Development and Evaluation of Vectorised and Multi-Core Event Reconstruction Algorithms within the CMS Software Framework

T Hauth, V Innocente and D Piparo
CERN, Geneva, Switzerland
E-mail: thomas.hauth@cern.ch vincenzo.innocente@cern.ch danilo.piparo@cern.ch

Abstract. The processing of data acquired by the CMS detector at LHC is carried out with an object-oriented C++ software framework: CMSSW. With the increasing luminosity delivered by the LHC, the treatment of recorded data requires extraordinary large computing resources, also in terms of CPU usage. A possible solution to cope with this task is the exploitation of the features offered by the latest microprocessor architectures. Modern CPUs present several vector units, the capacity of which is growing steadily with the introduction of new processor generations. Moreover, an increasing number of cores per die is offered by the main vendors, even on consumer hardware. Most recent C++ compilers provide facilities to take advantage of such innovations, either by explicit statements in the programs sources or automatically adapting the generated machine instructions to the available hardware, without the need of modifying the existing code base. Programming techniques to implement reconstruction algorithms and optimised data structures are presented, that aim to scalable vectorization and parallelization of the calculations. One of their features is the usage of new language features of the C++11 standard. Portions of the CMSSW framework are illustrated which have been found to be especially profitable for the application of vectorization and multi-threading techniques. Specific utility components have been developed to help vectorization and parallelization. They can easily become part of a larger common library. To conclude, careful measurements are described, which show the execution speedups achieved via vectorised and multi-threaded code in the context of CMSSW.

1. Introduction

The Large Hadron Collider (LHC) [1] at CERN in Geneva is the most powerful particle collider ever built. One of the major experiments at the LHC is the Compact Muon Solenoid (CMS) [2], a general-purpose particle detector. Its research program is extremely ambitious and spans from the search of the Higgs boson to the quest for signals originating from processes beyond the established Standard Model of particle physics.

In order to carry out these searches, a tremendous amount of data has to be processed. Part of this procedure includes the so-called event reconstruction, which transforms the digital signals coming from the detector’s read-out into physics objects, like leptons and hadronic jets. These procedures foresee the running of a wide range of algorithms like cluster finding, pattern recognition, linear algebra and function minimisation. For this purpose, the CMS Collaboration has developed an object-oriented C++ software framework, called CMSSW [3].
With the continuous increase of the luminosity delivered by the LHC machine, the treatment of the recorded data becomes more and more computationally intensive. The demand for computing capabilities will increase further with the upgrade to the Super-LHC [4] which will provide an instantaneous luminosity of $10^{35}$ cm$^{-2}$s$^{-1}$.

A solution to face these challenges is the exploitation of features offered by the latest microprocessor architectures. Over the last years, a clear trend in the hardware industry became apparent. The CPU clock frequency did not increase significantly, but vector units with growing capacities and an increasing number of cores per die are offered by all major vendors today.

In order to benefit from these developments in the hardware industry, the already existing CMS reconstruction software must be adapted. This paper describes two strategies and evaluates their potential in the context of the CMS event reconstruction. Since a large fraction of the application runtime can be accounted to floating point arithmetics, the data-parallel approach offered by vector units is very well suited to speed up this calculations.

On the other hand, the various algorithms employed by the event reconstruction offer a huge potential for parallel execution. Up to now, it was sufficient to process the whole events in parallel using multiple instances of the reconstruction application. With the growing number of compute cores per physical machine, a more fine-grained parallelization below the event level becomes attractive to enable additional lightweight scaling capabilities.

2. Vectorization

Starting in the late nineties, the Intel® Corporation integrated Single Instruction Multiple Data (SIMD) [5] capabilities into their commodity CPU product line. While classical x64-64 processor instructions only operate on one set of input values (like $c = a + b$), the underlying concept of SIMD is to apply one mathematical operation to a whole range of input values (like $c_i = a_i + b_i$ in which $i = 1 \ldots n$). The benefit of this approach is that more values at the same time can be processed by the CPU. All processors in use by the CMS Collaboration today support the SSE2 instruction set. For this particular instruction set, two double precision floating point values can be processed with one instruction. The more recent AVX instruction set introduced in 2011 can apply operations on four double precision floating point values using one CPU instruction.

