

Proceedings of the CTD 2022
PROC-CTD2022-29
LHCb-PROC-2022-012
July 10, 2023

Fast and flexible data structures for the LHCb Run 3 software trigger

ARTHUR HENNEQUIN

Massachusetts Institute of Technology, USA

MICHEL DE CIAN

École polytechnique fédérale de Lausanne, Switzerland

SEVDA ESEN

University of Zurich, Switzerland

On behalf of the LHCb Collaboration

ABSTRACT

Starting in 2022, the upgraded LHCb detector will collect data with a pure software trigger. In its first stage, reducing the rate from 30MHz to about 1MHz, GPUs are used to reconstruct and trigger on B and D meson topologies and high- p_T objects in the event. In its second stage, a CPU farm is used to reconstruct the full event and perform candidate selections, which are persisted for offline use with an output rate of about 10 GB/s. Fast data processing, flexible and custom-designed data structures tailored for SIMD architectures and efficient storage of the intermediate data at various steps of the processing pipeline onto persistent media, e.g. tapes, is essential to guarantee the full physics program of LHCb. In these proceedings, we will present the event model and data persistency developments for the trigger of LHCb in Run 3. Particular emphasize will be given to the novel software-design aspects with respect to the Run 1+2 data taking, the performance improvements which can be achieved and the experience of restructuring a major part of the reconstruction software in a large HEP experiment.

PRESENTED AT

Connecting the Dots Workshop (CTD 2022)
May 31 - June 2, 2022

1 The LHCb upgrade

In the last years, the LHCb detector[1] has been upgraded to run at a five times higher instantaneous luminosity (\mathcal{L}) than during Run 1+2 of the LHC, corresponding to $\mathcal{L} = 2 \cdot 10^{33} \text{ cm}^{-2}\text{s}^{-1}$. All tracking detectors and most of the readout electronics of the subdetectors have been replaced[2, 3, 4]. The change in hardware is also reflected by a change in the trigger strategy: for Run 3, starting in 2022, LHCb will use a pure software trigger, processing events at the 30 MHz non-empty bunch-crossing rate[5]. The first stage of the software trigger, HLT1, uses a set of GPU cards to perform a partial event reconstruction[6, 7, 8]. HLT2, the second stage of the software trigger, employs a farm of CPU servers to fully reconstruct the event with offline quality and perform an event selection, given an input rate of about 1 MHz and an output data rate of about 10 GB/s[9, 10].

In order to cope with the high data rate and the throughput requirements, the HLT2 event model was rewritten to use modern software paradigms such as SIMD (single instruction, multiple data) instructions. Its different building blocks and the performance will be explained in the following sections.

2 Introduction to the LHCb event model

The LHCb event model consists of all classes, implemented in C++, that represent the data flow from the detector raw banks to the charged and neutral particles used for data analysis. It is used to pass information between the algorithms in the reconstruction chain and to consistently write and read information from and to files. In Run 1+2 of the LHC, the LHCb event model used so-called “keyed containers” where every object in a container is identified by a key. These containers were implemented as array of structures (AOS) leading to slow data access in parallel-processing environments. Additionally, the keyed containers held pointers to objects they contained, making memory allocation and de-allocation slow.

For Run 3 of the LHC, the pure software trigger of LHCb required a redesign of the event model to reach the desired throughput. The new model stores the data in an Struct-of-Arrays (SOA) layout to be able to take advantage of SIMD (single input, multiple data) instructions on CPUs. The memory layout of AOS and SOA structures can be seen in Fig. 1.

