SIMD studies in the LHCb reconstruction software

Daniel Hugo C´ ampora P´ erez, Ben Couturier
CERN, PH Department. Geneva, Switzerland
E-mail: dcampora@cern.ch

Abstract. During the data taking process in the LHC at CERN, millions of collisions are recorded every second by the LHCb Detector. The LHCb Online computing farm, counting around 15000 cores, is dedicated to the reconstruction of the events in real-time, in order to filter those with interesting Physics. The ones kept are later analysed Offline in a more precise fashion on the Grid. This imposes very stringent requirements on the reconstruction software, which has to be as efficient as possible.

Modern CPUs support so-called vector-extensions, which extend their Instruction Sets, allowing for concurrent execution across functional units. Several libraries expose the Single Instruction Multiple Data programming paradigm to issue these instructions. The use of vectorisation in our codebase can provide performance boosts, leading ultimately to Physics reconstruction enhancements.

In this paper, we present vectorisation studies of significant reconstruction algorithms. A variety of vectorisation libraries are analysed and compared in terms of design, maintainability and performance. We also present the steps taken to systematically measure the performance of the released software, to ensure the consistency of the run-time of the vectorised software.

1. Introduction

The Single Instruction Multiple Data (SIMD) paradigm [1] allows CPUs to operate on several operands simultaneously, by only issuing one instruction at a time. This paradigm optimizes the occupancy of execution units, and leaves it to the programmer to prepare the data structures and design algorithms properly to fully take advantage of it.

Intel first introduced Streaming SIMD Extensions (SSE) in 1999 as a set of 128-bit registers and an extension to the x86 instruction set, also known as vector units [2]. Further iterations of their vectorisation technology have brought wider registers, namely 256-bit ones with AVX, and 512-bit ones with AVX-512, as well as additional instructions allowing greater control over these registers.

At LHCb, particle collision data known as events are collected at a designed bunch crossing rate of 40 MHz. A selection is performed over these data in a real time process known as the online reconstruction, where uninteresting events are filtered out, bringing the rate down to affordable levels for the data collection stage. The online reconstruction is a two-stage process consisting of a Hardware Level Trigger selection on FPGA boards, and a High Level Trigger (HLT) performed over a farm counting around 15000 cores [3].

The High Level Trigger is processed by a chain of algorithms that reconstructs the data from each subdetector, and operates on it to perform the event selection. Two concrete subdetectors are the VErtext LOcator (VELO) and the Ring Imaging CHerenkov subdetectors (RICH), where particle tracks and photon Cherenkov radiation are reconstructed for each event, respectively.
These are two interesting use-cases as they are time consuming processes, and particularly the RICH is only partially processed online due to its complex nature.

Vectorisation in the LHCb reconstruction Software has recently become a topic of attention, as several algorithms are being studied and redesigned to fully harness the potential of widely available SIMD architectures. Not only x86_64 architectures would benefit from such a design, but also increasingly popular manycore architectures require of a SIMD-like software design to become an option.

Furthermore, performance plays a key role in obtaining quality results in our software infrastructure, where the selection algorithms must meet tight time constraints. Automated performance and regression tests are a need in such an environment, to ensure a controlled optimization process in the development of the algorithms.

2. Vectorisation use-cases
Vectorisation can be performed in several ways. One of the most common ways to vectorise code is to identify parallelisable loops, and instruct the compiler to generate vectorised code, in an explicit manner. This process is known as horizontal vectorisation, and it is a well documented process [6].

Horizontal vectorisation has data and logical requirements. From the data point of view, the structure must be a Structure of Arrays (SoA). This kind of structure allows a vector register\(^1\) to be filled with data from subsequent steps of the loop in an efficient manner, without requiring additional overhead data preparation instructions. The data should also present a proper alignment to the vector width required. From the logical perspective, the steps in the loop should not incur into any data dependency with each other, to allow independent operation within them.

Horizontally vectorised code is a scalable solution to future architectures, however it requires a deep change in the data structures and in the design of current algorithms in the LHCb reconstruction. An example of this kind of vectorisation in LHCb can be found in G. Raven’s vectorisation of the PatForward algorithm [4].

In contrast with the above, vertical vectorisation consists in identifying and vectorising a process with many arithmetic operations and few loads and stores. The data structure requirement is then an Array of Structures (AoS), which we have found to be the common denominator in the LHCb reconstruction code. The logical constraints still exist in that no dependencies must exist within the arithmetic operations of the code, that is, an order must be followed.

Even though seemingly requiring less changes at a fundamental level on the underlying framework, vertically vectorised code is written with a certain vector width, making it not scalable for future architectures. For the results presented in section 3, we focus on vertically vectorised code analysis.