Although the SIMD instruction sets are a well-established feature of CPUs for quite some time, these extended functionalities could only be accessed by programmers willing to interleave their high-level C or C++ source code with quasi-assembler instructions called compiler intrinsics. The SSE2 and AVX technologies use different instructions. If all must be supported, the same algorithm has to be implemented several times.

Modern compilers, like the GNU Compiler Collection (GCC) [6], include the ability to transform regular C or C++ source code into vector operations. This process is called auto-vectorization and is transparent to the programmer.

The CMS Collaboration opted for the usage of open-source software technologies and therefore relies on the GCC compiler to build the complete CMSSW software stack. The presented results were achieved by using the GCC 4.6.1, if not otherwise stated.

2.1. Benefits of auto-vectorization

The benefits of using the auto-vectorization features of modern compilers instead of low-level intrinsics are manifold. The algorithms can still be expressed as high-level C/C++ code, which brings advantages for the development and maintainability. In addition, a regular programmer without SIMD-expertise is able to understand and extend the algorithms.

Furthermore, existing algorithms can be adapted to be auto-vectorized by the compiler. If intrinsic instructions are used, the algorithm must be reimplemented from scratch using the dedicated SIMD commands.
Arguably, the biggest advantage is that, once the algorithm can be auto-vectorized, the compiler can generate machine code for every available SIMD instruction set and thus support all hardware architectures. Also future SIMD specifications can be supported by upgrading to a new version of the compiler. Therefore, an existing auto-vectorizable C/C++ code fragment is guaranteed to benefit from advancements of the SIMD technology in upcoming microprocessors.

2.2. A library of fast and approximate auto-vectorizable mathematical functions

A significant portion of the time necessary to reconstruct CMS collision events is spent in the evaluation of transcendental mathematical functions. The ambitious goal of reducing this contribution can be achieved developing a collection of mathematical functions which satisfies these criteria:

- Is Open-Source
- The implementation is faster than the traditional libm [7] one
- The GCC compiler can auto-vectorize loops in which the function appears

Starting from the well known Cephes mathematical library [8], the source code was adapted and re-engineered to take advantage of the auto-vectorisation technology. A first implementation of a collection of inline double precision transcendental functions was provided that satisfies the aforementioned criteria. It will be referred to as vdt in the following. Table 1 and Figure 1 show the speed of these mathematical functions. All the timings reported are obtained with the 4.7 version of the GCC compiler suite and enabling the -Ofast [9] switch. The comparison is made with libm and the Intel svml library [10]. The latter is a commercial product and a reference on the market in terms of speed. The inverse square root function is the only one not present in the Cephes library. The implementation in this case is inspired from the “Quake3 Fast InvSqrt” (for the details of the implementation see [11]) and the number of iterations adopted for the measurements is 4.

| Function | libm | vdt | vdt SSE | libm SSE | vdt AVX | libm AVX |
|----------|------|-----|--------|---------|--------|---------|
| exp      | 19.7 | 8.6 | 5.4    | 4.3     | 4.0    | 4.2     |
| log      | 35.4 | 11.1| 6.9    | 6.1     | 5.5    | 5.7     |
| sin      | 21.7 | 11.2| 8.2    | 5.5     | 4.2    | 5.3     |
| cos      | 19.6 | 9.4 | 8.0    | 6.1     | 4.0    | 5.9     |
| asin     | 20.4 | 12.0| 9.5    | 7.6     | 9.2    | 7.4     |
| acos     | 21.1 | 18.1| 13.7   | 7.4     | 13.0   | 7.0     |
| tan      | 28.2 | 21.0| 10.8   | 6.0     | 9.6    | 5.7     |
| atan     | 7.4  | 10.9| 8.2    | 9.4     | 7.8    | 9.4     |
| inv. sqrt| 11.8 | 6.6 | 4.4    | 6.2     | 3.2    | 3.2     |

Table 1. Timing performance comparison of libm, vdt and svml mathematical libraries. The units are arbitrary and smaller figures are better on the scale considered.

