Performance optimization of the online data processing software of CERN's LHCb experiment

Thesis report

Péter Kardos

2018-2019

1 Abstract

The LHCb experiment at the high energy physics research institute CERN conducts research to answer why the universe is made up of matter as opposed to anti-matter. Like any modern experiment, LHCb relies on a complex software infrastructure to collect and analyze the data generated from particle collisions. To filter out unimportant data before writing it to permanent storage, particle collision events have to be processed in real-time which requires enormous computing power. This paper focuses on performance optimization of parts of the software to make it performant in multi-threaded environments with modern, superscalar processors. Vectorization, memory management and cache efficient algorithms are the primary methods to achieve a significant speedup. Improvements by this work combined with further software optimziations and hardware upgrade allows the infrastructure to handle 30 times the throughput in order to maximize the physics research potential of the experiment.

2 Introduction

describe the problem in detail
specific to my thesis:
environment:
- CERN's goals/activity
- CERN's hardware infrastructure (accelerators, experiments)
- LHCb's hardware infrastructure
- LHCb's software reconstruction system
problem:
- event rate from detector
- slow trigger → loss of physics (ACTUAL PROBLEM)
- by optimizing individual algorithms (in this thesis)

2.1 About CERN

CERN (European Organization for Nuclear Research) is an international high energy experimental physics research organization situated near Geneva, on the Franco-Swiss border. CERN is host to the world's largest particle accelerator and numerous experiments which aim to provide a better understanding of the fundamental laws of the universe. [1]

2.2 What are particle accelerators

2.2.1 Idea and purpose of accelerators

Particle accelerators accelate charged particles to high velocities using magnetic and electric fields. The particles are made to collide with a stationary target or each other (particles of opposing directions having a frontal collision), which results in a shower of new-born particles flying away from the collision point. In high-energy collisions, exotic particles that are normally not seen are born, and their properties can be examined.

2.2.2 Theory of operation

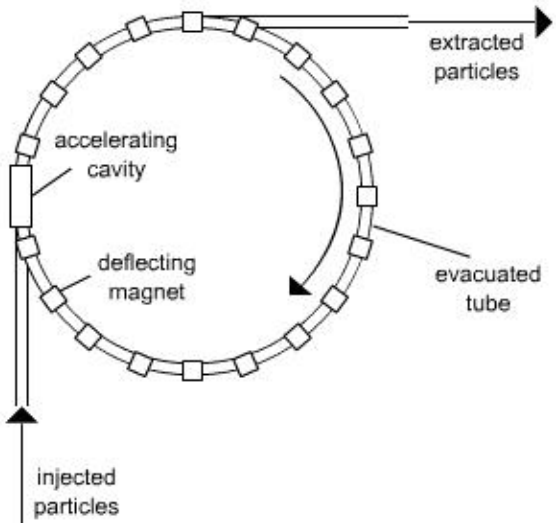


Figure 1: Simplified schematic of a circular particle accelerator with the crucial functional parts labeled.

Particle accelerators can be circular or linear. Figure 1 depicts a circular accelerator, which consists of the following parts:

- Evacuated tube (beam pipe): a closed, circular tube in which the particles can travel. To avoid the particles colliding with air particles, it is strictly under vacuum.
- Particle source: injects the particles into the beam pipe. For proton accelerators, a bottle of hydrogen serves as the source. The hydrogen atoms are ionized, and then linearly accelerated by an electric field before entering the circular beam pipe.
- Accelerating cavity: uses oscillating electromagnetic fields which are timed to provide a push to the charged particles via electric force, accelerating them.
- Deflecting magnet: using the Lorentz-force, a strong magnetic field steers the particles inwards to the center of the circle, keeping it on the circular trajectory.
- Extraction pipe: once the particles are fast enough, they are extracted from the circular beam pipe to collide with a stationary target.

The particle is at first injected to the beam pipe at a low energy. Thanks to the deflecting magnets, it keeps revolving in the beam pipe for thousand of revolutions. Each turn, particles get boosted by the accelerating cavities as they pass by, increasing their energy. When particles reach their maximum energy, which is determined by the construction and size of the accelerator, they are extracted from the beam pipe to collide with a stationary target.

It is not individual particles that revolve around the accelerator, rather, it's a swarm of millions of particles distributed throughout the entire circle. The distribution of the

particles is not uniform. The particles group into *bunches* which are equally spaced along the circle.

In addition to deflecting magnets, accelerators also employ *focusing magnets*. Their purpose is to squeeze the bunches in the directions perpendicular to the particles' velocity, resulting in the beam having a smaller cross-section. Without focusing magnets, the bunches would slowly disintegrate and the particles would hit the wall of the beam pipe.

Linear accelerators contain the same functional elements as circular ones, but they don't need deflecting magnets since the particles travel in a straight path.

2.2.3 Particle colliders

Colliders have two beam pipes right next to each other in the circular tunnel. The two beams circle in the opposing directions. At some specific points, the beam pipes are crossed, and the intersecting beams' bunches collide with each other.

2.3 The accelerator complex

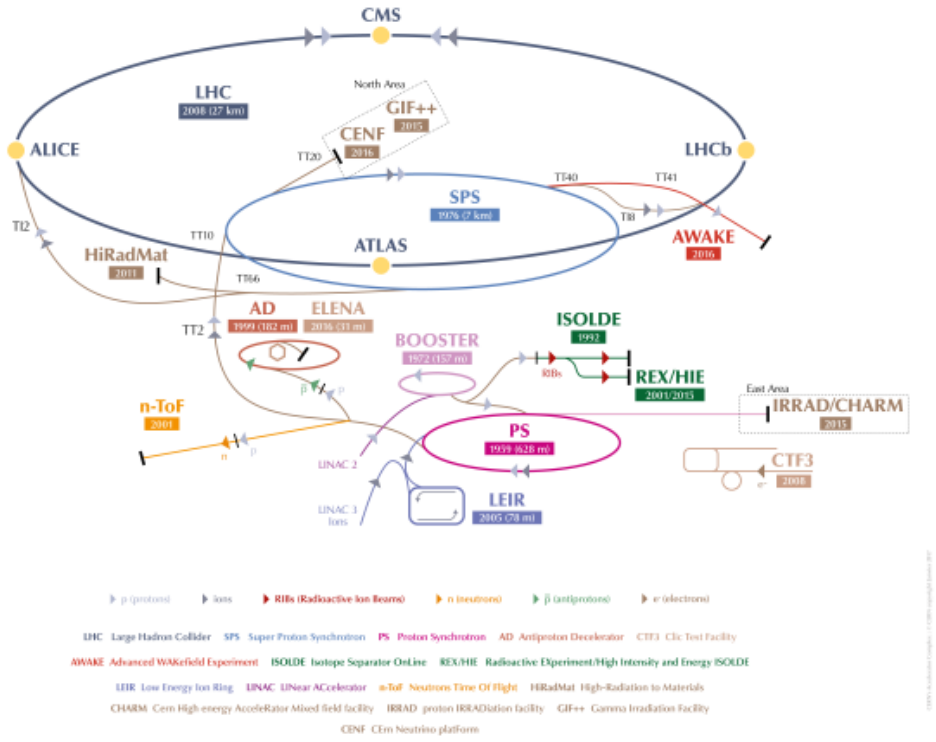


Figure 2: Schematic view of CERN's particle accelerators and experiments. LHC is shown on top by the largest circle. The four main experiments, CMS, ALICE, ATLAS and LHCb are marked with yellow dots along the LHC's circle.[2]

As seen on 2, CERN hosts a complex system of particle accelerators. The LHC, shown in dark blue, is the subject of this thesis project. The LHC is a particle collider, the energy of individual particles reaches about 7 TeVs at maximum. The particle collisions occur at 4 points, marked CMS, ALICE, LHCb and ATLAS. At these points, particle detectors are installed to analyze the collisions. Detectors track the path of the particles and measure their properties such as momentum, charge and mass. Using these properties, particles can be identified. The raw data provided by the detectors is processed by software.

2.4 LHCb experiment's detector

2.4.1 Construction of the detector

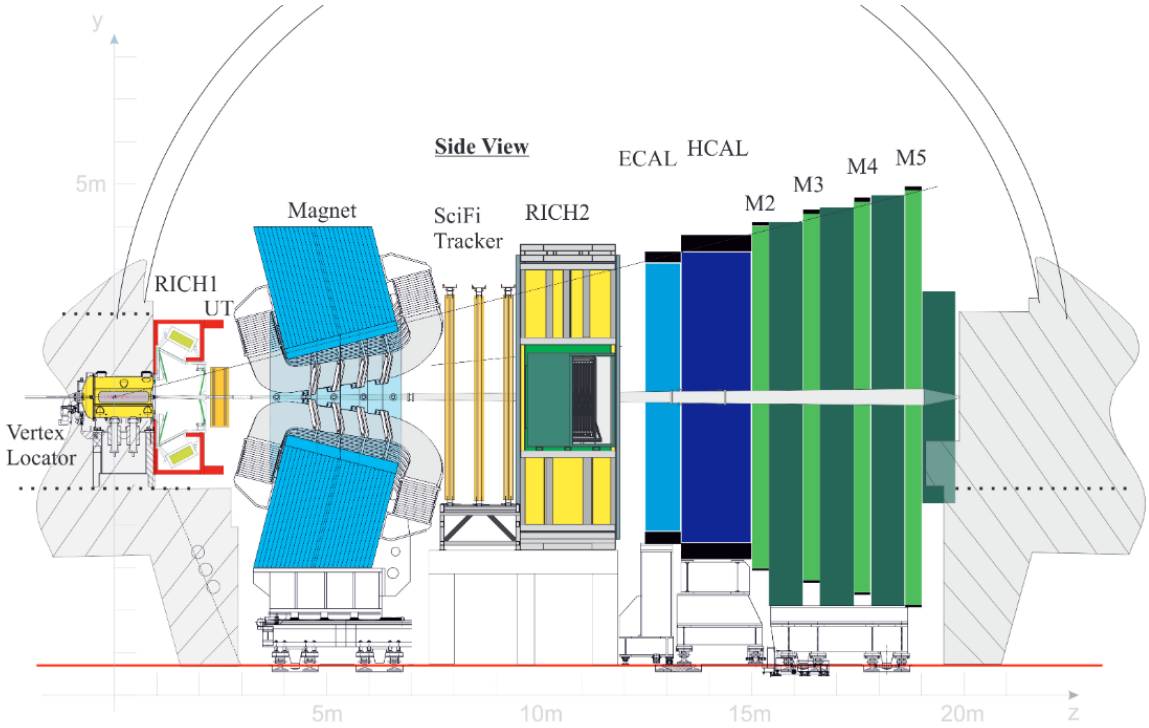


Figure 3: Side view of the LHCb detector.

Figure 3 shows the LHCb detector from the side, the two beams of the LHC go in the horizontal direction on the drawing, through the middle of the detector. The detector is both vertically and horizontally symmetric to the beampipe.

As seen on the labels, the detector consists of multiple layers of sub-detectors. Each layer has a hole in the center to let the beam pipes through. The two particle beams cross each other inside the Vertex Locator (VELO, at the right in yellow). As opposed to most other detectors, this one only analyzes the products of the collision in a narrow cone away from the VELO. The parts from RICH to M5 could be mirrored around the VELO to have two cones that touch each other by the tip, however it is not done for financial reasons.

The goal of the LHCb detector is the same as for all other detectors: reconstruct the paths and types of the particles. Even though full reconstruction uses all sub-detectors,

we are only interested in partial reconstruction that can be done in real-time. This only involves the VELO, the UT (in orange, left of VELO) and the FT (SciFi Tracker, in the middle in orange).

2.4.2 Coordinate system

The LHCb detector's coordinate system is used for calculations. The Z axis points down the beamline, the Y axis points upwards and the X axis to the left (as seen when looking down the Z axis). The LHC itself and civil engineering works use different labeling or alignment for the their coordinate axes.

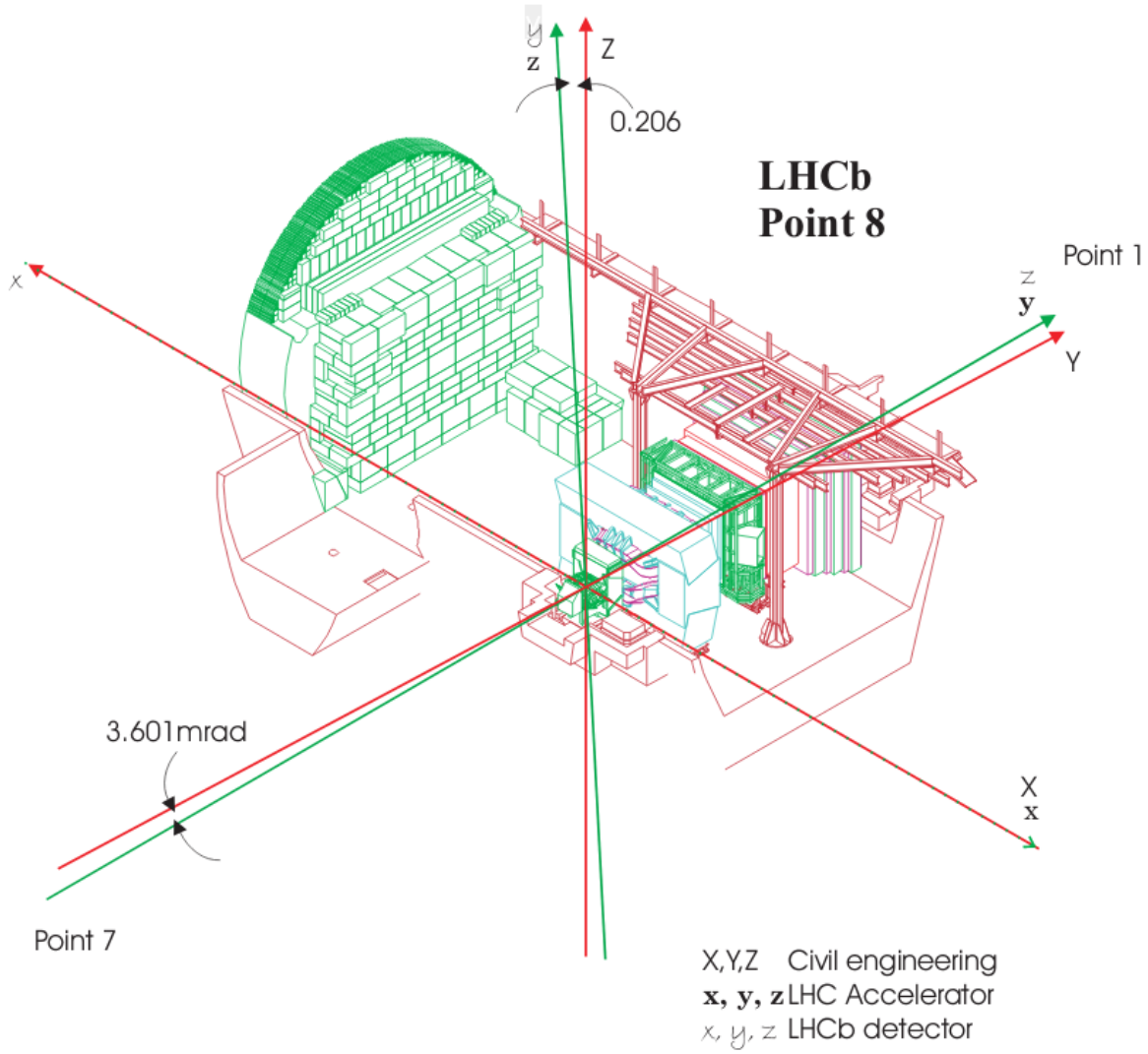


Figure 4: The different coordinate systems used in the LHCb cavern.

