Event and data persistency models for the LHCb Real Time Analysis System

Michel De Cian[1,*], *Sevda* Esen[1,**], *Arthur* Hennequin[2,***], and *Xavier* Vilasís-Cardona[3,****]

[1]Physikalisches Institut, Ruprecht-Karls-Universität Heidelberg, Heidelberg, Germany;
[2]CERN;
[3]La Salle, Universitat Ramon Llull, Spain.

Abstract. Starting in 2022, the upgraded LHCb detector is collecting 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-pT 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 10GB/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. We present the event model and data persistency developments for the trigger of LHCb in Run 3. Particular emphasis is 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.

1 Introduction

The configuration of the LHCb detector for the LHC Run 3 [1] includes the readout of all subdetectors at the nominal bunch-crossing rate of the LHC of 40MHz. In terms of the non-empty bunch crossing rate of 30MHz, this amounts to a data flow of 5Tb.s^{-1} [2]. The full event data is processed by a two-staged trigger system following the Real Time Analysis (RTA) paradigm, shown in figure 1. The first stage performs a partial reconstruction and initial selections based on a GPU platform called Allen [3]; it plays the role of a first high level trigger (HLT1) and reduces the data rate to less than 200Gb.s^{-1} [3–5]. The data is then put to a buffer, where it is used to perform alignment and calibration tasks. The second stage is a second High Level Trigger (HLT2) running on a CPU farm. It performs a full reconstruction and the appropriate selections cutting the data rate down to 10Gb.s^{-1} [6, 7]. Stored data corresponds to a small fraction of calibration events and some full events that will follow an offline processing, but the largest part corresponds to fully reconstructed events with off-line reconstruction quality, following the turbo model already used in LHCb in RUN2 [8].

[*]e-mail: michel.de.cian@cern.ch
[**]e-mail: sevda.esen@cern.ch
[***]e-mail: arthur.hennequin@cern.ch
[****]e-mail: Xavier.Vilasis.Cardona@cern.ch

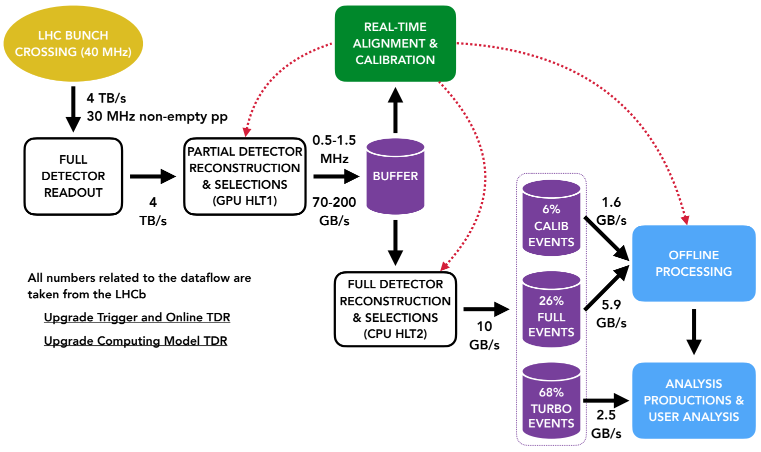
LHC BUNCH CROSSING (40 MHz)
4 TB/s
30 MHz non-empty pp
FULL DETECTOR READOUT
4 TB/s
PARTIAL DETECTOR RECONSTRUCTION & SELECTIONS (GPU HLT1)
0.5-1.5 MHz
70-200 GB/s
BUFFER
REAL-TIME ALIGNMENT & CALIBRATION
FULL DETECTOR RECONSTRUCTION & SELECTIONS (CPU HLT2)
10 GB/s
6% CALIB EVENTS
26% FULL EVENTS
68% TURBO EVENTS
1.6 GB/s
5.9 GB/s
2.5 GB/s
OFFLINE PROCESSING
ANALYSIS PRODUCTIONS & USER ANALYSIS
All numbers related to the dataflow are taken from the LHCb
Upgrade Trigger and Online TDR
Upgrade Computing Model TDR

Figure 1. LHCb Data flow [3]

Given the restrictions of the time budget in order to meet the data rate requirements and the need for persistency of the stored data for future use, the event model has been rewritten using modern software paradigms such as Single Instruction Multiple Data (SIMD). This note describes the building blocks and illustrates its performance following reference [9].