It can be observed that the usage of vdt can be profitable both in the scalar and in the vectorized version.

The functions of the vdt and svml are less accurate as compared with the corresponding ones offered by the libm library. The full evaluation of the accuracy of these mathematical functions is beyond the scope of this document and the effect on the final output of the application in which they are used must be studied case by case. Nevertheless, a simple Monte Carlo study
Figure 1. Timing performance comparison of libm, vdt and svml mathematical libraries. The units are arbitrary and smaller figures are better on the scale considered. vdt is profitable even in the scalar case except for the atan function.

was performed in order to assess the accuracy of the vdt functions with respect to the libm ones.

For every function, a set of one million random input values in the domain was created. The binary representations of the double precision floating point results of the libm and vdt functions were compared. Thus, for every comparison, the position of the most significant different bit was stored. Table 2 shows the domains of the vdt functions and their estimated accuracy, quoting the superior and the average of the position of the most significant different bit. It must be noted that the superior is a pessimistic estimate, since it represents the largest deviation observed.

Table 2. The interval of definition and accuracy of vdt with respect to the corresponding libm implementation. The accuracy was estimated evaluating the functions over one million randomly distributed numbers in the domain. The superior and the average of the most significant different bit are reported. A difference of zero means identity of the two numbers, considering all the bits of their representations.

| Function | Interval of Definition | Superior | Average |
|----------|------------------------|----------|---------|
| exp      | [-708,708]             | 2        | 0.14    |
| log      | (0,1e307]              | 2        | 0.37    |
| sin      | [-2π,2π]               | 21       | 1.2     |
| cos      | [-2π,2π]               | 21       | 1.3     |
| asin     | [-1,1]                 | 2        | 0.32    |
| acos     | [-1,1]                 | 8        | 0.45    |
| tan      | [0,2π]                 | 21       | 2.1     |
| atan     | [-1e307,1e307]         | 0        | 0       |
| inv. sqrt.| (0,1e307]             | 2        | 0.48    |
2.3. Vectorized Vertex Clustering

In the high luminosity regime of the LHC machine, multiple proton-proton collisions occur in the same bunch crossing. Therefore, a high-quality reconstruction of the various interaction vertices is essential to achieve a precise understanding of the CMS measurements. The location of each of these vertices is computed via the reconstructed particle tracks as an input. The CMS event reconstruction relies on a deterministic annealing algorithm [12] to compute the location of the primary vertices in one event. Especially with an increasing number of pileup-interactions, this reconstruction step in CMSSW amounts for a considerable part of the overall application runtime.

After adapting two compute intensive loops of the deterministic annealing algorithm, the GCC compiler was able to auto-vectorize the performed mathematical operations. Furthermore, a large fraction of the algorithm’s runtime is spent in the evaluation of the exponential function. The previously introduced fast and auto-vectorizable exponential function of the vdt library was used to speedup this calculation.

The physics output of the improved algorithm has been fully validated using the extensive set of quality monitoring tools of CMS and a perfect agreement with the baseline is observed. Neither the vectorization process, nor the application of the approximate exponential function provided by the vdt library did influence the physics performance in any way.

The following timing measurements have been performed considering simulated $t\bar{t}$ events with 22 pile-up interactions on average. Table 3 show the runtime gain achieved by the two levels of optimization. The first stage is to enable GCC to auto-vectorize compute-intensive parts of the algorithm. Combining this with the vdt library results in an overall speedup of more than a factor of two.