2.4.3 Operating principles

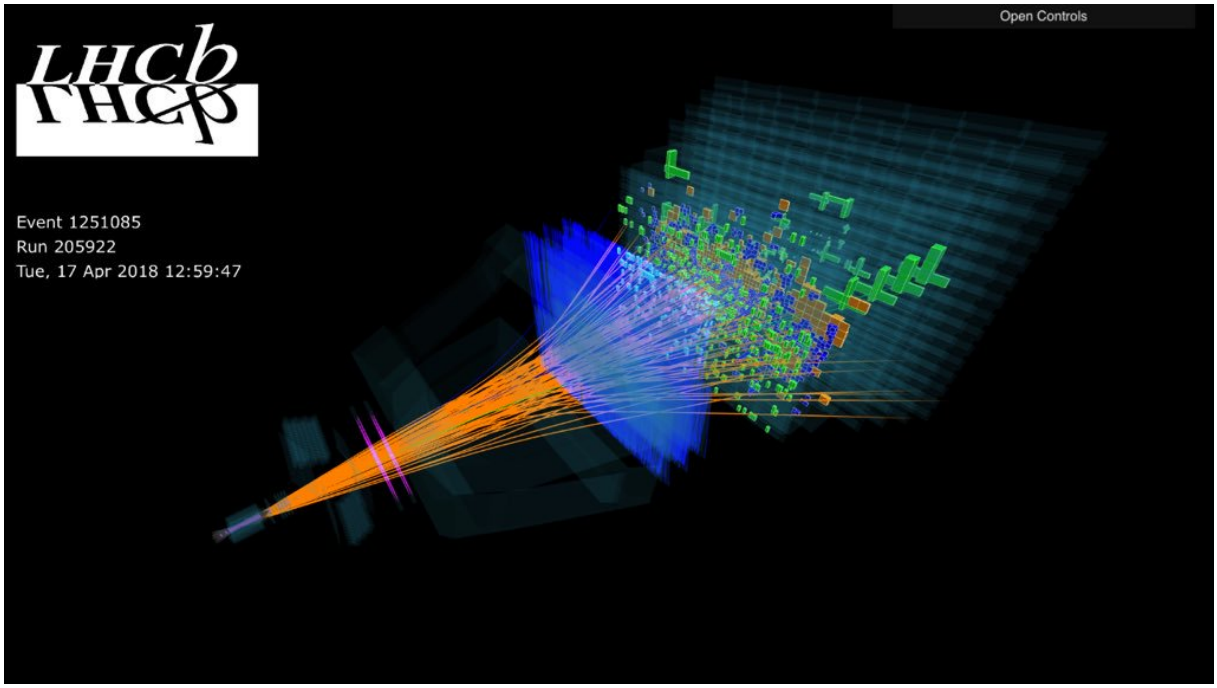


Figure 5: Particles created in a real collision and their interaction with the detector.

Figure 5 shows one particle collision event's results. The particles that were born in the collision are shown by the orange lines. The three aforementioned detectors, the VELO, UT and FT, can be identified by the origin of the orange particles, the pink cloud of lines and the bright blue cloud lines, respectively. The green and yellow cubic illustrations on the far-end of the detector belong to other sub-detectors, and are out of the scope of this paper.

The VELO is a small detector, measures less than a meter in length. It is a silicon pixel detector, which looks much like a modern CCD camera: a rectangular array of pixels which detect light (or in this case, particles) that hit it. The difference is that particle detectors don't absorb the particle, rather, it passes through largely undisturbed. Additionally, the VELO consists of 26 such CCD-like rectangles, instead of just one.

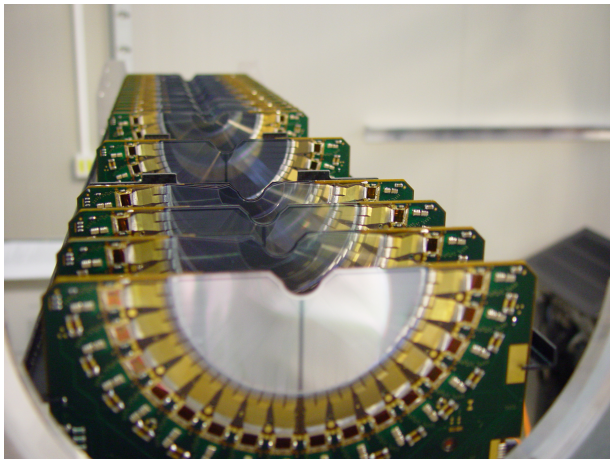


Figure 6: The 26 layers of the VELO. The pixels reside on the silver area, the PCB around contains the reading electronics, and the wedge at the top-center of the silver area is where the beam passes through.

The UT is a silicon strip detector which consist long fibers which act much like a pixel as they also signal if a particle passes through them at any point along their length. A fiber of the UT is generally 10 centimeters long, and has a width of only 192 micrometers. This means that while on one axis, the particles position can be told with great accuracy, on the other axis the uncertainty if 10 centimeters. The entire detector measures about 1.7 meters in width and 1.4 meter in height.

Similarly to the UT, the FT is also a silicon strip detector, but the length of its fibers is 2.5 meters. This means the 5 meter tall detector needs only two fibers to cover the full height.

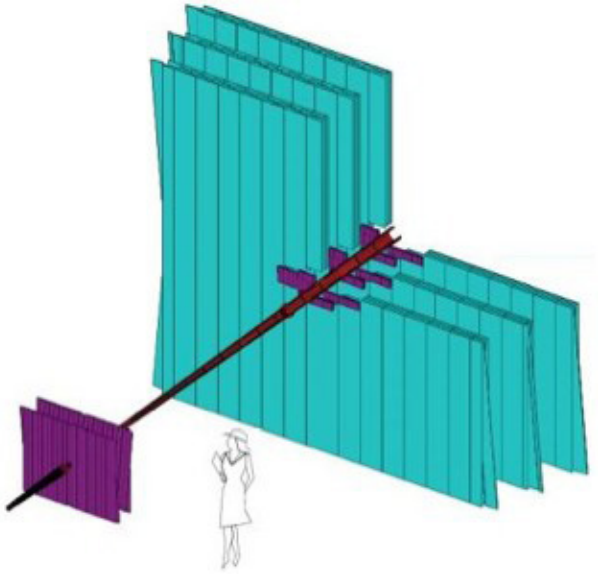


Figure 7: The UT in purple and the FT in blue, with a person next to them to illustrate the scale. The silicon strips are aligned vertically, leading to a good horizontal resolution but a poor vertical resolution.

In between the UT and the FT, a strong magnet is placed which bends the particles on the horizontal axis.

When a collision occurs, the particles follow a straight path and they are recorded passing through the pixels of the VELO. Initial particle trajectories can be reconstructed by finding hits in the VELO that align to form a straight line. These paths are then linearly extrapolated through the UT, and some of the silicon strips that were lit up by the particles are assigned to the initial trajectories acquired from the VELO. Inside the UT, particles paths are mostly straight, but they already experience a slight bending because of the magnet. From the amount of bending, the particle's momentum can be estimated. As the charged particles pass through the magnet, their trajectory bends, however, the amount of bending is a function of the particle's momentum. With a good momentum estimate, a guess can be given as to where the particle would hit the FT. In the suspected region, the silicon strips of the FT are searched and if the corresponding fibers are found, they are assigned to the particle.

At the end of the process, all the pixels or fibers that were touched by a particle are known, which makes it possible to know the exact path of the particle. The amount of bending from the UT to the FT allows the calculation of the momentum, which, when paired with information from other detectors, such as energy and velocity, makes it possible to identify a particle. (Identification means knowing the name of the particle, such as electron or muon.)

2.5 Events and triggering

2.5.1 Collision events

The LHC has 2808 bunches of protons in the accelerator at a time, for both beams. We refer to the collision of two particular bunches as an *event*. Events are completely independent, that is, a bunch-bunch collision only happens after the previous collision's products have been analyzed and flushed from the detector. The time between two subsequent bunches is 25 nanoseconds, which would set the rate of collisions at 40 million per second. In reality, there are some gaps between two adjacent bunches that are a lot longer than 25 nanoseconds. Due to these large gaps, the average rate of collisions is about 30 MHz, but a 40 MHz rate of processing is necessary to go without buffering.

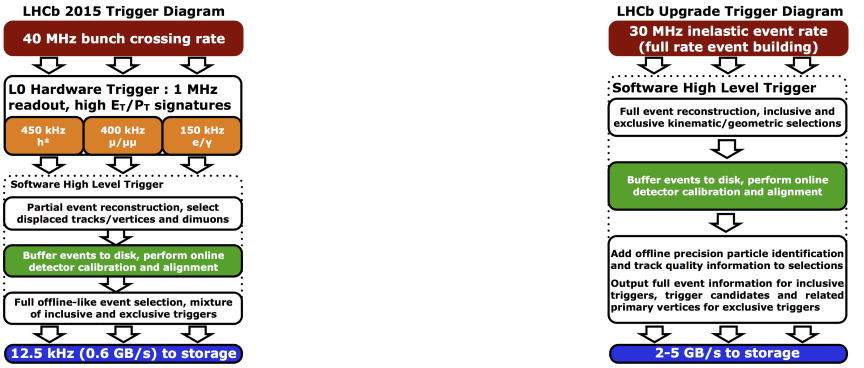
2.5.2 Real-time reconstruction and triggering

Most of the 30 million events that occur every second are absolutely uninteresting, with no exotic particles of interest being created. On the other hand, each event amounts to a significant amount of data which is a challenge to store. To reduce the amount of data written to permanent storage, each event is processed in real-time to determine weather it is interesting or not. Uninteresting events are simply dropped, the interesting ones are stored long-term for offline analysis. The act of deciding if an event has to be stored is called *triggering*. The real-time processing of events is a simplified method that relies on the VELO, UT and FT detectors as described previously.

2.6 The 2019/20 upgrade of LHCb

The LHC shuts down for maintenance for two years from the end of 2018. The LHCb detector also undergoes maintenance and upgrade during that time, where the VELO is upgraded and the old TT is entirely replaced by the UT.

The pre-upgrade detector does the real-time event processing in two stages. The first stage employs FPGAs to do a preselection, which cuts down the 30 million events per second to roughly 1 million per second. The second stage uses a large farm of CPUs to do finer reconstruction and final trigger decision. During the upgrade, however, the FPGAs will be retired, and the CPU farm has to take the whole load of 30 million events per second. This puts a stress on the software stack that it was not written to handle, and needs a significant overhaul.



(a) Trigger data flow scheme before the upgrade

(b) Trigger data flow scheme post-upgrade

Figure 8: Comparison of the two triggering solutions

2.7 Overview of the real-time processing software

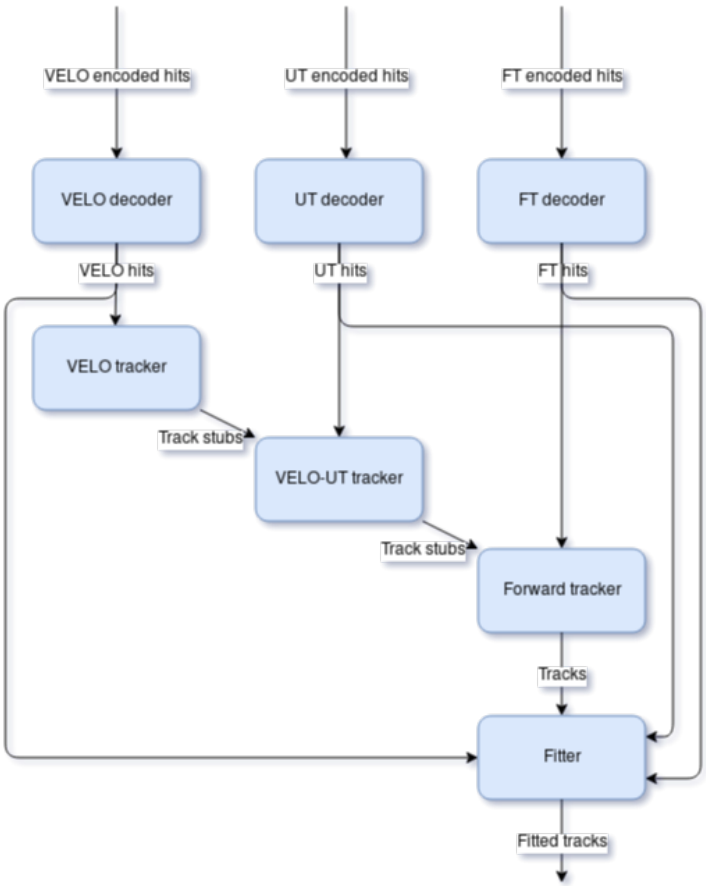


Figure 9: Simplified view of algorithms that perform online reconstruction.

The real-time processing software consists of algorithms which do a specific piece of the reconstruction of the particle paths. In addition to the reconstruction, there are other algorithms which do the selection of interesting events. Since the selection algorithms are generally a lot faster than reconstruction algorithms, they are excluded them from this discussion.

The general principle behind reconstruction is that a list of hits (where a pixel or silicon strip was hit) are given, and regular alignment of the hits indicate the path of the particle: hence the name *pattern recognition*. Once all particles have been found by gathering the hits they created, they can be identified and fed into the selection decision making.

The information on which pixels and strips were hit comes straight from the detectors, in a heavily compressed binary format. It has to be first decoded to give the position of the individual strips, a process done by the **VELO decoder**, **UT decoder** and **FT decoder**.

Once pixel hit positions are known by their global X,Y,Z coordinates, the **VELO tracker** find hits that align to form a straight line. It estimates the starting position and

the slope of the line. The line is extended into the UT, where the **VELO-UT tracker** searches silicon strip hits that are very close to the extended line. From the slight offset of the strip hits from the extended line, the VELO-UT tracker can estimate the amount of bending in the magnetic space, thus the momentum of the particle. The last step of tracking is done by the **forward tracker**, which uses the momentum estimate to extrapolate the bent path of the particle through the magnet, to the layers of the FT. It then finds silicon strips of the FT which align into a straight line (no magnetic field in the FT either), the line having the right direction and position to be a possible path the particle took after going through the magnet.

Finally, the **fitter** uses all the hits to align the particles path more closely with the position of the hits. In principle, it works similarly to a curve fitting solution, but uses a Kalman filter internally. Having the most accurate path, the decision is made to keep or drop the event.

2.8 The aim of this thesis project

As mentioned, the abandoning of the hardware trigger stage highly increases the load on the software trigger. The main goal of this project is to optimize the current software trigger to make it about 3 times as fast. Failure to do so will result in valuable events being dropped, thus reducing the physics potential of the experiment.

Current computing hardware has changed significantly from the ones the software trigger was originally made for. The even larger gap between memory and CPU speeds demands a more efficient use of CPU caches. Additionally, CPU instruction sets now include SIMD operations, which can, for example, do 4 floating point operations in place of one in the same amount of time. Furthermore, modern CPUs have a complex logic for branch prediction and instruction pipelining, which require code to be tailored to serve them.

To exploit the full capability of current hardware, not only individual pieces of the trigger software need to be changed, but the global data flow also has to be rethought and optimized.

During this thesis project, I will be helping the LHCb collaboration to reach its optimization goals for the software trigger.

3 Choosing optimization targets

As mentioned in 2.7, the reconstruction consists of individual algorithms which account for the bulk of the computation. (Scheduling the algorithms and culling decisions account for a much smaller CPU load.) It is straightforward to first start optimizing the algorithms which take the largest chunk of available computing power.

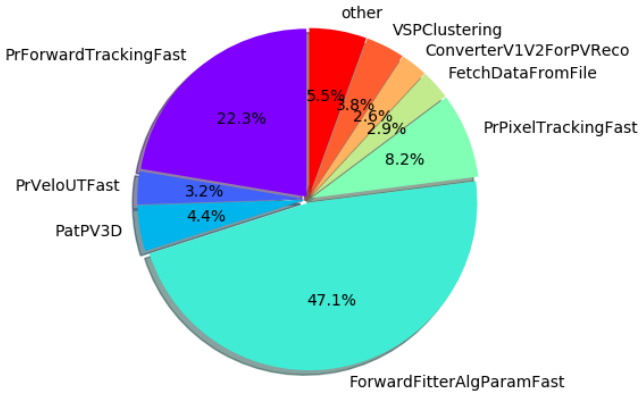


Figure 10: Workload split among HLT1 algorithms.

Looking at figure 10, we can see that the parametrized Kalman fitter takes nearly half the CPU budget, followed by the forward tracking which takes roughly a quarter. Based on this and initial performance profiling of the algorithms for hotspots, I decided to first examine and optimize the Kalman fitter.

4 Parametrized Kalman Fitter

As described in 2.7, the track is reconstructed incrementally, start with velo hits, extended by UT hits and finally adding the FT hits. This process, however, is not so accurate. This manifests itself in the creation of *ghost tracks* and missed tracks, and generally, tracks are only roughly aligned with the hits they were made from. Ghost tracks are tracks that did not exist in the real collision, they are merely artifacts of the reconstruction algorithms. As such, ghost tracks are highly undesirable, but the Kalman fitter helps to weed them out. The Kalman fitter basically refines the rough tracks that are spit out by preceding algorithms. The state of a particle can be described by its position, direction, and the quotient of its charge and momentum. The Kalman fitter first estimates the particle's state at its birth position based on the VELO hits alone. After that, it extrapolates the state of the particle to the next hit, or in other words, simulates the particle's travel until the next hit using the laws of physics. The new, *predicted* state will have some deviation from the *observed* state (that is, the hit), however, the Kalman fitter can make a mathematically optimal estimate for the true state based on the prediction and observation. The very new optimal state estimate will then be extrapolated to the next hit again, and this repeats for all the hits of the track. As a result, the estimated state or path of the particle aligns more closely with the observed hits. In the case of ghost tracks, we can expect to have large deviations between the optimal estimated states and the observed hits, which could have slipped through initial reconstruction algorithms but show up for the fitter. Such tracks are removed from the list of tracks, and that's why fitting is important.