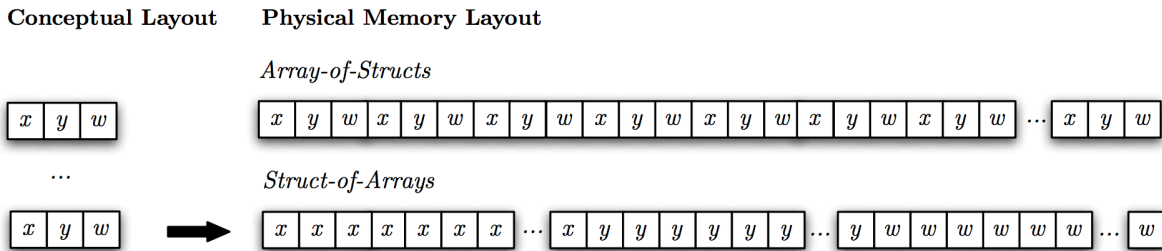


Figure 1: A comparison of AOS and SOA layouts. Taken from Ref. [11].

While developing the new event model, several key points for the event reconstruction, but also for the analysis of particle decays, have been taken into account. They include having flexible data structures that can be grown and shrunk at run time using dynamic memory allocation, but also the possibility of traversing decay trees for the analysis of multi-staged particle decays. In order to reach a high computational speed, the model needs to allow easy vectorisation[12, 13, 14, 15]. At the same time, the new model needs to be compatible with the old event model also during the development phase to not break the workflow of the full reconstruction sequence and for quality assurance.

3 SOA collections

An SOA collection is a dynamically-resizable collection of arrays in an SOA layout. Each array or field is represented by a tag which carries all the information about the field: its type, its packed representation for offline storage, etc... For example, a simple track* collection can be created with:

```
// Define tags:
struct Momentum : float_field {};
struct Index : int_field {};
struct LHCbID : lhcbid_field {};
struct Hits : vector_field<struct_field<Index, LHCbID>> {};

// Define collection:
struct Tracks :
    SOACollection <Tracks, Momentum, Hits>{};
```

with `Momentum` the absolute momentum of the track, `LHCbID` a unique identifier for a charged cluster on a track, `Index` the index of the LHCbID on the track and `Hits` a class representing the collection of charged clusters on the track. The design goal of SOA collections is to provide a user friendly structure, replacing an AOS structure such as `std::vector<Track>`, that allows for efficient vectorization.

Access to the individual tags is provided via proxies, where the specific SIMD[†] or scalar backends can be chosen at compile time, with an automated detection of the largest vector width available on the specific architecture. A proxy therefore represents a chunk of N objects in the collection where one object, e.g. a track, is a slice through the collection.

Elements to the collection can be easily added to the end, similarly to a `std::vector`, with the possibility of masking some elements, i.e. not actually adding them. This allows for selecting some objects while discarding others in parallel, e.g. when applying track quality or momentum requirements.

```
// Push N elements to the end of tracks, masking some
// Set the momentum of the track
auto proxy = tracks.emplace_back <simd>(mask);
proxy.field<Momentum>().set(momentum);

// Iterate over tracks N elements at a time
for (const auto& proxy : tracks.simd())
    auto momentum = proxy.get<Momentum>();
```

The same operations in scalar:

```
// Push 1 element to the end of tracks, possibly masking it
// Set the momentum of the track
auto proxy = tracks.emplace_back <scalar>(mask);
proxy.field<Momentum>().set(momentum);

// Iterate over tracks one at a time
for (const auto& proxy : tracks.scalar())
    auto momentum = proxy.get<Momentum>();
```

*A track represents the trajectory of a charged particle.

[†]SIMD is used in the following for all architectures which provide a vector width larger than 1.

4 Connecting SOA Collections

4.1 Zipping

In the transient event store (TES)[16], which is used to pass objects from one algorithm to the next, data objects need to be constant to allow safe memory allocation for multi-threading. However during the event reconstruction, more information can become available for some objects which are already in the TES. For example, after tracks are reconstructed, particle identification (PID) algorithms are executed, providing additional information for these tracks. Instead of making a copy of the objects in the TES, two methods can be used to connect the new information to the original object. The first is using ‘zipping’, which is similar to python `zip()`. A zip is a set of SOA collections of the same size that can be iterated as one and carries the information on how to iterate and access the collection, i.e. the actual SIMD backend and the proxy behaviour. An example of a possible zip between tracks and PID information can be seen in Fig. 2. Zips only keep pointers to existing containers and do not own any memory. An (example) zip with tracks and PIDs can be created with and iterated over with:

```
auto zipped = make_zip<simd>(tracks, PIDs);
for (const auto& zipproxy : zipped).simd() {
    auto momentum = zipproxy.get<Momentum>(); // from tracks
    auto pid = zipproxy.get<pid>(); // from PIDs
}
```

The fact that the code for looping over an SOACollection or a zip of SOACollections is identical leads to increased code flexibility.