2 The LHCb Event Model

The LHCb event model is the set of classes that represent the data flow from the detector raw banks to the particles used for data analysis. It is implemented in C++ for code efficency criteria. It serves to pass information between the algorithms in the reconstruction chain and to write and read information consistently from and to files. In the current trigger framework, it is applied in HLT2. The LHCb event model for Run 1 and Run 2 was based on containers where every object in a container was identified by a key. These containers followed an Array of Structures (AOS) format. This format is however not suited for parallel-processing environments and slows down the access to data. Additionally, the keyed containers held pointers to objects they contained, making memory allocation and de-allocation slow.

Because of the aforementioned requirements in terms of computing efficiency and data rate, both in terms of event reconstruction and the analysis of particle decays, the event model had to be redesigned for Run 3. The key elements in the process included 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 needed to allow easy vectorisation [10–13]. At the same time, the new model had 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.

All these features would be met by a Struct-of-Arrays (SOA) model [14], which was taken as the reference. Figure 2 shows the differences in the memory layout between the previous AOS model and the current SOA one.

The SOA structures are then organized as SOA collections. These are dynamically resizeable collections 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

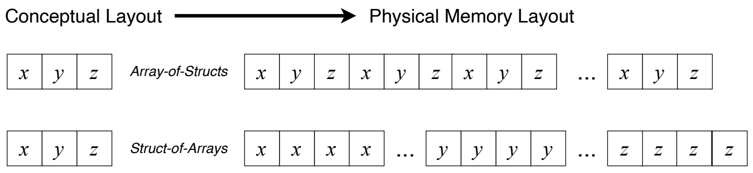

Figure 2. A comparison of AOS and SOA memory layouts. Taken from Ref. [14]

storage, and so on. To illustrate the collections, we shall use the representation of the trajectories of charged particles, known as tracks. 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>{};
```

being `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. SOA collections provide a user friendly structure, replacing AOS structure such as `std::vector<Track>`, and allow for efficient vectorization. Access to the individual tags is provided via proxies, where the specific SIMD or scalar backends can be chosen at compilation 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 in 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, such as applying track quality or momentum requirements. For operations in the track case, this would look like,

```
// 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>();
```

while 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>();
```

3 Connecting SOA collections

In LHCb, the Transient Event Store (TES) [15] 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.

3.1 Zipping

The first approach is using a method similar to python `zip()` that is denoted by *zipping*. 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, that is, the actual SIMD backend and the proxy behaviour. Figure 3 illustrates an example of 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.

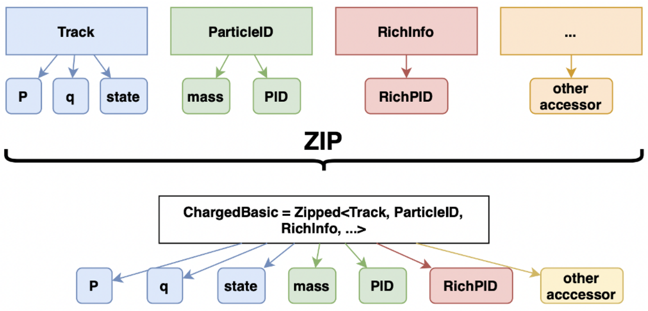

Figure 3. An example zip combining track, particle ID and RICH PID to a charged particle. Taken from Ref. [16]

The code to create the zip from Figure 3 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 }
```

This approach permits an increased code flexibility since the code for looping over an SOA-Collection or a zip of SOACollections is identical. 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.

3.2 Relation tables

To tackle situations in which there is no one-to-one correspondence between the individual entries in the SOA collections, like, for instance, two tracks could both point to the same calorimeter cluster, another approach is required. This second method to add information to an existing object are the *relations*. Relations connect elements in a collection to another object, 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. An example of a relation is the one between particles and their primary vertices.

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

4 SIMD wrappers

One of the reasons for the LHCb event model redesign is the possibility to use SIMD instructions in the processors. Their efficiency relies on using *intrinsics* for vector operations, which depend on the architecture and instruction set used (x86, ARM; SSE, AVX). Wrapper classes for commonly used intrinsics, called *SIMDWrapper* were introduced at LHCb [17]. These 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. The instruction set is fixed at compilation 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 mathematical functions and matrix operations are defined for all architectures to allow easy switching from one to another. The code below shows, for example, the function to find the minimum.