4.1 Performance profiling for hotspots in the Kalman fitter

Callees	CPU Time: Total ▼	CPU Time: Self
▼ ParameterizedKalmanFit::fit	100.0%	2.950s
▶ ParKalman::LoadHits	49.4%	8.786s
▶ ParameterizedKalmanFit::PredictState	24.4%	7.760s
▶ ParKalman::ExtrapolateToVertex	9.4%	0.400s
▶ ParKalman::UpdateState	8.5%	4.691s
▶ ParKalman::AverageState	6.6%	7.560s
▶ ParKalman::addInfoToTrack	0.9%	0.370s
▶ ParKalman::DoOutlierRemoval	0.2%	1.250s
▶ ParKalman::CreateVeloSeedState	0.1%	0.310s
▶ StatusCode::StatusCode<StatusCode::ErrorCode, voi	0.0%	0.150s

Figure 11: Hotspots, or which parts of the Kalman fitter takes most of the time. Measured by Intel VTune Amplifier XE.

Figure 11 shows what fraction of the CPU time is spent in individual functions of the code. We can nicely see how the theoretical steps of the Kalman fitting map to the functions:

- LoadHits: acquires position and measurement error of hits
- PredictState: extrapolates the state to the next hit

- UpdateState: makes an optimal estimate for the true state using the predicted state and the measured hit
- AverageState, ExtrapolateToVertex, etc.: various operations

There is a major and obvious problem however: just acquiring the data on which the computation is done should not take over 50% of the Kalman fitting, but more like 1%.

4.2 Loading hits in detail

Careful examination reveals the way hits are loaded through the so-called *Measurement providers*.

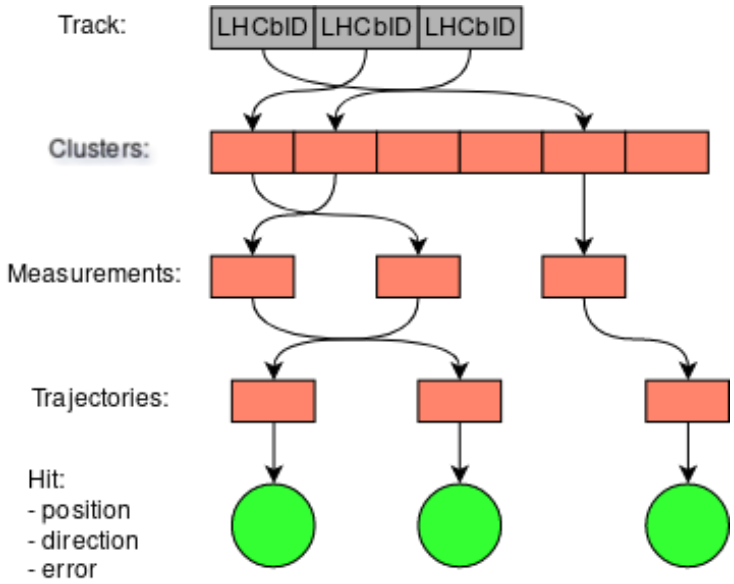


Figure 12: Illustration of how hit information is acquired from the array of LHCbIDs stored inside the Tracks. Contiguous array of clusters correspond to contiguous DRAM memory regions, while distinct objects, i.e. measurements have no spatial locality.

When a particle hits a detector, the identifier of the element of the detector that was hit is recorded. (Detector elements are analogous to the pixels of a digital CCD camera.) These elements are basically unambiguously identified by the so-called *LHCbIDs*, so it is enough to store the IDs inside the Track object and all information (such as location of the hit, measurement error) can be recovered.

Over the years however, this system grow unnecessarily complex resulting in a dramatic slowdown. Clusters, containing some basic information about the hit, such as its location, are stored inside measurement providers as a large array. In order to find the cluster that corresponds to the ID, this whole array is searched linearly. Once the cluster is found, a *Measurement* object is allocated on the heap and initialized from it. Finally, another object, called a *trajectory*, is queried from the measurement, from which the

data actually required can be extracted. The storage of clusters and creation of measurements is handled by *MeasurementProviders*. Additionally, we can distinguish separate measurement objects for the Velo, UT and FT hits.

As seen, this is a convoluted process, involving an asymptotically unacceptable linear search and a lot of dynamic memory allocation. Dynamic allocation is not only slow, it highly suffers from thread contention at the operating system level in our multi-threaded software. Additionally, the individually allocated objects are scattered around in memory, resulting in poor CPU cache performance.

Callees	CPU Time: Total ▼	CPU Time: Self
ParKalman::LoadHits	100.0%	8.786
MeasurementProviderT<MeasurementProviderTypes	45.4%	1.340
find_if<__gnu_cxx::__normal_iterator<const LHCb:	27.0%	0
DataObjectHandle<AnyDataWrapper<std::vector<L	12.7%	0
VPClusterPosition::position	2.3%	1.810
LHCb::VPMeasurement::VPMeasurement	1.5%	0.860
operator new	1.1%	3.880
GaudiHandle<IVPClusterPosition>::operator->	0.3%	0.050
LHCb::LHCbID::vpID	0.1%	0.050
func@0x3df1d0	0.0%	0.060
LHCb::LHCbID::isVP	0.0%	0
func@0x3e21a0	0.0%	0.030
FTMeasurementProvider::measurement	42.9%	1.419
std::__find_if<__gnu_cxx::__normal_iterator<LHCb	30.9%	39.068
FTMeasurementProvider::clusters	7.2%	0.080
LHCb::FTMeasurement::init	3.5%	0.371
operator new	0.9%	3.192
func@0x3df1d0	0.0%	0.060
func@0x3dfc30	0.0%	0.020
func@0x3df800	0.0%	0.020
func@0x3e30b0	0.0%	0.020

Figure 13: Breakdown of CPU usage of the LoadHits function

Figure 13 shows an excerpt from the CPU profiler and helps to understand where LoadHits spends its time. The most obvious thing is the std::find_ifs that take nearly 60% of LoadHits. This is the function that does the linear search among clusters. The rest of the overhead comes from various boilerplate code, clear trends cannot be understood, but the volume of the overhead is seen to be significant.

4.3 Simplifying the data loading

To avoid this long chain to acquire the required data, the hits should be directly stored inside the Track rather than only by their IDs. Ideally, this would not incur any performance penalty, since the algorithms preceding the Kalman fitter all use the position and error information associated with a hit, so the detector element identifier is fully decoded anyway.

As described, the Track object has the following content (largely simplified):

```
struct Track {
  std::vector<LHCbID> ids;
};
```

In the new model, the following structure is used:

```
struct TrackHit {
  Vector3D beginPosition;
  Vector3D endPosition;
  float errorX;
  float errorY;
};

struct Track {
  std::vector<LHCbID> ids;
  std::vector<TrackHit> veloHits;
  std::vector<TrackHit> utHits;
  std::vector<TrackHit> ftHits;
};
```

Notice how the IDs are kept: the unfortunate reason for this is that other algorithms rely on these, and they cannot be removed in this first iteration. This structure, however, completely eliminates clusters, measurement and trajectories from the chain, and the Kalman fitter reads the contiguously stored information straight out of the track. This does not stress the memory allocator and is friendly for the caches.

4.4 Performance profiling of the simplified model

Callees	CPU Time: Total ▼	CPU Time: Self
▼ ParameterizedKalmanFit::fit	100.0%	2.340s
▶ ParameterizedKalmanFit::PredictState	45.3%	6.420s
▶ ParKalman::ExtrapolateToVertex	23.7%	0.400s
▶ ParKalman::UpdateState	13.2%	4.020s
▶ ParKalman::AverageState	12.9%	5.950s
▶ ParKalman::addInfoToTrack	2.4%	0.400s
▶ ParKalman::LoadHits	1.2%	2.091s
▶ ParKalman::DoOutlierRemoval	0.2%	0.791s
▶ ParKalman::CreateVeloSeedState	0.1%	0.231s

Figure 14: Breakdown of the Kalman fitter after the simplified data loading

Figure 14 shows that with the new data model, the previous CPU hog, LoadHits, has completely disappeared, now accounting only for 2% of the fitting.

As the parametrized Kalman fitter takes about 47% of the entire reconstruction sequence, and about 50% of the fitter's computing load was removed by the above described code changes, we would expect an overall speedup of 31%. When measured, throughput increases from 4450 events processed per second to about 4850 events/second, or about 9.2%. As this is way less than the predicted 31%, the question arises as to where the performance is gone. First of all, three new data members were added to the Track to store the new TrackHits, and nothing has been removed. As Tracks are copied in the

code, the three std::vectors also have to be copied, which involves dynamic memory allocation (a well-known performance drag) and memory copying. Second, part of the code that produces the TrackHits from other objects was not removed, but merely moved out of the fitter to other algorithms. The data conversion to TrackHits, along with adding the TrackHits to the vectors and allocating the memory of the vectors adds additional overhead. In order to achieve the projected performance improvements, these issues have to be fixed and optimized.

4.5 High level optimizations

Besides the TrackHits (or IDs), the Track contained three additional dynamically allocated std::vectors, which were filled with valid data but were not necessary from a computing point of view. Removing these data members confirmed the hypothesis by which the additional data members in the track slowed down the algorithm sequence: I observed an increase of 17% in throughput (on top of the 9.2%) when removing these members. While this can be regarded as an optimization independent to the fitter itself, it gains back the speed lost with the additional members required by the fitter.

4.6 Micro-optimizations

To trade physics quality for performance, event culling decisions can be made earlier in the reconstruction sequence, before fitting. This results in fitting taking a lot smaller part of the entire sequence while other algorithms become more prevalent. My work, although sped the fitting up, slightly slowed other algorithms down. Consequently, the *best physics* case experienced a large increase in throughput, but the *best throughput* case got slightly slowed down. In an attempt to restore the performance of the other algorithms, I had to further analyze performance.

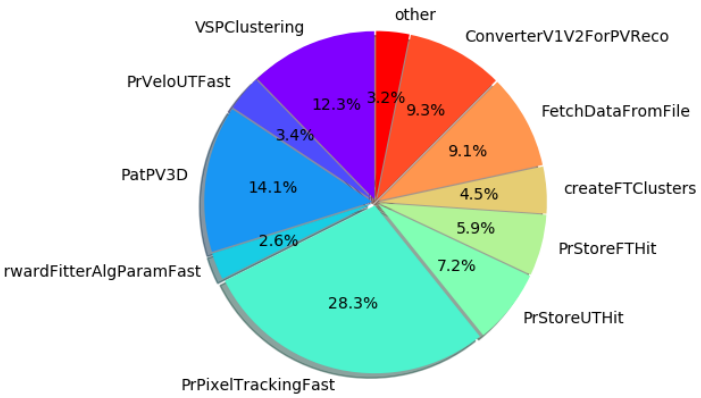
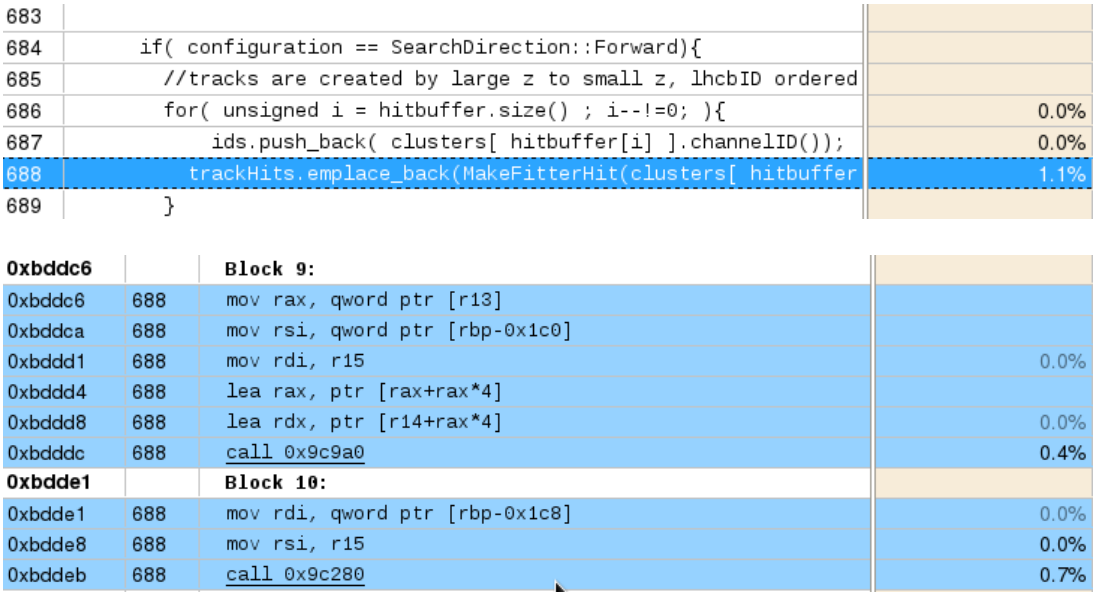


Figure 15: Distribution of CPU time among algorithms in the *best throughput* case with early event culling

Figure 15 shows the in the best throughput case, the pixel tracking algorithm takes the most amount of time. This algorithm is responsible for finding particle track stubs from only Velo hits, and was negatively affected by my fitter optimizations.



```
683
684       if( configuration == SearchDirection::Forward){                    
685         //tracks are created by large z to small z, lhcbID ordered       
686         for( unsigned i = hitbuffer.size() ; i--!=0; ){          0.0%      
687             ids.push_back( clusters[ hitbuffer[i] ].channelID());  0.0%    
688           trackHits.emplace_back(MakeFitterHit(clusters[ hitbuffer 1.1%    
689         }
```

```
0xbddc6          Block 9:
0xbddc6   688    mov rax, qword ptr [r13]
0xbddca   688    mov rsi, qword ptr [rbp-0x1c0]
0xbddd1   688    mov rdi, r15                                    0.0%
0xbddd4   688    lea rax, ptr [rax+rax*4]
0xbddd8   688    lea rdx, ptr [r14+rax*4]                        0.0%
0xbdddc   688    call 0x9c9a0                                    0.4%
0xbdde1          Block 10:
0xbdde1   688    mov rdi, qword ptr [rbp-0x1c8]                  0.0%
0xbdde8   688    mov rsi, r15                                    0.0%
0xbddeb   688    call 0x9c280                                    0.7%
```

Figure 16: Code snippet from the profiler which shows the code I added to the pixel tracking algorithms in blue highlight. The upper image shows the C++ source code, the lower image shows the corresponding x86-64 disassembly.