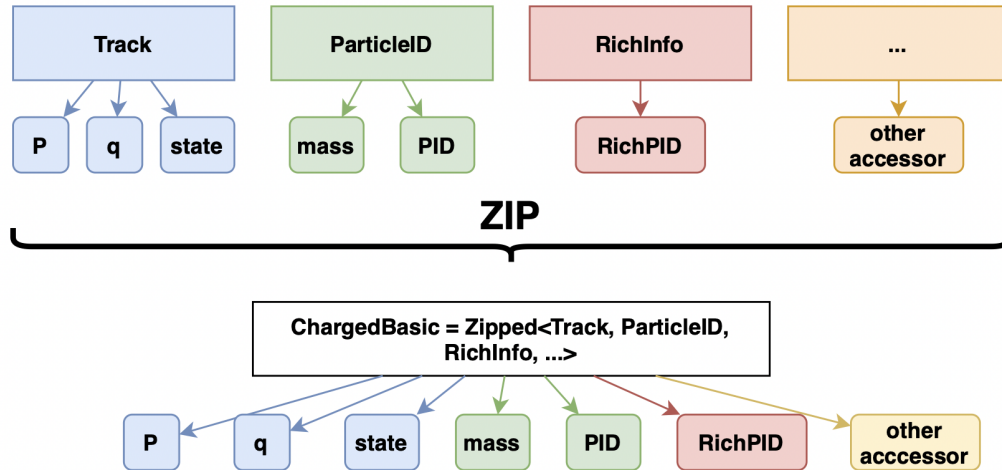


Figure 2: An example zip combining track, particle ID and RICH PID to a charged particle. Taken from Ref. [17].

4.2 Relation tables

Zipping only works if both SOA collections have the same size and there exists a one-to-one correspondence between the individual entries in the SOA collections. However, there are situations where this is not the case, i.e. two tracks could both point to the same calorimeter cluster.

The second method to add information to an existing object are therefore ‘relations’. Relations connect elements in a collection to something else, which can be another collection. An additional weight information can be added to each relation. SOA Relations are SOA Collections representing relations between two SOA Collections. For example a relation can be used between particles and their primary vertices with:

```

struct TracksPVsRelWithWeight:
    RelationTable2D<Tracks, PVs, Weight>{};
TracksPVsRelWithWeight table {tracks, pvs};
auto proxy = table.emplace_back<simd>();
proxy.set(tracks.indices(), pvs.indices(), weight);

```

5 SIMD Wrappers

The efficient use of SIMD instructions relies on using “intrinsic” for vector operations, which depend on the architecture and instruction set used (x86, ARM; SSE, AVX). In order to allow for a consistent use of vector operations, an easy switching between the backends and a more familiar look-and-feel similar to the scalar instructions formerly used in the LHCb code, wrapper classes for commonly used intrinsics, called `SIMDWrapper` were introduced at LHCb[18]. The instruction set is fixed at compile time, by selecting an architecture using compiler flags and target, to allow the compiler to do more optimizations. Given that changing the architecture during runtime is unlikely, this limitation does not have a negative impact for the LHCb software. The wrapper is fully integrated into the LHCb software and templated when possible to have only one implementation for all backends. Also common math functions and matrix operations are defined for all architectures to allow easy switching from one to another. An example for the function to find the minimum is given below:

```

// scalar
scalar::float_v min( scalar::float_v lhs, scalar::float_v rhs ) {
    return std::min( lhs.data, rhs.data );
}
// neon
neon::float_v min( neon::float_v lhs, neon::float_v rhs ) {
    return vminq_f32( lhs, rhs );
}
// avx
avx::float_v min( avx::float_v lhs, avx::float_v rhs ) {
    return _mm256_min_ps( lhs, rhs );
}

```

with `scalar::float_v` a float with vector width one, `neon::float_v` a float on the ARM architecture, `vminq_f32` the function to find the minimum between two ARM float numbers, `avx::float_v` a float in the AVX instruction set and `_mm256_min_ps` the function to find the minimum between two AVX float numbers.