In opposition to manual vectorisation, most compilers offer the option to automatically generate vectorised code for a certain target architecture, also known as auto-vectorisation [5]. This solution has the advantage that it does not require any manpower to maintain an explicitly vectorised version of the code, and that it offers a performance gain at practically no cost. An example of a code that auto-vectorised with only a few changes in our codebase is presented in algorithm 1.

---

\(^1\) Vector register is used here as an architecture independent term. In x86_64, it would refer to either a XMM, YMM or ZMM register.
Algorithm 1: TrackFit solveParabola

```c
1 // const float det = (z1*z1)*z2 + z1*(z3*z3) + (z2*z2)*z3 - z2*(z3*z3) - z1*(z2*z2) - z3*(z1*z1);
2 // const float det1 = (x1)*z2 + z1*(x3) + (x2)*z3 - z2*(x3) - z1*(x2) - z3*(x1);
3 // const float det2 = (z1*z1)*x2 + x1*(z3*z3) + (z2*z2)*x3 - x2*(z3*z3) - x1*(z2*z2) - x3*(z1*z1);
4 // const float det3 = (z1*z1)*(z2-z3)+ ((z2*z2)*(z3-z1)) + ((z3*z3)*(z1-z2));
5 const float det = ((z1*z1)*(z2-z3))+ ((z2*z2)*(z3-z1)) + ((z3*z3)*(z1-z2));
6 const float det1 = (x1)*(z2-z3) + (x2)*(z3-z1) + (x3)*(z1-z2);
7 const float det2 = (z1*z1)*(x2-x3) + (z2*z2)*(x3-x1) + (z3*z3)*(x1-x2);
8 const float det3 = (z1*z1)*(z2*x3 -z3*x2) + (z2*z2)*(z3*x1 -z1*x3) + (z3*z3)*(z1*x2 -z2*x1);
```

In the C++ code presented above, the first four commented lines have been replaced by the four following lines, applying the associativity property on the products. gcc 4.8 reports a 1.18x speedup by doing this small exercise. However, such cases are scarce and heavily compiler-dependent: icc 14.0 reported a 1.04x, and clang 3.4 a 1.16x. Also, different reorderings have different effects on each compiler, and changes in the code can affect severely auto-vectorisation performance, making it a very fragile technology.

In our experience, the exercise of predicting what the compiler will do is not good practice. The focus should be in producing readable good code, and this should prevail over compiler-dependent predictions.

2.1. High level vectorisation libraries under study

In this paper, we have evaluated the viability of four well-known vectorisation libraries, focusing on the performance, scalability, readability and maintainability of each of them for the LHCb reconstruction software use case. We briefly introduce them below.

- **vc** stands for vector class, and is a library developed at the Frankfurt Institute of Advanced Studies, primarily by M. Kretz [6]. It is a library whose primary focus is horizontal vectorisation, and provides mechanisms to operate on width-independent vectors.
- **Vectorclass** is a high level abstraction class for vector operations developed by A. Fog [7]. It provides high level constructs of the functionality provided by the ISA, with additional functionality. As opposed to vc, Vectorclass is not focused on horizontal vectorisation.
- **gcc intrinsics** are an extension in the form of **builtins** to the C language, providing instructions that map onto assembly vector instructions [8]. They are processor-independent, however they are compiler dependent.
- **Intel intrinsics** are vendor-specific **builtins**. Both of the intrinsics presented represent the lowest-level instructions put down to the test. Intel intrinsics are the closest to the hardware, and latency and throughput on a per-instruction level can be found in its documentation [9].

3. Results

In table 1, a comparison is shown between implementations using different libraries of two algorithms, namely a VELO PrPixel and a RICH example. The intrinsics implementations are left out for the RICH case.

All shown versions of the algorithms have been validated to produce very similar results, that guarantee identical results when applied on the full reconstruction algorithms. From the
performance point of view, experiments are repeated millions of times and statistical invariance of the timings is guaranteed to the millisecond.

<table>
<thead>
<tr>
<th>library</th>
<th>PrPixel addHits</th>
<th>RICH EigenGeom</th>
<th>arch. portable</th>
<th>compiler portable</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>node_1(^2)</td>
<td>node_2(^2)</td>
<td>node_1</td>
<td>node_2</td>
</tr>
<tr>
<td>Intel intrinsics</td>
<td>1.94x</td>
<td>1.00x</td>
<td>1.00x</td>
<td>no</td>
</tr>
<tr>
<td>gcc intrinsics</td>
<td>1.45x</td>
<td>2.09x</td>
<td>yes</td>
<td>no (icc)</td>
</tr>
<tr>
<td>vc</td>
<td>1.66x</td>
<td>2.26x</td>
<td>1.35x</td>
<td>yes</td>
</tr>
<tr>
<td>Vectorclass</td>
<td>1.60x</td>
<td>2.23x</td>
<td>1.35x</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 1. Speedup obtained using various libraries for the PrPixel and RICH vertical vectorisation case studies. All results were obtained with the same executable, generated with the latest stable gcc compiler on SLC 6 at the time of writing, which is gcc 4.8.1, with optimization −O2 enabled.