Looking at the disassembly, we can see two *CALL* instructions, which correspond to the two function calls *MakeFitterHit* and *emplace_back*. This means that the functions haven't been inlined. Inlining is a complex topic, because it can make code faster by removing function prologues, but excessive inlining can also make code slower by polluting the instruction caches with two many repeated code snippets. In this case, the latter is unlikely, since these function are only present at this location, so inlining would be preferable.

```
684       if( configuration == SearchDirection::Forward){
685         //tracks are created by large z to small z, lhcbID ordered
686         for( unsigned i = hitbuffer.size() ; i--!=0; ){                    0.0%
687           const auto cluster = clusters[ hitbuffer[i] ];                   0.0%
688             ids.push_back( cluster.channelID());
689
690           //LHCb::TrackHit hit;
691           Gaudi::XYZPointF beginPoint = { cluster.x(), cluster.y()
692
693           // Get the sensor and calculate error.
694           const LHCb::VPChannelID channel = cluster.channelID();
695           const DeVPSensor* sensor = m_vp->sensorOfChannel(channel
696           const unsigned int sensorNumber = sensor->sensorNumber()
697
698           bool isLong = sensor->isLong(channel);                           0.2%
699           float errorx = isLong ? m_errorXLong[sensorNumber] : m_e          0.0%
700           float errory = isLong ? m_errorYLong[sensorNumber] : m_e          0.0%
701
702           trackHits.emplace_back(beginPoint, beginPoint, errorx, e
703         }
704       }
```

```
0xbdc30          Block 9:
0xbdc30   702    mov rbx, qword ptr [rbp-0xd8]
0xbdc37   702    cmp rbx, qword ptr [rbp-0xd0]                              0.0%
0xbdc3e   702    mov rsi, rbx                                               0.0%
0xbdc41   702    jz 0xbdee8 <Block 35>                                      0.0%
0xbdc47          Block 10:
0xbdc47   702    pxor xmm0, xmm0
0xbdc4b   702    mov dword ptr [rbx+0x40], 0x0                              0.0%
0xbdc52   702    add rbx, 0x48                                              0.0%
0xbdc56   702    pxor xmm2, xmm2                                            0.0%
0xbdc5a   702    pxor xmm1, xmm1
0xbdc5e   702    cvtss2sd xmm0, dword ptr [rbp-0x1b0]                       0.0%
0xbdc66   702    cvtss2sd xmm2, dword ptr [rbp-0x1b8]                       0.0%
0xbdc6e   702    movsd qword ptr [rbx-0x38], xmm0                           0.3%
0xbdc73   702    movsd qword ptr [rbx-0x20], xmm0                           0.0%
0xbdc78   702    pxor xmm0, xmm0                                            0.0%
0xbdc7c   702    cvtss2sd xmm1, dword ptr [rbp-0x1a8]                       0.0%
0xbdc84   702    movsd qword ptr [rbx-0x48], xmm2                           0.0%
0xbdc89   702    cvtss2sd xmm0, dword ptr [rbp-0x1c8]                       0.0%
0xbdc91   702    movsd qword ptr [rbx-0x40], xmm1                           0.1%
0xbdc96   702    movsd qword ptr [rbx-0x18], xmm0                           0.0%
0xbdc9b   702    pxor xmm0, xmm0                                            0.0%
0xbdc9f   702    movsd qword ptr [rbx-0x30], xmm2
0xbdca4   702    movsd qword ptr [rbx-0x28], xmm1                           0.0%
0xbdca9   702    cvtss2sd xmm0, dword ptr [rbp-0x1c0]                       0.0%
0xbdcb1   702    movsd qword ptr [rbx-0x10], xmm0                           0.0%
0xbdcb6   686    cmp dword ptr [rbp-0x1a4], 0xffffffff                      0.0%
0xbdcbd   702    mov qword ptr [rbp-0xd8], rbx                              0.0%
0xbdcc4   686    jz 0xbe180 <Block 54>                                      0.0%
```

Figure 17: Source code and disassembly after manually inlining *MakeFitterHit* inside the *for* loop.

As can be seen on figure 17, both *CALL* instruction have disappeared. The first one for *MakeFitterHit* due to the manual inlining, and the second for emplace_back because

of the compiler automatically inlining it. Note that the automatic inlining was enabled by passing the constructor arguments of *TrackHit* to *emplace_back* instead of the ready object, exactly as *emplace_back* was meant to be used. Now, theoretically, the compiler could optimize out both cases as their semantics are equivalent, but it is apparently not capable of doing so.

Besides the absence of function calls, there is another thing noticeable on the assembly instruction. There is a large number of instructions moving quad words (*MOVSD*), that is 64 bit double precision numbers. Furthermore, the *CVTSS2SD* instructions are converting 32 bit single precision numbers to 64 bit doubles. Looking at the source code, we can indeed notice that input data from which the *TrackHit* is made is stored as single precision floats, but the *TrackHits* themselves are double precision because the fitter is using double precision calculations. Changing *TrackHit* to store single floats as well, thus delaying the conversion, will hurt performance at another place where it has less of an impact.

0xbdc28		**Block 9:**	
0xbdc28	702	mov rbx, qword ptr [rbp-0xd8]	
0xbdc2f	702	cmp rbx, qword ptr [rbp-0xd0]	0.0%
0xbdc36	702	mov rsi, rbx	0.0%
0xbdc39	702	jz 0xbded8 <Block 35>	0.0%
0xbdc3f		**Block 10:**	
0xbdc3f	702	movss xmm4, dword ptr [rbp-0x1b8]	
0xbdc47	702	mov dword ptr [rbx+0x20], 0x0	0.0%
0xbdc4e	702	add rbx, 0x24	0.0%
0xbdc52	702	movss xmm5, dword ptr [rbp-0x1a8]	0.0%
0xbdc5a	702	movss xmm6, dword ptr [rbp-0x1b0]	
0xbdc62	702	movss xmm1, dword ptr [rbp-0x1c8]	0.0%
0xbdc6a	702	movss xmm7, dword ptr [rbp-0x1c0]	0.0%
0xbdc72	702	movss dword ptr [rbx-0x24], xmm4	0.0%
0xbdc77	702	movss dword ptr [rbx-0x20], xmm5	0.0%
0xbdc7c	702	movss dword ptr [rbx-0x1c], xmm6	0.2%
0xbdc81	702	movss dword ptr [rbx-0x18], xmm4	0.0%
0xbdc86	702	movss dword ptr [rbx-0x14], xmm5	0.0%
0xbdc8b	702	movss dword ptr [rbx-0x10], xmm6	0.0%
0xbdc90	702	movss dword ptr [rbx-0xc], xmm1	0.0%
0xbdc95	702	movss dword ptr [rbx-0x8], xmm7	0.0%
0xbdc9a	686	cmp dword ptr [rbp-0x1a4], 0xffffffff	0.0%
0xbdca1	702	mov qword ptr [rbp-0xd8], rbx	0.0%
0xbdca8	686	jz 0xbe168 <Block 54>	0.0%

Figure 18: Disassembly after changing *TrackHits* to store single floats.

The disassembly on figure 18 clearly shows that the single precision to double precision conversions are gone just like the *PXOR* instructions, and now it only moves double word memory units. With this little change, I managed to throw out lots of unnecessary instruction and the amount of memory moved around is also smaller.

Due to the complex interactions inside modern, pipelined CPUs and between the CPU, the DRAM and caches, it is hard to explain how and why the changes affected the performance. Nevertheless, inlining and trimming the assembly code has increased performance of the *best throughput* scenario with early event culling from 13300 events per second to around 13700. Notably, the basic case without my code changes has produced about 14500 events per second. (As measured with my development branch on our

performance test machines during development.)

4.7 Results, conclusion

I managed to significantly increase the throughput of the *best physics* case from 4450 events per second to 5870 events per second, or a 32% increase. Unfortunately, the *best throughput* case slowed down from 14500 events/sec to about 13700, or 5% decrease in throughput.

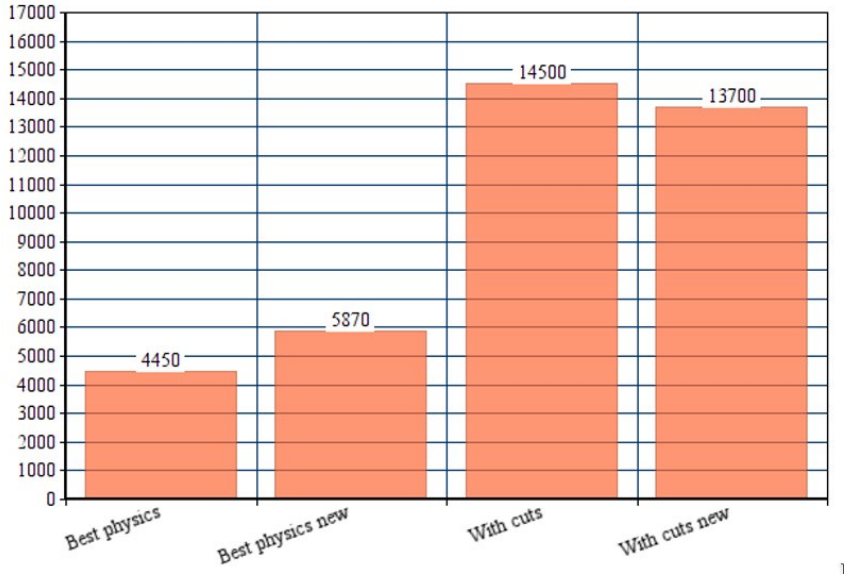


Figure 19: Throughput of the particle path reconstruction before and after my modifications, for the *best physics* and *best throughput* cases.

Another interesting figure to look at is how the weight of the parameterized Kalman fitter in the entire algorithm sequence has changed. Previously, it took 47% of the whole sequence, and now it only takes 22%. If in addition the fact that the whole sequence is significantly faster is factored in, the fitter is well above two times faster. (These type of measurements are to be taken with a grain of salt because of complex system interactions and the consequent inaccuracy of profiling, but are interesting and provide a good general view.)

(a) Before optimizations

(b) After optimizations

Figure 20: The distribution of processor time requirements of each algorithms. The parameterized Kalman fitter is identified by the label *ForwardFitterAlgParamFast*.

In light of the code changes and their effect on the overall performance, we can safely say that code should strive to do the data transformations in the simplest possible way. Adding extra layers on top, if not done carefully using zero-cost abstractions, will dramatically slow the code down. In performance critical applications, a good data oriented design can give far better benefits that assembly-level micro-optimizations. Additionally, a good data structures opens up the doors to more effective micro-optimizations, such as vectorization.

5 Streamlining the computation's data model

While profiling the Kalman filter, I had to change the data model by adding additional data to the `Track` which had adverse effects of performance. As the changes were temporary, the performance issue wasn't a concern, but the general performance of the data model was. Over the years the code accumulated quite some unnecessary data and code which resulted in issues with performance and maintainability. This chapter explains how data structures can be streamlined to better accodomate modern computer architectures.

5.1 The existing data model

The algorithms that constitute the processing chain are illu strated on figure 9. There are two main categories of data passed from one algorithm to the other: hits and tracks. All data related to a track is represented by the `Track` class, and in general, `std::vector`s of `Track`s are passed from one algorithm to the next. The `Track` was designed to be a generic object that is able to represent all the different types of tracks, for example a *velo track* or a *forward track*.

```
class Track {
  /* ... */
private:
  std::vector<LHCbID> lhcbIDs;
  std::vector<State> states;
  std::vector<Track*> ancestors;
  std::map<int, double> extraInfo;
  std::vector<Measurement*> measurements;
  std::vector<Node*> nodes;
};
```

Listing 1: Simplified code of the `Track` class.

In case of the trigger software, all tracks begin with the VELO. The VELO tracker finds hits that are aligned in a straight line and creates a *velo track* that is then forwarded to the VELO-UT tracker. The so called *velo tracks* contain the IDs of the pixels that were hit in the VELO detector, as well as the estimated state of the particle at its birth and when it left the VELO. Similarly, the VELO-UT tracker appends the IDs of the UT hits that it found as well as an additional state of the particle inside the UT detector.

5.2 Connection of the data model and the algorithms

Multiple algorithms have their own internal data structures for their computations, including special track objects, hit objects and particle states. In all cases, however, the generic input `Track` has to be converted to these internal data structures, which incurs an unnecessary overhead. The faster the algorithm itself gets, the larger in proportion the conversion overhead. The main tool to speed up algorithms is vectorization, which requires highly specialized data structures to be efficient, therefore requiring careful tuning of the general data layout to avoid conversion.

5.3 Issues with the existing model

5.3.1 Overgeneralization and code bloat

The `Track` object was meant to store every kind of track for any purpose. It is used by all the tracking algorithms, including the VELO, UT and FT tracking. Additionally, it contains derived values of the tracking, such as the estimated particle states, but also information related to final fitting. Most algorithms, however, access only a small fraction of this data, and make expensive checks to verify if the particular data they need is inside the `Track`. The `Track` is a prime example of the *god class* anti-pattern, which in this case not only compromises code quality but also negatively affects performance due to its heavy weight.

5.3.2 Memory allocations

The code snippet 1 shows only the heavy members of the `Track`. The `extraInfo`, `measurements` and `nodes` members are completely unused for the trigger, so they don't amount to any dynamic memory allocation, but they still amount to about 100 bytes of unused data (depending on implementation). The remaining members are actually used, each of them holding dynamically allocated memory. While the IDs and ancestors are only vectors of simple integer data, a `State` contains about 80 bytes.

5.3.3 Expensive copies

Due to the framework's implementation, each algorithm writes its output (a `vector` of `Track`s) into an immutable storage. Since the `Track` was intended to be mutated by the algorithms, each algorithm has to make a copy of the immutable storage. Because of the several dynamic allocations for each `Track` and the sheer size of the members and dynamically allocated storage, the copying takes a significant amount of time.

5.3.4 Inefficient memory access patterns

Dynamically allocated memory reduces spatial locality of the data, which pollutes the CPU cache when an entire cache block is loaded for a much smaller piece of data. Additionally, lack of locality increases the chance of cache misses and disables prefetching techniques, negatively affecting performance. In this particular case, locality is an issue on the level of an individual `Track` which stores its data in dynamically allocated blocks and also on the level of the vector of `Track`s, which could have an SOA layout to store the `State`s of all tracks in the same vector.

5.3.5 Cannot be vectorized

To achieve maximum performance, it is necessary to make use of the SIMD capabilities of modern CPUs. Vectorization inherently works on SOA data structures, so it makes sense to have the data laid out in that scheme by default. With the current model, one would have to access the first state's x coordinate for 8 consecutive tracks and load them into a vector register to allow for vectorized calculations. This takes 8 memory accesses to non-contiguous locations to fetch the pointers to the vector of states, then another 8

memory accesses to fetch the actual x coordinates. This conversion takes significantly longer than the entire calculation, hiding all the profits of vectorization.

5.4 Proposed model

5.4.1 Hits

Before getting to the `Track` object, the structures representing the hits had to be refactored. Generally, there were 3 different hit `class`es, one for the VELO hits, one for the UT hits and one for the FT hits.

```
class Hit final {
public:
  Hit(LHCb::UTChannelID chanID, unsigned int size, bool highThreshold, double dxDy, double xat0,
     double zat0, double yBegin, double yEnd, double cos, double error, unsigned int strip, double
     fracStrip) { ... }

  float cos() const { return m_cos; }
  float cosT() const { return (fabs(m_xAtYEq0) < 1.0E-9) ? 1. / std::sqrt(1 + m_dxDy * m_dxDy) : cos()
     ; }
  float dxDy() const { return m_dxDy; }
  bool highThreshold() const { return m_highThreshold; }
  bool isYCompatible(const float y, const float tol) const { return yMin() - tol <= y && y <= yMax() +
      tol; }
  bool isNotYCompatible(const float y, const float tol) const { return yMin() - tol > y || y > yMax()
     + tol; }
  LHCb::LHCbID lhcbID() const { return LHCb::LHCbID(m_chanID); }
  LHCb::UTChannelID chanID() const { return m_chanID; }
  int planeCode() const { return 2 * (m_chanID.station() - 1) + (m_chanID.layer() - 1) % 2; }
  float sinT() const { return tanT() * cosT(); }
  int size() const { return m_size; }
  float tanT() const { return -m_dxDy; }
  float weight() const { return m_weight * m_weight; }
  float error() const { return 1.0f / m_weight; }
  float xAt(const float globalY) const { return m_xAtYEq0 + globalY * m_dxDy; }
  float xAtYEq0() const { return m_xAtYEq0; }
  float xAtYMid() const { return m_x; }
  float xMax() const { return std::max(xAt(yBegin()), xAt(yEnd())); }
  float xMin() const { return std::min(xAt(yBegin()), xAt(yEnd())); }
  float xT() const { return cos(); }
  float yBegin() const { return m_yBegin; }
  float yEnd() const { return m_yEnd; }
  float yMax() const { return std::max(yBegin(), yEnd()); }
  float yMid() const { return 0.5 * (yBegin() + yEnd()); }
  float yMin() const { return std::min(yBegin(), yEnd()); }
  float zAtYEq0() const { return m_zAtYEq0; }
  unsigned int strip() const { return m_strip; }
  double fracStrip() const { return m_fracStrip; }
  int pseudoSize() const;

private:
  float m_cos;
  float m_dxDy; ///< The dx/dy value
  float m_weight; ///< The hit weight (1/error)
  float m_xAtYEq0; ///< The value of x at the point y=0
  float m_yBegin; ///< The y value at the start point of the line
  float m_yEnd; ///< The y value at the end point of the line
  float m_zAtYEq0; ///< The value of z at the point y=0
  float m_x;
  LHCb::UTChannelID m_chanID;
  unsigned int m_size;
  bool m_highThreshold;
  unsigned int m_strip;
  double m_fracStrip;
};
```

Listing 2: The original class definition for UT hits.

As it can be seen on code example 2, these were full-fledged classes which accumulated quite some cruft over the years, having unused data members and a long list of trivial and less trivial methods. The requirements for the hit classes are the same as for the track: plain old data objects that can be represented in an SOA format for efficient vectorization. Additionally, it is very important to keep the size of these objects to the minimum because there are a large number of hits and they are heavily used in computations.