6 Throughput Oriented (ThOr) selections

In Run 2, the event reconstruction at LHCb was about 70% and the selections were about 30% of the time spent in HLT2. This is expected to be similar in Run 3. Currently, more than 1000 exclusive HLT2 lines are being tested, each performing selections (cuts, vertex fitting, combinations etc...) on basic particles. In order to benefit from the speed improvement provided by SIMD instructions and the usage of SOA collections also in selections, a new framework was developed simultaneously for the old and the new SOA-based event models.

In order to select interesting decays in trigger lines, functors (i.e. function objects) are used. The so-called Throughput Oriented (ThOr) functors, are designed to be agnostic about the input and output type to be flexible on what they operate on. A significant gain in speed is achieved when using SIMD instructions on SOA containers compared to the old implementation as seen in Table 1. Additional speed is gained by using a functor cache instead of Just-In-Time compilation: This means that functors, which are defined in `python`, are compiled into a cache during the build process to be then used directly in the application without further interpretation. To simplify user experience, functors are templated and are using `SIMD` wrappers, so the code is the same for every architecture and no specialization is needed at the functor level.

Implementation	$D^+ \rightarrow K^+\pi^+\pi^-$ execution time
CombineParticles	256 μ s
NBodyDecays	77.1 μ s
ThorParticleCombiner	38.8 μ s
ThOrCombiner Scalar	10.2 μ s
ThOrCombiner SSE	7.5 μ s
ThOrCombiner AVX2	6.9 μ s

Table 1: A benchmark comparing the timing of the reconstruction of a $D^+ \rightarrow K^+\pi^+\pi^-$ particle decay, using different algorithms to perform the particle combination. CombineParticles and NBodyDecays use the legacy framework from Run 1+2; ThorParticleCombiner uses functors, but still the old data structures; ThOrCombiner uses SOA structures with different vector widths. Taken from Ref. [17].

7 Persistency

The final step of the event reconstruction and the selection of candidates is the persistency of the data for future use. In the AOS event model, this is done in two steps: filtering what needs to be persisted and creating persistent representations i.e., conveying the data to more basic data structures. The SOA collections are already mostly in a format that is ready to be persisted so the second step is simpler. While creating the collection, each tag can be customized for how and if it is persisted and versioning can be introduced. For the example below, one field is defined to be packed as float, one field is not to be persisted at all, and one field is to be persisted only for the newest versions of the collection.

```
// Define tags :
struct Momentum : float_field {
using packer_t = SOAPackFloatAs <short,
                                std::ratio<1, 100 > >;
};
struct Unwanted : int_field {
    using packer_t = SOADontPack ;
};
struct OpsIForgotThisField : int_field {
using packer_t =
    SOAPackIfVersionNewerOrEqual<1, SOAPackNumeric <int>>;
};
// Define collection :
struct Tracks : SOACollection<Tracks,Momentum,
                             Unwanted, OpsIForgotThisField> {};
```

8 Throughput improvements in the HLT1 prototype sequence

A prototype for the HLT1 trigger was implemented on CPU in parallel to the GPU prototype[19]. It featured most of the improvements explained in these proceedings, most importantly the SOA Collection and a widespread use of SIMD instructions. The HLT2 trigger uses same event model as the HLT1 and benefits from the same improvements mentioned in these proceedings.

In order to test the impact of using SIMD instructions compared to scalar instructions, the sequence was once run with the AVX2 instruction set, and once with scalar instructions, while keeping the event model the same. The throughput was about twice when using vector instructions compared to scalar instructions.