```
    // 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 );
}
```

Here, `scalar::float_v` is 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.

Table 1. A benchmark comparing the timing of the reconstruction of a $D^+ \to K^+\pi^+\pi^-$ 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 [16].

Implementation	$D^+ \to K^+\pi^+\pi^-$ execution time
CombineParticles	256μs
NBodyDecays	77.1μs
ThOrParticleCombiner	38.8μs
ThOrCombiner Scalar	10.2μs
ThOrCombiner SSE	7.5μs
ThOrCombiner AVX2	6.9μs

5 Throughput Oriented selections

Based on Run 2 figures and on estimations, HLT2 timing budget is split roughly 70% in event reconstruction and 30% in selections. Currently, almost 2000 exclusive HLT2 lines are being tested, each performing selections 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 has been developed for the old and the new SOA-based event models using functors (function objects). 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 may be gained by using a functor cache instead of Just-In-Time compilation: 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 SIMDwrappers, so the code is the same for every architecture and no specialization is needed at the functor level.

6 Persistency

The persistency for future use of the data after the event reconstruction and the selection of candidates is one of the key elements in the design of the event model. Persistency is ensured in the AOS framwork in two steps. First, filtering what needs to be persisted and second, creating persistent representations, in other words, conveying the data to more basic data structures. The SOA collections are already mostly in a format that is ready to be persisted making the second step simpler. Tags can be customized for how and if it is persisted and versioning can be introduced at the creation of the collection. In the example below, one field is defined to be packed as float, one field is not to be persisted, 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 OopsIForgotThisField : int_field {
using packer_t =
    SOAPackIfVersionNewerOrEqual<1, SOAPackNumeric <int>>;
};
// Define collection :
struct Tracks : SOACollection<Tracks,Momentum,
                              Unwanted, OopsIForgotThisField> {};
```

7 Conclusions

For the Run 3 of the LHC, the LHCb collaboration has implemented a new event model for the second stage of the software trigger, HLT2. Its key features are the use of a SOA layout, the native usage of SIMD instructions and SIMD wrapper classes to encapsulate common intrinsics. The result is an increased throughput that allows to run about 2000 trigger lines with a full offline-quality reconstruction, without the need for any post-processing. On one hand, this permits a large flexibility in the physics program, and, on the other, lowers the computing resources needed of the experiment by eliminating a good part of the post-processing. An illustration of the improvement of the model can be found in Ref. [9, 18] where the efficiency of using SIMD acceleration on different algorithms is shown. This event model therefore is well suited for the coming decade of data taking of the LHCb experiment.

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. 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 Sklodowska-Curie grant agreement No. 101027131. X. Vilasís acknowledges support by Spanish MICINN under Grant PID2019-106448GB-C32 and by Generalitat de Catalunya under Grant 2021 SGR 01398.

References

[1] LHCb collaboration, *Framework TDR for the LHCb Upgrade: Technical Design Report*, CERN-LHCC-2012-007.

[2] LHCb collaboration, *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.

[3] LHCb Collaboration, *LHCb Upgrade GPU High Level Trigger Technical Design Report.* Technical report, CERN, Geneva, May 2020. https://cds.cern.ch/record/2717938.

[4] Thomas Boettcher, *Allen in the first days of Run 3.* Aug 2022. https://cds.cern.ch/record/2823780.

[5] Alessandro Scarabotto, *Tracking on GPU at LHCbś fully software trigger.* Aug 2022. http://cds.cern.ch/record/2823783.

[6] LHCb Collaboration, *Computing Model of the Upgrade LHCb experiment*. Technical report, CERN, Geneva, May 2018. https://cds.cern.ch/record/2319756.

[7] Paul Andre Günther, *LHCbś Forward Tracking algorithm for the Run 3 CPU-based online track reconstruction sequence.* Jul 2022. http://cds.cern.ch/record/2819858.

[8] Sean Benson, et al. *The LHCb turbo stream* Journal of Physics: Conference Series. **664** No. 8. IOP Publishing, 2015.

[9] Esen, Sevda, Arthur Marius Hennequin, and Michel De Cian. *Fast and flexible data structures for the LHCb Run 3 software trigger.* in David Lange. (2023, July 6). Proceedings of the 2022 Connecting the Dots Workshop. https://doi.org/10.5281/zenodo.8119864 .

[10] 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.

[11] Florian Lemaitre, Benjamin Couturier, and Lionel Lacassagne, *Cholesky Factorization on SIMD multi- core architectures.* Journal of Systems Architecture, **79** June 2017. https://hal.archives-ouvertes.fr/hal-01550129.

[12] 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.

[13] 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.

[14] 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.

[15] G Barrand, I Belyaev, P Binko, M Cattaneo, R Chytracek, 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.* LHCb-TALK-2000-002. 2000.

[16] Niklas Nolte, *A Selection Framework for LHCbś Upgrade Trigger.* PhD thesis - Dec 2020. https://cds.cern.ch/record/2765896.

[17] Arthur Hennequin. *Performance optimization for the LHCb experiment.* Theses, Sorbonne Université, January 2022. https://tel.archives-ouvertes.fr/tel-03640612.

[18] 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.