Figure 1. Cross compiler times for PrPixel case study.

\(^2\) node_1 is an Intel Xeon X5650 @ 2.67 GHz

\(^3\) node_2 is an Intel Xeon E5-2630 v3 @ 2.40 GHz
The best performance is yielded by the Intel intrinsics implementation, with the caveat of having a very low readability, and hence a high maintenance cost. As the LHCb code is maintained by a varied community, we consider readability as an important factor. On top of that, both intrinsics implementations suffer from not being completely portable either across architectures or across compilers.

Even though vc and Vectorclass produce a very similar speedup, since vc is not geared towards vertical vectorisation, the Vectorclass implementation was more naturally readable. We notice an increase in speedup by analyzing the newer Haswell architecture of node 2 on all cases.

Figures 1 and 2 show cross compiler analysis for the PrPixel case study for three different nodes, compared against time and speedup to the sequential implementation, respectively.

Even though only one executable was produced per library, the speedup of the vectorised versions is noticeable for all versions of the code, with the difference being highest in both gcc examples. These results are in accordance with other Haswell studies [10], and make the case stronger towards manual vectorisation, as the tendency is that future architectures mean a higher gap between a sequential and a vectorised implementation.

A vectorised version of the PrPixel algorithm has been pushed to the production version. The speedup obtained in the full algorithm, due to the effect of Amdahl’s law [11] is a $1.15x$, which in the full LHCb online reconstruction translates into a 2% performance gain. In order to validate the timing effects of this change in production code, the LHCb Performance and Regression tool has been used, which we discuss in the following section.

![Figure 2. Cross compiler speedups for PrPixel case study.](image-url)
4. Performance and Regression tests

Optimising the event processing software in order to improve its speed is a challenge that has to be addressed in the coming years. As previous paragraphs are showing, vectorising the code, either explicitly or using vectorised libraries can definitely help to achieve that goal. It is however necessary to guarantee the accuracy of the results, especially in the case where the code has different implementations, depending on the underlying hardware.

To solve this issue, suitable tests are necessary to validate any changes to the software. Unit tests have been part of the LHCb software management tools since its inception, and they are run after every every build performed by the continuous integration system [12]. Units tests are however not enough, and the LHCb experiment computing team has put in place an application to run and record the results of integration tests [13]. This was first put in place to validate software migration for the LHCb simulation software, but is also a prerequisite to the refactoring of the software for optimization.

Validating the results of the optimisation of the software stack is however not an easy task for different reasons:

- Validating the performance improvements requires dedicated environments, where tests can be run without perturbations by other users. Very often special configurations will be required (e.g. disabling CPU throttling...) to make sure that the results are reproducible. Furthermore, some tests have to be done by loading a whole server in order to calculate its maximum throughput [14].
- When several implementations are available, e.g. depending on a specific instruction set available on the host, the tests have to be run on all versions of the software to make sure that the results are consistent.
- Computation results on different types of hardware are not necessarily identical to the last digit. The comparisons have to take the precision of the algorithms into account to validate the results.

The effort to put in place the tools and methods necessary for the refactoring of the LHCb codebase is ongoing, with the need to adapt to the technologies and methods that will be chosen for optimization.

5. Conclusions and outlook

A study of the different existing kinds of vectorisation has been shown. Auto-vectorisation can produce efficient results, but it is a fragile technology that is not guaranteed to adapt to our codebase evolution.

Several manual vectorisation methodologies have been shown, with concrete successful examples in the LHCb online reconstruction code. Identification of hotspots and manual tuning can lead to a reproducible speedup by virtue of taking advantage of vectorisation units in SIMD processors, like the x86_64 Intel Xeon family, which has a critical impact on a real-time system like the one under analysis.

Well-known vectorisation high level libraries has been put to study with a cross-compiler, cross-processor analysis. We show a comparison between them, and we conclude Vectorclass and vc to be libraries with a good balance between performance and readability. A tendency of an increase in relative speedup to the sequential version on a per processor level has been shown, which suggests vectorisation is receiving attention from CPU designers.

Performance and regression tests are a requirement in order to guarantee a sustained, controlled evolution of our codebase. We have discussed the requirements of such a system, and we are developing tools and methods for systematically analysing the effects of code revisions, such as the ones shown in this paper.
References

[5] Intel(R) Compiler auto-vectorisation guide
[8] GCC, the GNU Compiler Collection X86 Built-in Functions
[9] Intel(R) Intrinsics Guide
[12] Clemencic M and Couturier B 2015 LHCb Build and Deployment Infrastructure for run 2, CHEP