A historical overview over the improvements achieved using the new event model, using SIMD instructions and having improved reconstruction algorithms can be seen in Fig. 3. It shows that thanks to a concentrated

effort on all three points, the throughput could be improved by about a factor 4, without compromising the reconstruction of physics quantities.

9 Conclusions

For Run 3 of the LHC, the LHCb collaboration implemented a new event model for the second stage of the software trigger, HLT2, using an SOA layout, native usage of SIMD instructions and more flexibility. This results in an increased throughput and allows to run more than 1000 trigger lines with a full offline-quality reconstruction, without the need for any post-processing. This event model therefore is well suited for the coming decade of data taking of the LHCb experiment.

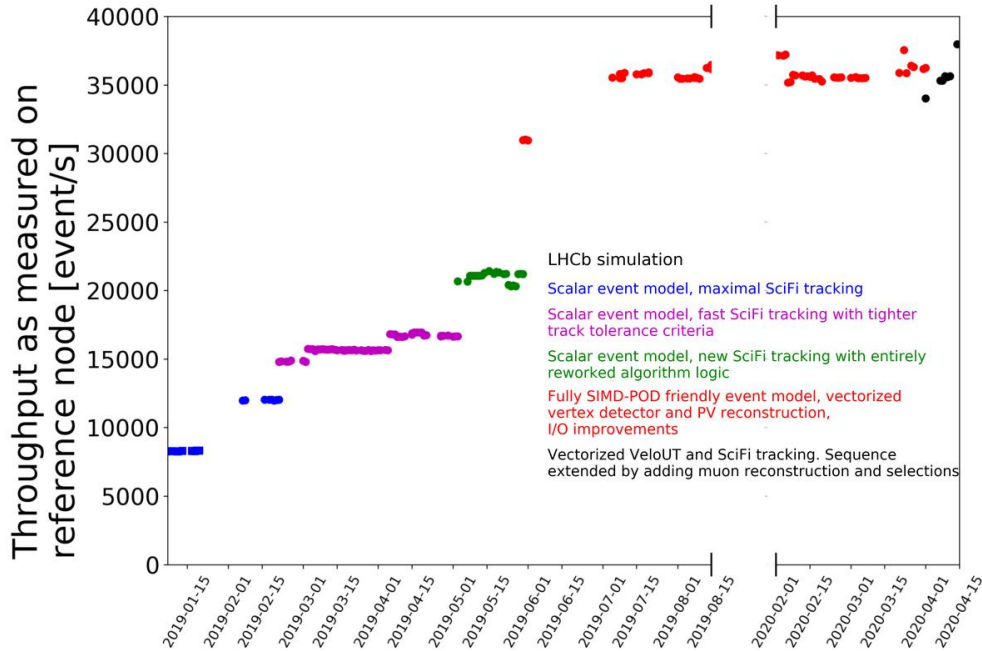


Figure 3: Evolution of the upgrade LHCb HLT1 throughput of the CPU prototype between December 2018 and April 2020. The period between August 2019 and February 2020 has been cut out as there were little changes to the throughput. More details about the indicated optimizations can be found in Ref. [20].

ACKNOWLEDGEMENTS

The authors would like to thank the LHCb computing, simulation and RTA teams for their support and for producing the simulated LHCb samples used for these proceedings. We also would like to thank the organisers of the CTD conference for the interesting workshop, nice location and the copious amount of delicious cookies.

M. De Cian acknowledges support from the Swiss National Science Foundation grant “Probing right-handed currents in quark flavour physics”, PZ00P2.174016. S. Esen acknowledges funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101027131.