Without further explanation of the original data members, this class can be reduced to the following:

```
struct {
  float x0;
  float y0;
  float z0;
  float y1;
  float dxdy;
  float error;
  uint32_t id;
};
```

Listing 3: Plain old data array of structure format for UT hits.

Despite the reduction in code size, the simplified structure is enough for the VELO-UT tracker to perform its task. The structure is a minimal representation of a UT silicon strip in 3D space: contains the $\begin{bmatrix} x_0 & 0 & z_0 \end{bmatrix}$ starting point, the $[y0, y1]$ interval that represents the two ends of the fiber and the dx/dy slope of the fiber. Note that the fiber is almost vertical (aligned with the Y axis), in same cases tilted slightly by a Z-axis rotation, so these parameters are valid and precise in all cases. The error represents the uncertainty of the particle due to the fiber's width and the ID is a unique identifier for each fiber.

A similar scheme can be applied to both the VELO and the FT hits.

5.4.2 Tracks

First, we have to realize that the `Track` class breaks the single responsibility principle by representing multiple objects of multiple purposes under the same class. Second, the only thing that's necessary from the `Track` object illustrated in code snippet 1 is the `LHCbID`s (or hits), the `State`s and the information as to which UT track came from which VELO track (`ancestors`).

In light of this, it makes sense to separate the states and the sets of hits into two separate `vector`s of the same size, and the output of the VELO tracking algorithm becomes

```
std::tuple<
  std::vector<VeloTrack>,
  std::vector<State>
>;
```

where a `VeloTrack` is defined as

```
struct VeloTrack {
  std::vector<VeloHit> veloHits;
};
```

Note that there is no need for separately specifying the `LHCbID`s as the `VeloHit`s contain them along with all the other information about the hit that is necessary for later analysis of the track.

Similarly, we can define further types of tracks:

```
struct UtTrack {
  std::vector<UtHit> utHits;
  const VeloTrack* veloSegment;
};
struct FtTrack {
  std::vector<FtHit> ftHits;
  const UtTrack* utSegment;
};
```

Listing 4: The UT (*upstream*) and FT *long* tracks also contain a reference to the track they were extended from.

The algorithms and their input and output are best illustrated with a graph. The decoders return a plain vector of hits. The tracking algorithms operate on the track stubs and the hits, and write out the extended tracks and the estimated states of the particle. Note that since the `Track` and are split, the VELO-UT tracker is able to consume only the particular states that it needs.

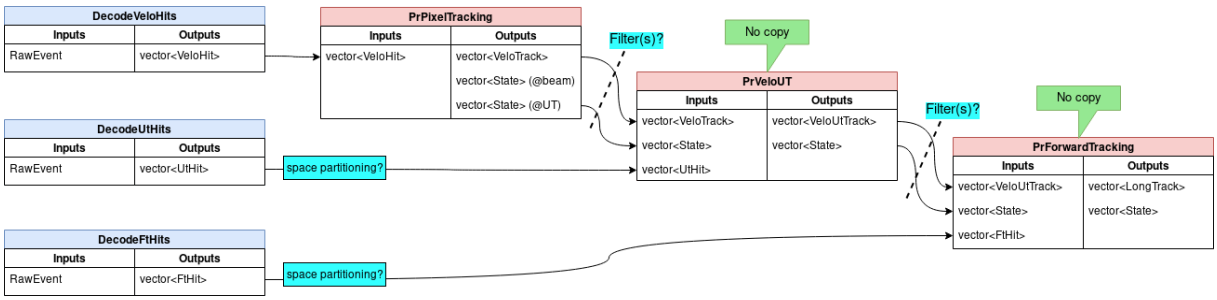


Figure 21: Algorithms and data flow with the simplified data model.

5.5 Results

5.5.1 Problem mitigation

As explained in 5.3, there were several issues with the original data mode.

As it can be seen on the code snippets, the amount of code has been greaty reduced. Most notably, trivial getter and setter methods have been replaced by public members, and non-trivial getters have been extracted to free functions to where they are needed. Code that is not used in the context of the trigger or tracking has been removed permanently.

The number of memory allocations was somewhat reduced. The states are now stored in a contiguous array instead of one dynamic block for every track, and the ancestors have also been restructured to be implicit to the track types instead of a dynamically allocated vector per track. Hits, however, are still reside in per-track heap storage.

Copies of dynamically allocated objects have been elminated, since when an algorithm extends an existing track it stores a pointer to it along with the new track instead of making a copy of the track and modifying it with the extended parts.

Memory access patterns are slightly better, as the states reside in a large contiguous chunk of memory. In practice, this unfortunately does not help as the algorithms don't process all states in a tight loop, but rather on a more on-by-one basis. Additionally, algorithms still use only about one third of the data in states.

Thanks to the plain old data structures, vectorization is easier, however it still requires deinterleaving the arrays of structures. Fortunately, POD AOS layouts are fairly easy to turn into SOA layouts.

5.5.2 Performance

The removal of the unused members of the track object resulted in a very significant, 17% gain in the best physics configuration. The realization of the split `Track` object between the VELO and the VELO-UT tracker alone resulted in a 4.8% gain in overall performance. Unfortunately, I wasn't able to measure performance in different configurations and with the entire network of algorithms refactored because of project management policies.

5.6 Future work

The primary goal of restructuring the data model was not actually to create a highly performant one, but to create a stepping stone for more efficient data models.

5.6.1 Moving to structures of arrays

The new plain old data structures use an array of structures format. As an example, let's take the UT hits in an AOS layout:

```
struct UtHit {
  float x0;
  float y0;
  float z0;
  float y1;
  float dxdy;
  float error;
  uint32_t id;
};

using UtHits = std::vector<UtHit>;

UtHits hits = { /*...*/ };
const float& x = hits[0].x0;
```

The SOA layout contains the same data members, however, single objects cannot be used anymore:

```
struct UtHits {
  std::vector<float> x0;
  std::vector<float> y0;
  std::vector<float> z0;
  std::vector<float> y1;
  std::vector<float> dxdy;
  std::vector<float> error;
  std::vector<uint32_t> id;
};

UtHits hits = { /*...*/ };
const float& x = hits.x0[0];
```

Let's take a simple algorithm that calculates the x coordinate of the fiber at the $y0$ height, using the SOA layout:

```
UtHits hits = { /*...*/ };
std::vector<float> xAtY1{numHits};

for (int i = 0; i<numHits; ++i) {
  xAtY1[i] = hits.x0[i] + hits.y0[i] + hits.dxdy[i];
}
```

The first notable difference between the SOA and AOS layouts is the memory acces pattern of the algorithm. In the SOA case (as on the code snippet), the algorithm reads consecutive elements from three vector and writes contiguous elements into a fourth vector. A cache line is generally 64 bytes, while a floating point variable is 4 bytes. When the first element of the vector is read, the CPU loads the entire cache line it is in to the L1 cache. This means that not only the floating point number we read get cached, but also the subsequent 16 values. In the SOA case, however, this is not a problem, because the subsequent 15 reads will access those very elements, which are now served from the extremely fast L1 cache. If `numHits` was 1024, then the CPU in total would fetch $3 \cdot (1024/16) = 192$ cache blocks from RAM and move them to L1.

In the AOS case, the `UtHit` structure takes up $6 \cdot 4 + 4 = 28$ bytes (and it has no padding), and the CPU is reading of a large array of 28 byte blobs. The first read to `hits[0].x0` pull in the next cache line, similarly. However, that cache line now contains `hits[0].y0`, `hits[0].z0`, `hits[0].y1` and so on, up to all members of `hits[1]` and some of `hits[2]`. As all the data is pulled in to the cache, to calculate the number of caches line we simply write $1024 \cdot 28/64 = 448$ cache lines. Due to the data members that are not actually used for the calculations, the CPU is forced to read 2.33 times more memory.

This disparity in the amount of memory read may be hidden by the calculations, but in some cases can greatly increase performance.

5.6.2 Vectorization

Vectorizing the above-mentioned algorithm in the AOS case can be realizing with gather instructions. We need to have the `x0` data member of the first 8 `UtHit`s in a vector register, these members are, however, 28 bytes apart, equally spaced. The CPU can execute a single gather instruction that perform the 8 reads, but this instruction is very inefficient and take only slightly less time than performing the 8 reads manually.

In case of an SOA layout, the `x0` members form a contiguous array with no padding, thus, fetching 8 of them is as simple as reading 32 bytes into the vector register. The CPU can very efficiently execute this, taking no more time than loading a single `float`.

Note that with the original data model, not even the AOS version of vectorization is feasible, because the members are only accessible through getter methods.

Vectorization, in the ideal case, can speed up code execution by 8 times. Though this is rarely achieved in practice, it is still one of the most effective ways to optimize the individual algorithms, so the data model must be able to support it.

6 Vectorizing and optimizing the Velo-UT algorithm

As highlighted in 2.7, the Velo-UT algorithm extends the straight line tracks created in the VELO tracking algorithm by assigning UT silicon strip hits to it. More importantly, the Velo-UT algorithm provides a course estimate for the momentum of the particle, allowing efficient tracking in the FT. The algorithm takes roughly 10 percent of the pipeline, so optimizing it is not expected to provide a high overall gain in performance, however its code is outdated and is in much need of an overhaul. My goal for the Velo-UT was to streamline the code to adhere to modern coding practices, and in the meantime, to make the algorithm play well with modern CPU architectures. In light of this, I was aiming at a highly SIMD-vectorized code, which operates with SOA data structures

6.1 Geometry of the UT detector

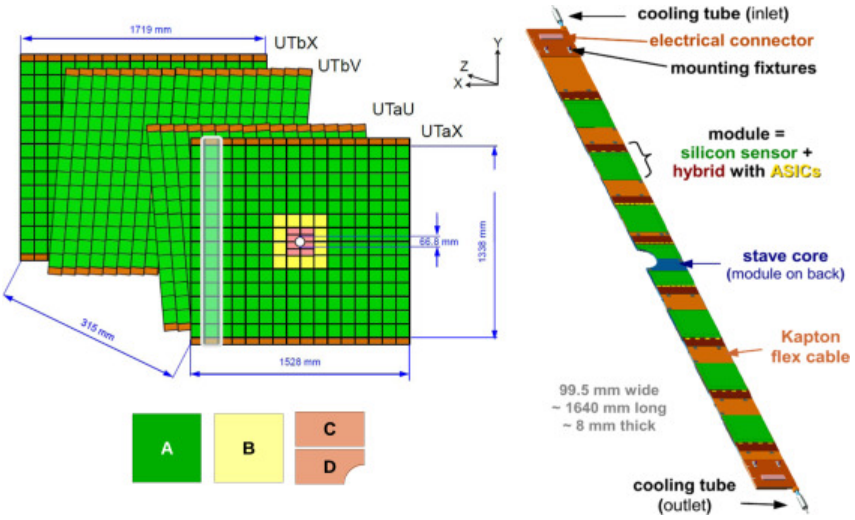


Figure 22: Illustration of the key elements of the UT hardware.

Figure 22 shows the rectangular panels of the UT as looking down the detector from the VELO. The UT detector consists of four panels, their placement in the entire detector can be seen on figure 3. Each of the four panels is made up 10 cm by 10 cm sensors. Though both the front two and rear two panels contain 14 rows of sensors, the rear panels are wider. The middle two panels have their sensors tilted by +5 and -5 degrees, which helps to mitigate the poor vertical resolution resulting from using silicon strips instead of pixel detectors. Each sensor contains silicon strips that run the full length of 10 cm of the sensor. There are 512 strips next to each other on every sensor. The exceptions are the sensor near the center of the panels, since the ones marked in yellow contain 1024 strips, and the ones marked in red contain 1024 half-length strips. The higher density area in the middle is required to deal with increased luminousity in the axis of the beam pipe. For each collision event, the software receives a list of the silicon strips that were hit by a particle. The Technical Design Report for the LHCb tracker[9] explains the geometry in more detail.

6.2 Decoding of the raw data from the detector

Each silicon strip has a unique numeric identifier. During a collision event, the electronics of the UT detector compile a list of identifiers of the strips that were hit by a particle, which the software receives in a heaavily compressed format. With a description of the exact geometry of the detector, that is, the location and dimension of sensors and strips in the global 3D space, the software is able to represent the strips that were hit by a line segment in the 3D space instead of just an ID. The UTHits resulting from the decoding process are gathered into a HitHandler structure, which groups the hits by which sensor they belong to. The Velo-UT algorithm consumes the HitHandler alongside the VELO tracks to produce *upstream* tracks.

6.3 The original Velo-UT algorithm

The main idea of the Velo-UT algorithm is to consume the VELO tracks and the UT hits, and extend the VELO tracks with hits from the UT to create so called *upstream* tracks. The algorithm handles each VELO track separately. The process for a single VELO tracks can be broken down into the following steps:

1. Skip track if unstuitable
2. Extrapolate VELO track to the UT detector
3. Gather UT hits that are close to the extrapolated track
4. Find 3 or 4 UT hits that align in a line
5. Analyze the *track candidates* that consist of the VELO track + 3-4 UT hits
6. Select the best candidate (if any) to extend the VELO track

6.3.1 Filtering tracks

Any kind of tracks can be fed into the algorithm from the VELO tracking. Some tracks point to the wrong direction, away from the UT, other tracks go very close to the beamline and hit the central hole in the UT's panels, and some tracks may also simply miss the UT's panels as they go too much to the sides. These tracks are not worth trying to extend because there will be no solution, so they are dropped.

6.3.2 Extrapolating tracks

Tracks from the VELO are described by the *state* of the particle. The state is a tuple of x,y and z coordinates, and the slopes tx=dx/dz and ty=dy/dz. In other words, the particle has a position [x,y,z] and has a velocity vector in the direction [tx,ty,1]. The usual coordinate system (see 2.4.2) is used for the states as well. To gather UT hits, one has to know where a track crosses a particular panel of the UT, which can be calculated by extrapolating the x,y,z position of the state to the z position of the panel along to line given by the slopes.

6.3.3 Gathering UT hits

Once the state is extrapolated to the z coordinate of a panel, one can collect hits that are close to the x and y position of the extrapolated state. The implementation of the algorithm first collects the list of the 10 by 10 centimeter sensors that fall close enough to the extrapolated state. In case the track passes through the middle of one sensor, only one sensor will be tagged, however if the track hits the corner between four sensors, all the four may be tagged. Afterwards, the algorithm iterates through all the hits that belong to the tagged sensors, and checks for each hit if they are within tolerance to the extrapolated state. This, including the extrapolation, is repeated for all four panels, thus the final result is a set of close-by hits for each of the four panels.

6.3.4 Finding aligned hits

There are on average 2-3 hits returned for each panel. Among these hits, the algorithm tries to find 3 or 4 hits that form a fairly straight line. These are called track candidates, and since there may be multiple sets of hits that form a straight line, the algorithm must decide on which is the best candidate.

6.3.5 Analyzing candidates

Each candidate that belongs to a track goes through a fitting stage. This is much like curve fitting or linear regression, the algorithm find a mathematically optimal set of parameters that best describe the path of the particle that have created the UT hits of the candidate.

6.3.6 Selecting the best candidate

The error of the fitting of a candidate can be calculated by determining how far the parameterized path of the particle lies from the position of the UT hits it was fitted from. If any of the candidates have a sufficiently low error, the one with the lowest is picked to form the newly created upstream track.

6.4 Analysis of the original Velo-UT algorithm

6.4.1 Hotspots

Callees	CPU Time: Total	CPU Time: Self
PrVeloUT::operator()	100.0%	2.2%
PrVeloUT::getHits<std::array<float, (unsigned long)	49.8%	9.2%
PrVeloUT::formClusters	25.0%	6.7%
PrVeloUT::makeSlimTrack<std::array<float, (unsign	18.9%	4.8%
std::vector<LHCb::State, std::allocator<LHCb::State	0.7%	0.0%
std::vector<LHCb::State, std::allocator<LHCb::State	0.7%	0.0%
TrackHelper::TrackHelper	0.7%	0.7%
(anonymous namespace)::isVeloTrackBackwards	0.6%	0.6%
std::vector<LHCb::HLT1::Track, std::allocator<LHC	0.3%	0.1%
std::vector<std::vector<UT::Hit, std::allocator<UT::H	0.3%	0.2%
PrVeloUT::isTowardsBeampipe	0.2%	0.2%
std::vector<LHCb::HLT1::Track, std::allocator<LHC	0.2%	0.2%
std::vector<LHCb::State, std::allocator<LHCb::State	0.1%	0.1%
std::vector<LHCb::HLT1::Track, std::allocator<LHC	0.0%	0.0%
PrVeloUT::isMissingUt	0.0%	0.0%
LHCb::State::tx	0.0%	0.0%
LHCb::State::y	0.0%	0.0%

Figure 23: Resolts of hotspots profiling with Intel VTune.

