At compile-time, we can choose the allocation policy. This can be one of three types: caching, synchronous, and asynchronous. The caching allocator is used for the caching policy type, whereas the allocation is performed normally without using the allocator for the other two types. The only difference between the other two types is that the allocated buffer is prepared for asynchronous copy operations for the asynchronous policy type.

c. INTERFACE

Allocation is managed in the form of unique pointers provided for host and device. A function each for host and device is provided for making this pointer, and is templated on the data type and takes a single parameter, which is the extent of the allocation. The function makes the pointer as follows:

If the policy is a caching policy type, we find the number of bytes corresponding to this allocation and allocate a buffer using the caching allocator's allocation method. Else, we normally allocate a buffer using the data type and extent. If the policy is of type asynchronous, we also prepare the buffer for asynchronous copies. Then, we create an instance of the unique pointer by obtaining the underlying native pointer from this buffer. We also pass in a custom deleter to this pointer, which stores this buffer. At destruction time, if the policy is a caching policy type, this deleter checks whether the underlying raw pointer is not null and if true, calls the freeing up function of the allocator with the stored buffer.

The process described above is similar for host and device. The only exception is that for the device pointer, if the back-end is not a GPU back-end, in which case the host and device are same, the device pointer is defined to be the same as the host pointer and the device pointer making function simply delegates to the host pointer making function.

6. RESULTS

Measurements were made on a machine having an Intel(R) Xeon(R) Silver 4114 CPU @ 2.20 GHz and an NVIDIA Tesla T4 GPU. The throughput when processing 5000 events was measured by taking the average of 5 runs for 3 back-ends: CUDA, Serial and TBB with and without the caching allocator(asynchronous). The results are summarized in the table below:

	Without allocator	With allocator
CUDA	205.328 events/s	519.500 events/s
TBB	9.453 events/s	10.056 events/s
Serial	21.102 events/s	21.749 events/s

We can see a phenomenal improvement of 2.5x in the throughput for CUDA. This is because, we are gaining by minimizing the number of CUDA API calls for allocating device memory and pinned host memory. The gains for TBB and Serial are small. This is because there is no external device for these cases and everything takes place on the host. Despite that, there is still some improvement because we are minimizing repeated allocations by reusing memory blocks.

7. CONCLUSION

We had set out with a goal to increase event processing throughput by minimizing device and host allocations for the Alpaka version of the Patatrack pixel tracks and vertices reconstruction repository. It was agreed that the way to do this was to not simply keep allocating chunks of memory when each request came by, but to somehow come up with an efficient scheme that will allocate memory blocks of the right size and cache these blocks when they are no longer needed and use them again when a request of the same or smaller size comes up in the future.

The key to solving this problem lies in not allocating exactly for every request, but rather denominating a number of bins and allocating with respect to these bin sizes, so as to maximize reuse when that block is returned. The management of these live and cached blocks involving memory allocations and free-ups was up to the caching allocators, which were implemented for the host and device. These allocators, which had already been implemented for the CUDA version of the repository, were efficiently ported to Alpaka. Then, these allocators were exposed indirectly through a smart pointer interface, with the user being given the choice to disable them, if necessary.

Then, experiments were conducted to see if the introduction of these caching allocators led to performance improvements over the plain old non-caching allocation scheme for three back-ends: CUDA, Serial and

TBB. The results demonstrated an excellent increase in throughput for CUDA (2.5x), and fair improvements for the other two back-ends, thus proving that the objectives of the project were realized.

8. REFERENCES

- https://home.cern/science/accelerators/large-hadron-collider
- https://home.cern/science/experiments/cms
- https://cms.cern/detector/identifying-tracks/silicon-pixels
- https://github.com/alpaka-group/alpaka
- https://arxiv.org/abs/2008.13461