References

- [1] A. Augusto Alves, Jr. et al. The LHCb Detector at the LHC. *JINST*, 3:S08005, 2008. <https://doi.org/10.1088/1748-0221/3/08/s08005>.
- [2] Roel Aaij et al. LHCb Tracker Upgrade Technical Design Report. Technical Report CERN-LHCC-2014-001. LHCb-TDR-015, Feb 2014. <https://cds.cern.ch/record/1647400>.
- [3] LHCb VELO Upgrade Technical Design Report. Technical Report CERN-LHCC-2013-021. LHCb-TDR-013, Nov 2013. <https://cds.cern.ch/record/1624070>.
- [4] LHCb PID Upgrade Technical Design Report. Technical Report CERN-LHCC-2013-022, CERN, Geneva, 2013. <https://cds.cern.ch/record/1624074>.
- [5] LHCb Trigger and Online Upgrade Technical Design Report. Technical Report CERN-LHCC-2014-016. LHCb-TDR-016, May 2014. <http://cds.cern.ch/record/1701361>.
- [6] LHCb Collaboration. LHCb Upgrade GPU High Level Trigger Technical Design Report. Technical report, CERN, Geneva, May 2020. <https://cds.cern.ch/record/2717938>.
- [7] Thomas Boettcher. Allen in the first days of Run 3. Aug 2022. <https://cds.cern.ch/record/2823780>.
- [8] Alessandro Scarabotto. Tracking on GPU at LHCb's fully software trigger. Aug 2022. <http://cds.cern.ch/record/2823783>.
- [9] LHCb Collaboration. Computing Model of the Upgrade LHCb experiment. Technical report, CERN, Geneva, May 2018. <https://cds.cern.ch/record/2319756>.
- [10] Paul Andre Günther. LHCb's Forward Tracking algorithm for the Run 3 CPU-based online track reconstruction sequence. Jul 2022. <http://cds.cern.ch/record/2819858>.
- [11] Susan M. Mniszewski et al. Enabling particle applications for exascale computing platforms. *The International Journal of High Performance Computing Applications*, 35(6):572–597, jul 2021. <https://doi.org/10.48550/arXiv.2109.09056>.
- [12] Florian Lemaitre and Lionel Lacassagne. Batched Cholesky Factorization for tiny matrices. In *Design and Architectures for Signal and Image Processing (DASIP)*, pages 1–8, Rennes, France, October 2016. ECSI. <https://hal.archives-ouvertes.fr/hal-01361204>.
- [13] Florian Lemaitre, Benjamin Couturier, and Lionel Lacassagne. Cholesky Factorization on SIMD multi-core architectures. *Journal of Systems Architecture*, June 2017. <https://hal.archives-ouvertes.fr/hal-01550129>.
- [14] Florian Lemaitre, Benjamin Couturier, and Lionel Lacassagne. Small SIMD Matrices for CERN High Throughput Computing. In *WPMVP 2018 Workshop on Programming Models for SIMD/Vector Processing*, Vienna, Austria, February 2018. ACM Press. <https://hal.archives-ouvertes.fr/hal-01760260>.
- [15] A. Hennequin et al. A fast and efficient SIMD track reconstruction algorithm for the LHCb upgrade 1 VELO-PIX detector. *Journal of Instrumentation*, 15(06):P06018–P06018, jun 2020. <https://doi.org/10.1088/1748-0221/15/06/p06018>.
- [16] G Barrand, I Belyaev, P Binko, M Cattaneo, R Chytracsek, G Corti, M Frank, G Gracia, J Harvey, Eric Van Herwijnen, B Jost, I Last, P Maley, P Mato, S Probst, F Ranjard, and A Yu Tsaregorodtsev. GAUDI: The software architecture and framework for building LHCb data processing applications. 2000.
- [17] Niklas Nolte. A Selection Framework for LHCb's Upgrade Trigger, Dec 2020. <https://cds.cern.ch/record/2765896>.

- [18] Arthur Hennequin. *Performance optimization for the LHCb experiment*. Theses, Sorbonne Université, January 2022. <https://tel.archives-ouvertes.fr/tel-03640612>.
- [19] R. Aaij et al. A Comparison of CPU and GPU Implementations for the LHCb Experiment Run 3 Trigger. *Comput. Softw. Big Sci.*, 6(1):1, 2022. <https://doi.org/10.48550/arXiv.2105.04031>.
- [20] Throughput and resource usage of the LHCb upgrade HLT. Apr 2020. <https://cds.cern.ch/record/2715210>.