As we can see, the Velo-UT algorithm spends most of its time gathering the UT hits that could potentially belong to a VELO tracks. The second largest portion is *formClusters*, which performs the search for aligned UT hits as well as a preliminary fit to determine the best candidate. The third largest unit is the creation of the output (the *upstream* tracks), which performs a higher quality final fit and does some analysis to remove potentially undesirable tracks.

In light of this, the primary target for improvement is the largest chunk: gathering the hits, however, the other hotspots are also of significant interest.

6.4.2 Gathering hits

Examining the code of the hit gathering function(6), we can see that the main idea is to loop over the four planes, extrapolate the track to the plane, and collect the hits for each plane.

The first step in locating the close-by UT hits happens by finding the sectors (that is, the 10-by-10 centimeter sensors) that are adjacent to the point where the extrapolated track hits the current panel. The `findSectors` function performs this step via a rather complex logic (which is not shown here).

Given the sectors of interest, a loop goes over the sectors and queries the list of hits that belong to that sector from the `HitHandler`. The `HitHandler` already has the UT hits grouped by sectors, so this operation is rather simple.

Finally, the `findHits` function loops over the hits that belong to one sector, and keeps the hits that are truly close to the extrapolated track. Truly close here means that the fiber, when fattened into a cylinder, intersects with the line of the extrapolated track. (In

reality, a simpler x and y tolerance is used instead of the heavy ray-cylinder intersection calculations.)

The final result is an array which contains four arrays of hits, one for each panel.

Looking at the code, we can spot several issues:

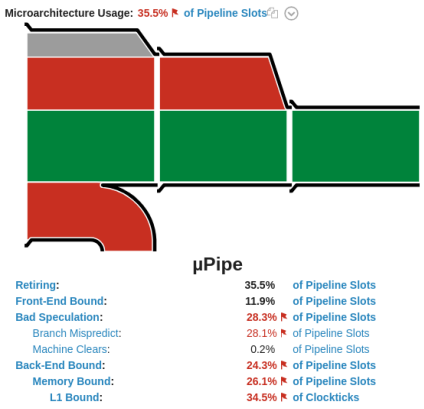
1. Getting the list of nearby sectors is complicated: the sectors do not form a regular grid because of the smaller ones in the middle, which needs to be handled when doing lookups by x,y coordinates.

2. Found sectors may be duplicated, as indicated by line 319-320, which is handled by special cases again.

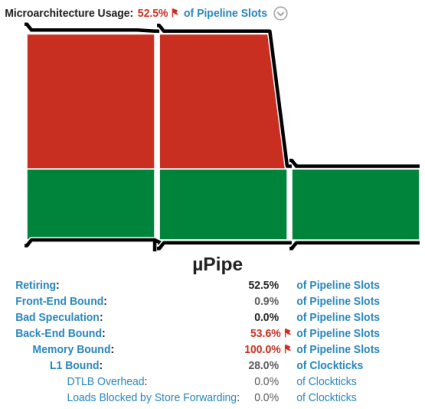
3. Early exit conditions to cut computation time on lines 299, 302 and 320.

Issues #1 and #2 indicate that more work is done than absolutely necessary, while branching caused #2 and #3 may directly affect the performance on modern superscalar CPUs with deep pipelining. The idea with early exit conditions is to skip computation that will prove useless, however if the CPU misses the branch prediction, it has to flush its entire pipeline and restart the other branch. It is often a better approach to vectorize the branchless code and do the unnecessary calculations as well, then filter the only the final results (with vectorized code). Overall, the code is not vectorized at all, and does not have data-structures that would allow for vectorization.

In addition to performance problems, as the code is not clearly structured, optimization opportunities are easy to miss and hard to exploit, for humans and compilers alike.



(a) Microarchitecture usage of gathering hits in the original code



(b) Microarchitecture usage of gathering hits in the refurbished code

Figure 24: Almost 30% of the instructions go to waste due to CPU branch mispredictions before refurbishing the code. (Illustration, in-depth analysis later.)

6.4.3 Finding aligned hits

The largest part of the hit finding function is in fact the fitting of the alignging track candidates. Fitting is a simple, linear stream of mathematical operation with no branching, thus makes a perfect candidate for vectorization. However, vectorization cannot be exploited unless it is completely separated from the alignment search.

PrVeloUT::formClusters	100.0%	26.9%
PrVeloUT::simpleFit<(unsigned long)4>	43.3%	8.6%
PrVeloUT::simpleFit<(unsigned long)3>	22.8%	3.7%
__gnu_cxx::__normal_iterator<UT::Mut::Hit const*,	0.9%	0.9%
std::vector<UT::Mut::Hit, std::allocator<UT::Mut::Hit	0.7%	0.7%
std::abs	0.7%	0.7%
__gnu_cxx::operator!=<UT::Mut::Hit const*, std::vec	0.5%	0.5%
__gnu_cxx::__normal_iterator<UT::Mut::Hit const*,	0.4%	0.4%
std::vector<UT::Mut::Hit, std::allocator<UT::Mut::Hit	0.4%	0.4%
std::abs	0.4%	0.4%

Figure 25: The breakdown of the aligned hit searching. The two simpleFit calls account for the fitting, while the remaining 30% is the alignment search.

6.4.4 Creating the final track

The data format of the final track is not optimal. A track is represented by its own class, which dynamically allocates memory for the small number of hits and other objects it has. Transforming the track into a structure of arrays format is something to consider, which means for example that all the hits for all the track would reside in a large array, and the individual tracks would be indices into that array. Additionally, as part of the creation of the track, a final, more accurate fit is performed. This is also a prime candidate for vectorization when separated from the rest of the code, and may also be merged with the preliminary fit to provide simpler and faster code, and potentially better results.

6.4.5 General code design

The code violates several coding best practices, which is not strictly an optimization problem, but is still an important concern as it is hard to find optimization opportunities in code that is hard to reason about. The most severe breaches are the single responsibility and dependency inversion priciples. An example for the former is `formClusters`, which not only find aligned hits but also parametrically fits them, an example for the latter is `simpleFit`, which modifies the flow control of its caller (formClusters) via modification of in-out (quasi-global) parameters. When the problems are alleviated, the search for aligned hits and the fitting can be done completely separately, which provides a massive gain thanks to the perfect vectorization of the fitting process.

6.5 Design of the improved algorithm

6.5.1 Uniform grid space partitioning

For the logic part, I left the algorithm as it was as I could not find better alternatives (maybe explain parabola and hough methods I tried), however I redesigned space partitioning from scratch.

Space partitioning, most prominent in 3D graphics and physics simulations, is a technique to accelerate computations that are done on a large number of objects contained in a 3D space. Frequently, there are only significant intereactions between nearby objects, so computations for the interactions of far away objects can be neglected. To be able

to quickly list objects nearby to a 3D coordinate, a space partitioning scheme is used. Arguably, the simplest sceheme is uniform grid space partitioning, which slices the 2D or 3D space by a uniform grid, and keeps a list of objects for each grid cell that fall into the cell. This way, to find objects near to a specific point, one only has to calculate the grid cell the point falls in (a few divisions) and read the list of that cell. (In some cases, the nearest four cells can be used not to miss any objects in case the specified point lies close to a grid line.)

To reduce the complexity of finding the nearby sectors of a plane for an extrapolated track, I replaced the sector-based lookup by a uniform grid-based lookup. Since there are 14 rows of sensors on every plane of the UT detector, I used a grid with 14 rows, carefully aligned to the geometrical position of the sensor rows. This ensures that a fiber falls entirely within its grid cell. For the half-height sensors in the middle, two sensors fall into the same grid cell. There are no such limits for the horizontal resolution of the grid cells, so the cell width can be set arbitrarily. As the search region in the order of a few centimeters, I chose to have 64 cells on the horizontal axis, resulting in a cell width of roughly 3 cm. The third axis in the direction of the beamline is divided so that each four panel of the UT falls into its own grid cell. To sum up, the grid is 64 x 14 x 4 cells.

add a figure for space partitioning

The binning (assignment of cell) of hits is done by taking the middle point of the fiber, and using a division on all three axes to determine the integer coordinates of the corresponding cell. Similarly, lookup of hits nearby to a point involves first a similar division to acquire the corresponding cell, then the hits assigned to that cell can be iterated over. One caveat, however, is that the middle two layers are tilted, so some tilted fibers cross multiple cells. In this case, we would like to have them show up when we query either of the cells. Instead of duplicating hits, I chose to simply sample the nearbest two cells instead of just one. With the cell size of 3 centimeters, it is unlikely to miss a hit in such way. The same issues persists in the vertical direction as well when the sampled point falls very close to a grid line. This is exacarbated by the small displacement on the Z axis that the sensors have by design of the hardware, and manufacturing errors also add to the problem. However, by design, there is a small overlap between two adjacent sensors, both on the vertical and horizontal axes. This means that a track hitting a grid line is likely to excite two fibers at the same time. The idea is that even when avoiding double-sampling on the vertical axis, at least one of those fibers will still be registeredd in the subject cells.

6.5.2 Data structures and steps

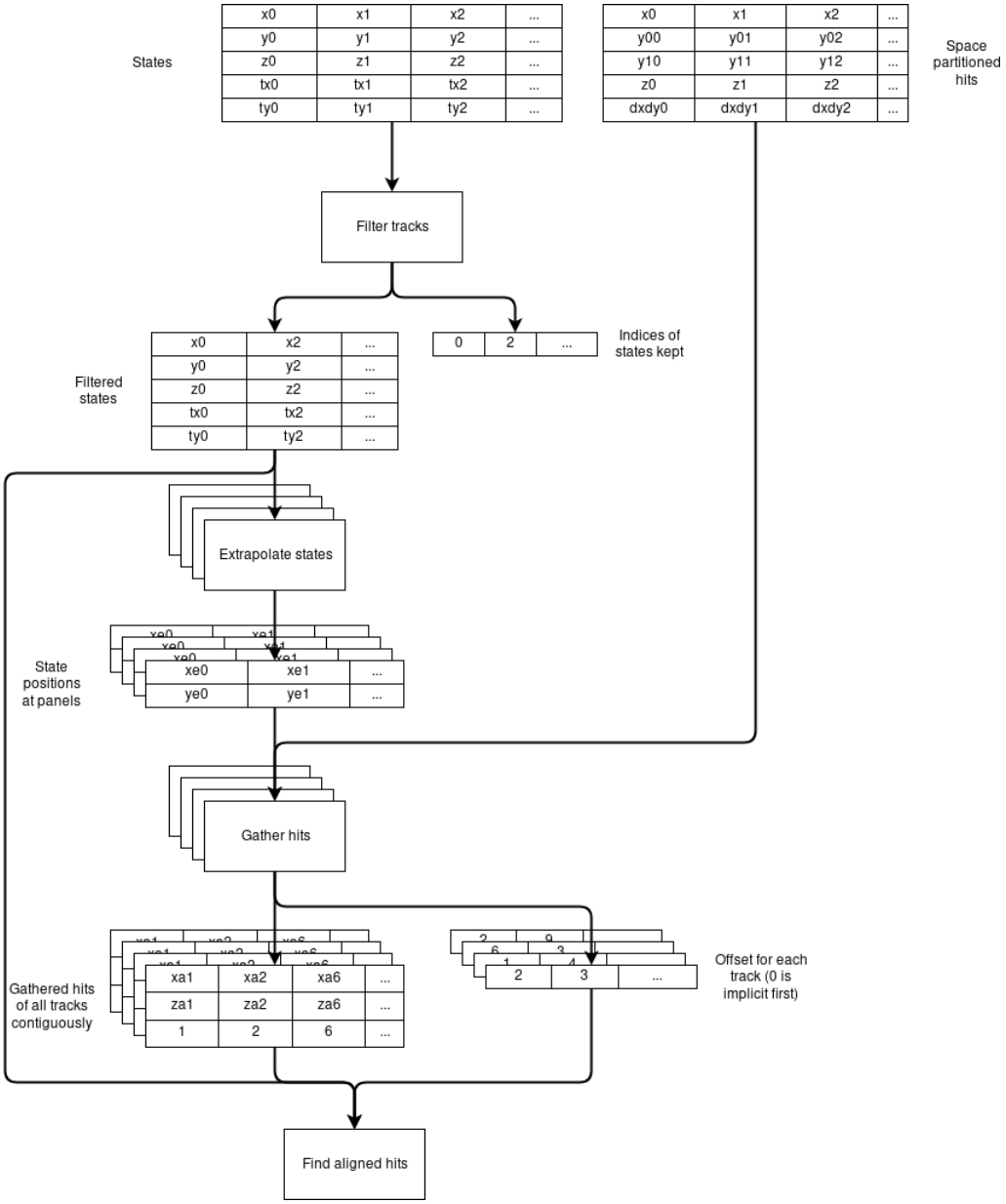


Figure 26: Data structures and process diagram of the new Velo-UT algorithm. (Part 1)

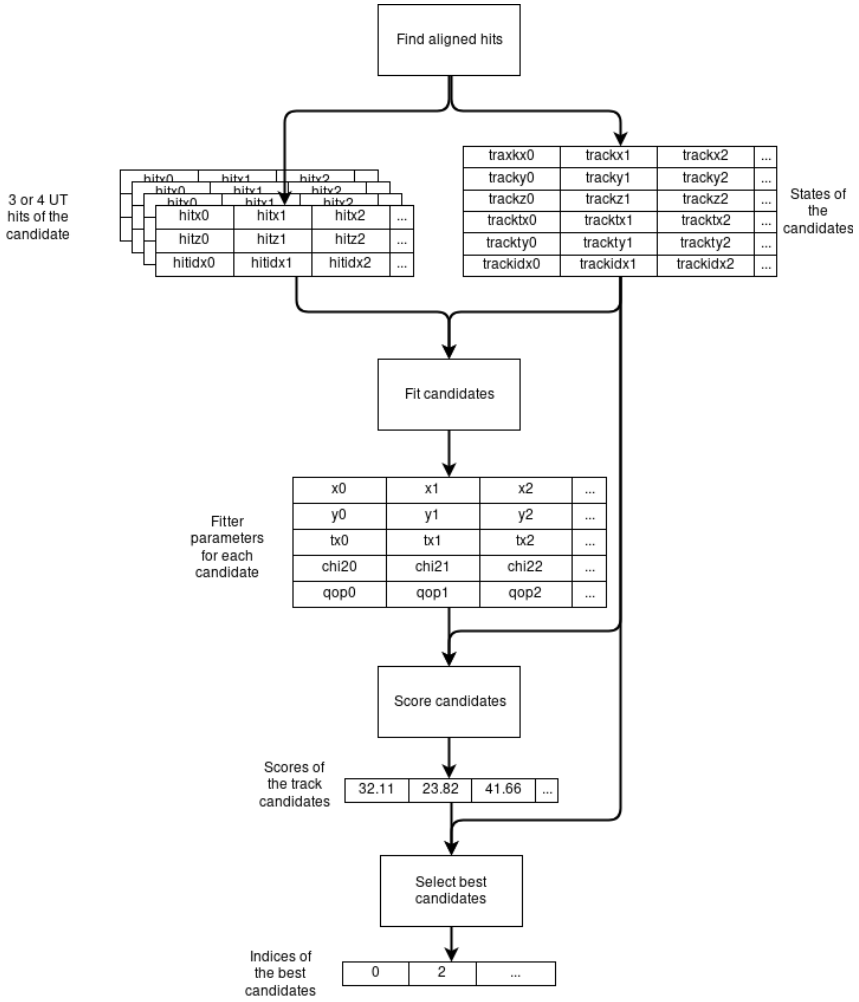


Figure 27: Data structures and process diagram of the new Velo-UT algorithm. (Part 2)

Layout of data

Figures 26 and 27 show the main steps of the algorithm as well as the exact data structures passed around. The data structures are structures of arrays (SOA), which are represented by tables on the drawing. Each row correspondonds to a certain variable, while the columns usually represent variables that belong to a certain track. In memory, rows are stored in a contiguous block separate from other rows.

The SOA layout serves three purposes:

- when reading a row, contiguous memory accesses are dispatched to the CPU, which makes it easy for the prefetcher to get data in the fast caches even before its used,
- for vectorization, operations can be done on multiple columns at the same time without de-interleaving the data,
- reading only a subset of the rows (i.e. only x and y but not z) incurs no waste unlike an AOS layout, where z would be loaded as it's in the same cache line.

Although SOA layouts generally make code harder to read and write, they are crucial to maximally exploit modern CPU architectures in performance critical applications.

Data

1. States: a structure of five arrays that contain an x, y and z the track passes through, and the track's direction vector tx,ty,1.
2. Space partitioned hits: as described in 6.5.1, the hits are space partitioned by panels by rows by bins. All the hits reside in a large structure of arrays. The array begins with the hits of panel 0, row 0 and bin 0, then continue with bin 1, bin 2 and so on.
3. Filtered states: a subset of States.
4. Extrapolated states: one for each filtered state, only the x and y positions of the state are stored as z is known and tx and ty are the same as in the filtered states.
5. Gathered hits: a partitioned array of hits that begins with the hits that belong to extrapolated state 0, then the hits for extrapolated state 1 follow and so on. Only the x and z of the fiber's closest point to the track is stored in addition to the index of the original hit in the space partitioned hit array. There are four such arrays, one for each panel.
6. Offset for each track: for each extrapolated state it tells the index of the last hit inside the gathered hits array.
7. 3-4 UT hits and states of the candidates: for each extrapolated state, zero or more candidates are constructed. The arrays store various parameters, included the four hits the make the candidate and the index of the extrapolated state the candidate belongs to. Thanks to the index, there is no need for an explicit array that tells the offsets into this array for each extrapolated state.
8. Fitter parameters: contains a set of parameters one-to-one correspondance with the candidates.
9. Scores: a simple real number for each track candidate.
10. Indices of best candidates: an array with as many elements as unique indices to extrapolated states in the array of candidates. Contains indices to the array of candidates that refer to the best candidate of the same extrapolated track.

Steps

The step illustrated on figures Figures 26 and 27 are very much the same as described in 6.3. However, the crucial difference is that in case of the optimized version, all these steps handle all tracks at once instead of being called once for every track. This is what essentially gives opportunity for vectorization.

1. Filter states: takes an array of states (a point described by x,y,z and the direction described by the tx,ty,1 vector) and outputs the array of states that pass through the UT detector.

2. Extrapolate states: takes the filtered states and extrapolates them to a specified z coordinate, writing out the x and y of the extrapolated state. The extrapolation is called four times, producing one output array for each of the four UT panels.

3. Gather hits: takes the extrapolated tracks and the partitioned array of hits and writes the hits that match with a state into the output array. Called once for each UT panel.

4. Find aligned hits: takes the gathered hits and partitioning information, and compiles a list of track candidates for each track. A candidate is a set of 3 or 4 UT hits that align to form a line that is nearly colinear to the track. A track may have zero or multiple candidates.

5. Fit candidates: estimates the path of the particle that produced the candidate's UT hits by parameters, taking as input the candidates and writing out said parameters.

6. Scoring candidates: from the estimated parameters of the candidates, creates a score based on how well the parameters approximate the path determined by the hits. The better the fit, the higher the score.

7. Select best candidates: for each track, find the candidate with the highest score and writes out an array of which the ith element contains the index of the best candidate for the ith track.

6.5.3 Vectorization of filtering

The filtering algorithm takes a set of States that describe the VELO tracks, and decides which ones are possible to be be extended into a VELO-UT track. Thanks to the restructuring of the overall data structures by my colleagues, the Velo-UT algorithm now only receives so called *forward tracks*ref to track types that point towards the UT. Consequently, the filtering must only remove tracks that fall out of the acceptance of the UT, that is, miss all the panels of the UT and cannot produce any UT hits.

Rephrasing the specification in computing terms, for the vector of a states, a vector of booleans has to be produced where each boolean indicates weather to keep the corresponding state, and then the vector of states needs to be collapsed to a new vector that only contains accepted states. Using SIMD, on can easily take 8 elements (on AVX ISA) from the rows of the states, thus do calculations on 8 states at once to produce 8 booleans. It is possible to efficiently leverage Intel's AVX2 instruction set to do the pruning of the 8 items at a good performance[10]. First, the items of the 8-vector are permuted in order to bring the selected elements to the front of the vector and move the unused items at the back. Second, an unaligned memory store operation writes the entire 8 vector to the output data structure at a specific offset. For the next write, the offset is increased by the popcount (number of the selected items), which means the next write will simply overwrite the unselected items of the current operation.

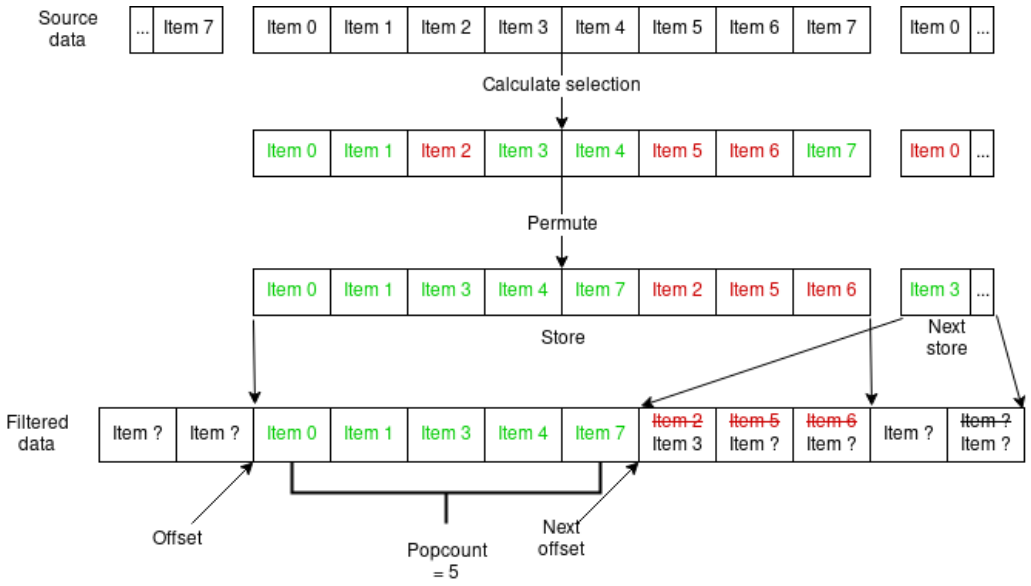


Figure 28: Visualization of the pruning step with 8-way SIMD

The large array of tracks are handled 8 at a time, producing the boolean mask, then doing the pruning, until the whole array is processed. The results of the filtering are stored in a format identical to the input states for consumption by later steps. Besides the states, an index array is also stored which contains the index of the selected states in the original table – this is necessary bookeeping to associate VELO track with their VELO-UT counterparts.

6.5.4 Vectorization of extrapolation and fitting

The extrapolation and fitting code operates on a large array of tracks, it is pure computation with no branching, and the tracks are completely independent – a perfect candidate for vectorization.

Extrapolation consumes full states that contain the position as well as the direction of the track, and writes out minimal states that contain only the x and y positions and the z of the UT panel we are extrapolating to. This makes sense, as tx and ty don't change, and the z is already known as it is tied to the panel, so the new x and y are enough. In case a subsequent process needs the remaining parameters, it can get it from the original array of states. This kind of split is an advantage of using SOA data layout. The implementation is a straight-forward loop that takes 8 tracks per iteration, does the extrapolation via vector registers, and writes out the 8 extrapolated tracks. The implementation of the fitter is in principle the same as the extrapolation, processing track candidates by batches of 8.

6.5.5 Vectorization of gathering hits

To recap, the gathering of hits must iterate over each extrapolated state, obtain the panel, row and bin indices for the state, and query the range of hits in that bin from the space partitioner. Then, it has to iterate over the range of hits and select those that are close enough to the track, and write the selected hits contiguously into the output array.

The problem can be split into two parts:

1. Getting panel, row and bin indices: as this step is the same, branchless pure computation (in theory) for each track, it can be perfectly vectorized.

2. Iterating over the range of hits: vectorization simply over tracks is not possible, compiling a contiguous array of all hit candidates of all tracks seems expensive (although best for vectorization), so what's left is vectorizing over the range of hits but going scalar over tracks. The problem is that the number of hits in a bin is less than would be ideal for vector register of width 8, so a lower efficiency if expected.

Getting bin indices

The x, y and z coordinates are given for each extrapolated state, the task is to figure out which grid cell they belong to. Given a uniformly partitioned axis, the bin index for a point can be calculated by evaluating

$$t = (x_{state} - x_{min})/(x_{max} - x_{min})$$
$$i = \lfloor t \cdot N \rfloor$$

where x_{min} and x_{max} are the limits of the binned range and N is the number of bins. The formula can be evaluated separately for the three axes, and it can be vectorized over 8 extrapolated states.

To obtain the offset and number of hits for a bin inside the partitioned array of UT hits, a lookup table that is indexed by the bin indices is available. The lookup can be either a scalar operation, or, as implemented, it can use the AVX gather instructions. For older CPUs, there is not much difference, gather instruction may be even slower, however recent Intel CPUs seem to perform better using gathers.add source One iteration thus returns two SIMD 8-vectors containing the offsets and the number of hits, which means it's completely vectorized.

Filtering the range of hits

As the hits inside the range are not ordered by for example their x coordinates, all of them have to be examined. A vectorized filtering loop described in 6.5.3 is employed to select nearby hits from the range. Unfortunately, the average number of hits in a bin are in the order of 5 to 15 (depending on grid resolution), so using a SIMD register width of 8 allows for only 1 or 2 iteration per loop, which is less than ideal.

Storage of this hits is similar to the method described in 6.5.3 with the difference that the output is one big array that contains the gathered hits for all tracks. Consequently, an array of offets is is also output which tells where to find gathered hits for a track in the aggregate array.

Double bins As described in 6.5.1, not a single bin but instead two adjacent bins are used. This changes the step for getting the bin indices minimally. Two lookups are performed, one for the actual bin index and another for the next or previous bin index, depending on which one the sampled coordinate is closer. (Double sampling only applies to bins, not row or panel indices.) The range is then defined by the offset of the first bin and the sum of the sizes of both bins. The hit filtering is unaffected, as it still takes an offset and a size, not caring where those come from.

6.5.6 Hit alignment search

The hit alignment search proved to be impractical to vectorize. The search algorithm operates on the nearby his gathered for a track. First, a pair of hits from two different panels are selected, a line is drawn through them, and the remaining two panels are searched for the hits closest to the line. In addition to the pair, an extra hit is accepted if it is withing a threshold to the line. In case 3 or 4 hits (including the pair itself) were found that fall on the line, the track candidate is written to the output.

One average, there are only 2 hits per track per panel, so vectorizing over one member of the pair, the other member of the pair, and the extra hits is not practical. Vectorizing over the tracks if not feasible either, as the number of hits in the four layers can be very different between the track, and branching would ruin the vectorization. Other attempts, such as producing all 3 or 4 hit combinations and then determining if they are on a line is also impractical, as the generation would either be scalar code or inefficient vectorized code.

6.5.7 Custom memory allocation

With the optimized version of the Velo-UT algorithm, up to 20% of the computation time was spent with memory allocation. The cost of memory allocation in an application depends most on the number of allocations, not the size of the allocations. Large allocations are much cheaper per byte allocated, so it is worth allocating an array for all objects upfront instead of allocating each object individually. The optimized Velo-UT algorithm uses this technique, however, the SOA data layouts require the allocation of a separate array for each data member of the structure. This combined with the fact that the algorithm generally works on a small amount of data makes memory allocations a significant problem.

Two options seems straightforward to reduce the number of memory allocations: allocating one large block of memory for all the contents of the SOAs, or use a memory pool instead of turning to the system memory allocator. The latter results in simpler code and make it possible to reuse existing code such as the `std::vector`. Additionally, the way I'm implementing SOA containers through code generation with template metaprogramming and `std::vector`s, extending my containers with custom allocators like the STL itself is more natural.

The idea is to have a fast memory pool that it initialized at the beginning of the algorithm and torn down at the end. All memory allocations for the steps in-between are served from the pool instead of the OS memory allocator. This is achieved by initializing all `std::vector`s with this memory pool. The choice fell on a stack memory allocator.

The stack memory allocator work by requesting a large chunk of memory upfront from the operating system. A variable which shows the `offset` from the beginning of the chunk is initialized to zero.

Figure 29: Initial state of the stack allocator.

When memory is requested from the pool, a pointer at the current `offset` is returned to the caller. Internally, the `offset` is incremented by the size of the requested block, plus any padding if necessary.

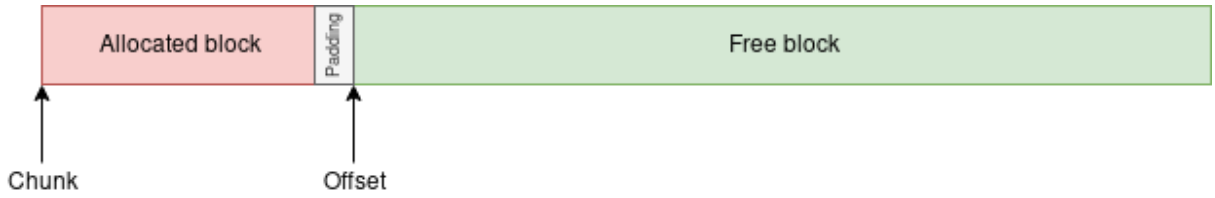


Figure 30: The allocator after the first allocation.

Subsequent allocations are performed similarly.

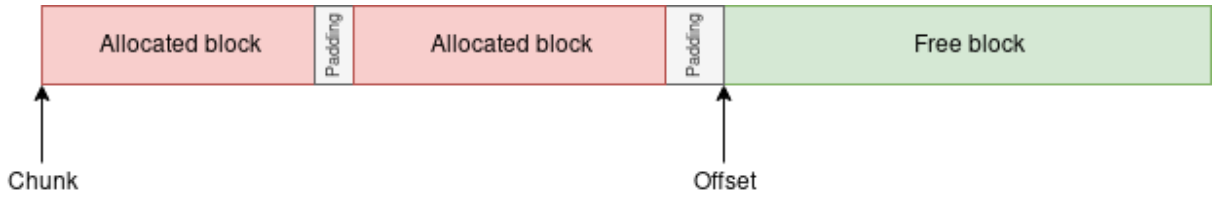


Figure 31: State of the allocator after a second allocation.

Stack allocators are wasteful, because previously requested blocks are never individually deallocated and their memory reused, rather, all the memory is deallocated at once when the memory pool is destroyed. This, however, is not a problem, since the entire Velo-UT algorithm requires about half a megabyte from a stack allocator, which amounts to 20-30 megabytes of memory when running on 40-60 threads simultaneously. That is still negligible compared to more than 32 gigabytes of RAM which these machines are equipped with.

In case the allocator would run out of space, a new empty block is requested from the operating system. Even though this case is handled, the goal is to minimize the number of requests to the OS, so the initial block size is tuned so as to fit all data of the algorithm.

Stack allocators are extremely fast, as all it takes to allocate a new memory block is to return the offset pointer and increment the offset – a couple of integer arithmetic instructions. Even though with the above extension an if statement handles the special case it runs out of free space, the if rarely ever actually takes this path so the CPU should be hitting near perfect rates for branch prediction.

6.6 Results

6.6.1 Overall performance

I acquired the results by running the code on a Haswell computer, looping over the same event for 100k iterations. The event of choice is a relatively busy event with many tracks. For the tests, the Velo-UT code was running alone, isolated from the rest of the reconstruction.

	GCC 9.1	Clang 8.0
Original	24157 ms	21241 ms
Optimized	9403 ms	9744 ms
Opt. w/o SP	6971 ms	7589 ms

Table 1: Runtimes of the different configurations.

As the space partitioning is a new workload that wasn't present in the original algorithm, timings are shown with it included. The space partitioning can be, however, loosely considered as a replacement to the construction of the `HitHandler`, so timings with it excluded are also worth mentioning.

Overall, the optimized algorithm is 2.57x faster with GCC and 2.18x faster with Clang. When the space partitioning is excluded, the numbers are 3.46x and 2.80x, respectively. These numbers fall short to what one would expect by using 8-way SIMD instruction, however they better represent reality. As previously mentioned, a significant portion of the code remained scalar as it was difficult or practically impossible to vectorize and some of the vectorized code suffers from inefficiencies due to small amount of data in the vectorized loops. Additionally, the original non-vectorized code gets auto-vectorized by the compilers, which further shifts the balance. Overall, the speedup if still significant and worthwhile.

6.6.2 Microarchitecture usage, bottlenecks

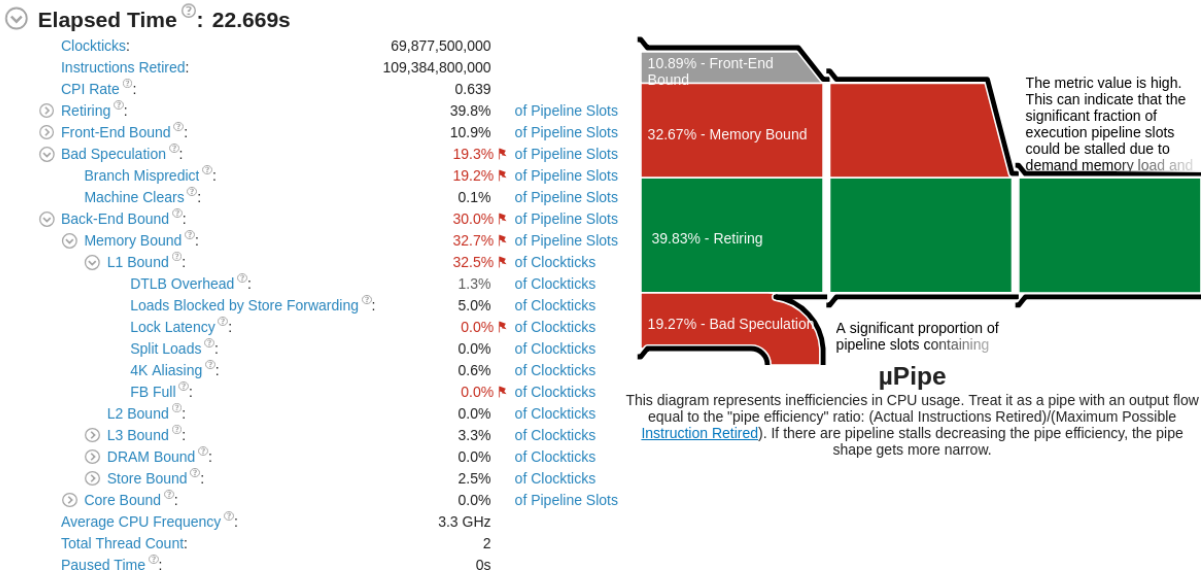


Figure 32: Microarchitecture usage of the original algorithm, results from Intel VTune.

The two most significant problems with the original algorithm are being bound by memory and by bad speculation.

Understanding bad speculation is straightforward, if the CPU guesses to go on the `true` branch of an `if` statement and queues the upcoming instructions of that branch, but it happens that the `if` executes the `false` branch, all the queued instructions have to be cancelled.[11] To alleviate this issue, one should remove branching altogether or, if not possible, make them more predictable.

As the profiler suggests, the code is mostly memory bound, more specifically, L1 cache bound. According to documentation from Intel, this metric shows up when the CPU is waiting for L1 load operations when the L1 cache was hit. This can happen, for example, when a load epends on an older store. To alleviate the problem, data dependencies can be rethought and the number of memory accesses per cycles spent with computation can be reduced.[11]

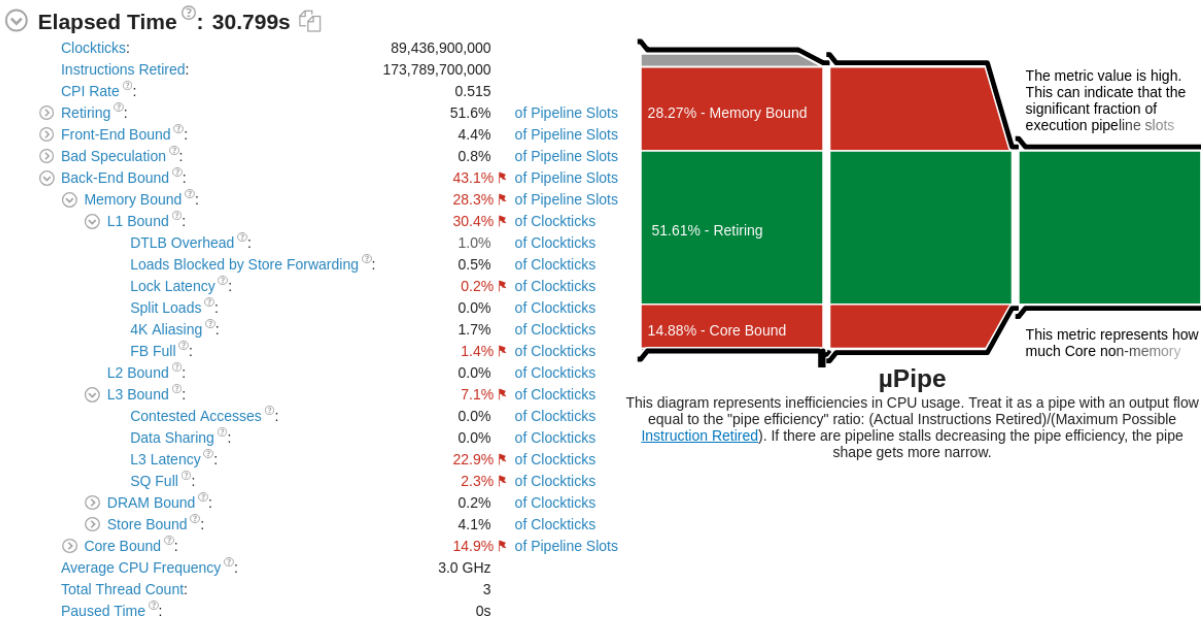


Figure 33: Microarchitecture usage of the optimized algorithm.

Despite the optimized version being 2.5 times faster, the microarchitecture usage pattern shows some resemblance.

The most notable difference is the complete absence of bad speculation problems. I made a good effort to remove branching from the code so that computing passes can iterate over the entire datasets without conditional code execution. One trick to mitigate branching is in the filtering loops (6.5.3 and 6.5.5) where masking and permuting of AVX registers is leveraged to emulate branching. Other tricks are, for example, the use of tiny branches which optimize out to CMOV instructions or the exploitation of boolean to integer conversion to obtain conditional numeric values by arithmetic instead of branching. The code for the fitting (6.5.4) needs to handle track candidates with either 3 or 4 hits, but instead of branching, the unused hit in the 3-hit-case is simply masked out of the computations by using a weight of zero. This incurs a very little penalty, as the weight is needed for the algorithm regardless, and the elimination of branches allows for perfect vectorization.

Unfortunately, the algorithm is still the most heavily bound by memory accesses. VTune highlights L1 memory accesses similarly to the unoptimized version, but it does not indicate any specific issue in connection with L1. I examined the code of the gathering of the hits for data dependencies that could cause high L1 latencies, but it does not seem to show any. On the other hand, the code needs to read a lot of data, and does minimal calculation on it in comparison. Though this case is not mentioned in the documentation by Intel, a toy example of adding two small arrays that fit in L1 using vectorized instructions results in a heavily memory bound program according to VTune. Based on that, it simply seems like L1 is unable to serve the execution units of the CPU with enough data when the operations they do are too simple and fast. Since the algorithm already uses SOA data layouts and vector register loads to read only the most necessary data in the largest possible chunks, this is most likely a characteristic of the algorithm.

In place of branch prediction, VTune shows the algorithm is core bound. VTune indicates that this problems is to a lesser extent caused by uveruse of the divider and to a larger extent by port utilization issues. The divider can be helped by using approximate divisions or factoring them out of loops, but the divisions were truly needed here. I tried to examine individual functions that are core bound due to port utilization, but I couldn't track down the source of the problem. The most promimnent core bound function was `ScanExtraHits`, which contains a chain of dependent FP instructions, as indicated as a problem by the documentation[11]. With GCC, which uses a fused multiply-add, the function is not core bound at all. Clang aggressively unrolls the loop even though it does two iterations on average, but does not use FMA, making the function core bound. When I forcefully disable loop unrolling, the function becomes even more core bound and the IPC dramatically decreases. It would need a significant amount of time to figure out and tune the function in isolation, which is not worth the effort.

```
134 inline LineExtraHit PrVeloUtOpt::ScanExtraHits(const GatheredHitsS& hits, uint32_t beginIndex,
        uint32_t endIndex, float x0, float dxdz, float maxDiff) {
135   float minDiff = maxDiff;
136   uint32_t index = 0;
137   for (size_t hitIdx = beginIndex; hitIdx < endIndex; ++hitIdx) {
138     const float xe = hits.x[hitIdx];
139     const float ze = hits.z[hitIdx];
140     const float xexp = x0 + dxdz * ze;
141     const float xdiff = std::abs(xexp - xe);
142     index = xdiff < minDiff ? hitIdx : index;
143     minDiff = xdiff < minDiff ? xdiff : minDiff;
144   }
145   return { minDiff, index };
146 }
```

Listing 5: Code of `ScanExtraHits`.

6.6.3 Effects of customized memory allocation

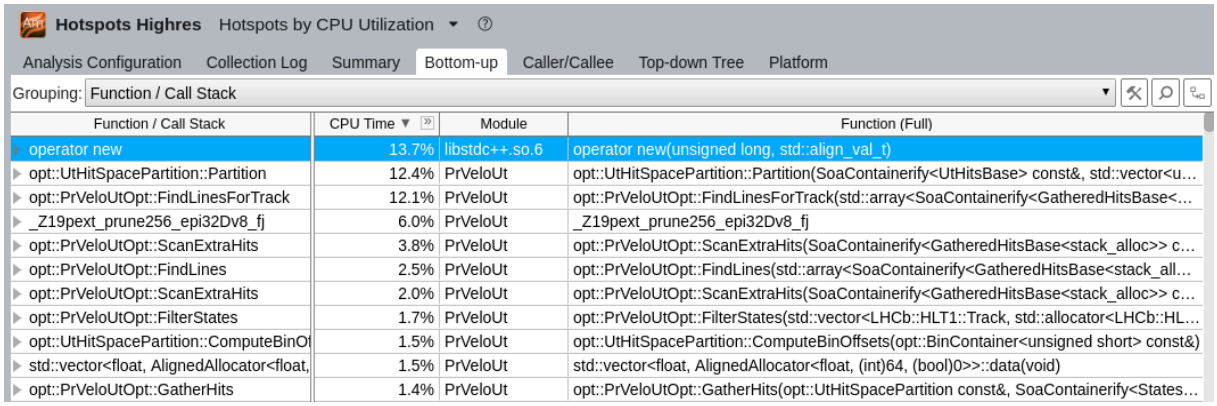


Figure 34: Hotspots profiling with VTune. Note the high amount of time spent in `operator new`.

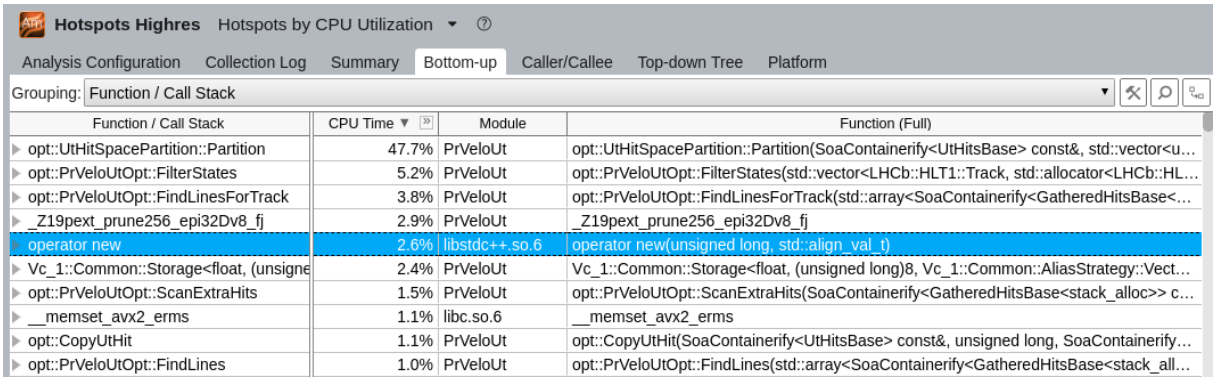


Figure 35: With the stack allocator, `operator new` has almost disappeared.

Figure 34 shows that without any special memory allocator, the program spends more than 13 percent in memory allocation. (Interestingly, the freeing of memory does not seem to take significant amount of computation.) However, thanks to the stack allocator, this can be reduced to less than 3 percent of the CPU time. The difference shown in the profiler manifests in the runtime as expected: 33.8 seconds with the default allocator compared to 29.1 seconds with the stack allocator, that is a gain of 14%.

6.6.4 Comparison of Clang and GCC

7 Conclusion

- summarize my own contributions
- summarize achieved results
- make conclusions about them
- how it affects the future
BRIEFLY

8 References

[1] About CERN:
https://home.cern/about

[2] The accelerator complex:
https://home.cern/about/accelerators

[3] About the Large Hadron Collider:
https://home.cern/topics/large-hadron-collider

[4] About the Large Hadron Collider beauty experiment:
https://home.cern/about/experiments/lhcb

[5] Why collide lead ions:
http://alicematters.web.cern.ch/?q=FAQ-why-lead-ions

[6] Energy of the LHC:
https://home.cern/about/engineering/restarting-lhc-why-13-tev

[7] LHC collisions:
https://lhc-machine-outreach.web.cern.ch/lhc-machine-outreach/collisions.htm

[8] LHC facts and figures:
https://public-archive.web.cern.ch/en/LHC/Facts-en.html

[9] Tracker (UT and FT) Technical Design Report:
https://cds.cern.ch/record/1647400?ln=en

[10] Pruning with AVX2:
https://github.com/lemire/simdprune

[11] Documentation of microarchitecture analysis by Intel:
https://software.intel.com/en-us/vtune-amplifier-help-core-bound
https://software.intel.com/en-us/vtune-amplifier-help-l1-bound
https://software.intel.com/en-us/vtune-amplifier-cookbook-top-down-microarchitec

9 Scratch

```
template <typename FudgeTable>
bool PrVeloUT::getHits( LHCb::span<UT::Mut::Hits, 4> hitsInLayers, const UT::HitHandler& hh,
                        const FudgeTable& fudgeFactors, MiniState& trState ) const {

  // -- This is hardcoded, so faster
  // -- If you ever change the Table in the magnet tool, this will be wrong
  const float               absSlopeY = std::abs( trState.ty );
  const int                 index     = (int)( absSlopeY * 100 + 0.5f );
  LHCb::span<const float, 4> normFact{&fudgeFactors.table()[4 * index], 4};

  // -- this 500 seems a little odd...
  const float invTheta =
      std::min( 500.0f, 1.0f * vdt::fast_isqrtf( trState.tx * trState.tx + trState.ty * trState.ty ) )
    ;
  const float minMom = std::max( m_minPT.value() * invTheta, m_minMomentum.value() );
  const float xTol   = std::abs( 1.0f / ( m_distToMomentum * minMom ) );
  const float yTol   = m_yTol + m_yTolSlope * xTol;

  int                                                   nLayers = 0;
  boost::container::small_vector<std::pair<int, int>, 9> sectors;

  for ( int iStation = 0; iStation < 2; ++iStation ) {

    if ( iStation == 1 && nLayers == 0 ) { return false; }

    for ( int iLayer = 0; iLayer < 2; ++iLayer ) {
      if ( iStation == 1 && iLayer == 1 && nLayers < 2 ) return false;

      const unsigned int layerIndex  = 2 * iStation + iLayer;
      const float        z           = m_layers[layerIndex].z;
      const float        yAtZ        = trState.y + trState.ty * ( z - trState.z );
      const float        xLayer      = trState.x + trState.tx * ( z - trState.z );
      const float        yLayer      = yAtZ + yTol * m_layers[layerIndex].dxDy;
      const float        normFactNum = normFact[layerIndex];
      const float        invNormFact = 1.0f / normFactNum;

      LHCb::UTDAQ::findSectors( layerIndex, xLayer, yLayer,
                                xTol * invNormFact - std::abs( trState.tx ) * m_intraLayerDist.value()
    ,
                                m_yTol + m_yTolSlope * std::abs( xTol * invNormFact ), m_layers[
    layerIndex], sectors );

      const LHCb::UTDAQ::SectorsInLayerZ& sectorsZForLayer = m_sectorsZ[iStation][iLayer];
      std::pair                           pp{-1, -1};
      for ( auto& p : sectors ) {
        // sectors can be duplicated in the list, but they are ordered
        if ( p == pp ) continue;
        pp                    = p;
        const int fullChanIdx = ( layerIndex * 3 + ( p.first - 1 ) ) * 98 + ( p.second - 1 );
        findHits( hh.hits( fullChanIdx ), sectorsZForLayer[p.first - 1][p.second - 1], trState, xTol *
     invNormFact,
                  invNormFact, hitsInLayers[layerIndex] );
      }
      sectors.clear();
      nLayers += int( !hitsInLayers[2 * iStation + iLayer].empty() );
    }
  }

  return nLayers > 2;
}
```

Listing 6: Hit gathering for the original Velo-UT

A Intel VTune Amplifier XE

B CPU pipelining

C Memory hierarchy of modern computers

D Inlining

E Vectorization

F LHCb track types