### Table 3. Runtime of the vertex clustering

| Version                  | Runtime for 50 Events [s] | Ratio to Regular [1] |
|--------------------------|---------------------------|----------------------|
| Regular                  | 26.64                     | 1.0                  |
| Vectorized               | 19.96                     | 0.74                 |
| Vectorized + vdt         | 11.46                     | 0.43                 |

3. Multi-threaded track seeding

One of the initial steps of the CMS particle track reconstruction chain is the seeding step [13]. Therein, the parameters of the candidate particle tracks are initialized with compatible reconstructed hits from the inner detector layers. One of theses procedures is the triplet seeding step where an already compiled list of hit-pairs, which contain two compatible hits of the inner layers of the detector, is combined with hits from a third detector layer. Every hit of the third layer must be evaluated for geometrical compatibility, which results in a large set of mathematical calculations.

The triplet seeding step accounts for about 14% of the overall CMSSW reconstruction runtime in a high-pile up measurement taken in the year 2011.

3.1. Implementation

The Intel® Threading Building Blocks (TBB) [14] library was used to implement the parallel triplet seeding. This library is released under a GPL-based open source licence and offers a comprehensive set of components to build multi-threaded applications. Within the CMS
Framework a dedicated TBB-Service was created to hold a thread-pool which is conserved during the runtime of the application. Before the fully parallel portion of the algorithm can be, measurement data must be loaded into local data structures. This operations are also present in the regular, serial code and account to about 10% of the algorithms overall runtime.

The full reproducibility of the output of an algorithm is an important requirement in a High Energy Physics experiment. In general, the order of an output list is not preserved across runs if multiple threads append their results. Depending on the number of threads and their execution speed, the list order will be different each time the application is run. A way to prevent this behavior is to sort the output list after the parallel section of the algorithm is completed. Another way, this implementation relies on, is to partition the input list in blocks. A private results list gets assigned to each input block and all result elements of one block are added to this result list.

Each block is assigned to only one thread, ensuring that only one thread processes the elements contained in one block. In a final merge step, the private result lists are combined to the output list of triplet seeds. This method does not require a time consuming sort-step of the final result and guarantees the reproducibility of the output in all threading scenarios. A visual representation of this scheme can be seen in Figure 2.

![Figure 2](Image)

**Figure 2.** Schematic of the parallel triplet seeding. The input list is partitioned into blocks, which in turn are processed by dedicated threads. The final result list will be built once all blocks have been processed.

### 3.2. Physics Validation

A validation of the physics output was performed with a bit-by-bit comparison of all reconstructed track parameters between the serial and parallel versions using input data consisting of high pile-up events recorded in the year 2011. The parallel implementation of the seeding algorithm was run with a varying number of threads and the output was compared to that of the serial version. For all events and all threading configurations, the output of the serial and parallel version were identical.

### 3.3. Performance Results

After a successful validation of the physics output, a detailed runtime analysis of the implementation has been performed. The impact on the overall runtime and memory behaviour of the CMSISW application was measured. As this is the first implementation of a parallel algorithm in CMSSW, overheads and scaling behaviour were characterized as well.
The test machine was an Intel® Core™ i7 CPU X 980 running at 3.33GHz with 6 physical cores, which were provided as 12 Hyper-threaded cores to the application. The total RAM of the machine was 6 GB and the Red-Hat based Scientific Linux 5.8 [15] operating system was used. For the Hyper-threading measurement at the end of this section, a comparable machine but with 16 GB of main memory was used.

Figure 3. Speedup of the triplet seeding algorithm with respect to the number of available threads.

Figure 4. Fraction of time spent in merging the local result list to the final result list.

Figure 3 displays the scaling behaviour of the parallel triplet seeding implementation. The runtime with a varying number of threads is divided by the serial runtime of the algorithm. The line “perfect scaling” represents the optimal case wherein the runtime decreases proportionally to $\frac{1}{N}$, where $N$ is the number of threads. The Amdahl’s Law line displays the expected speedup when Amdahl’s principle [16] is applied using the known serial fraction (10%) and parallel fraction (90%) of the implementation to estimate the possible speedup with $N$ threads.

The parallel triplet seeding shows a good scaling up to five cores. Going beyond six threads, where the Hyper-threading is active, only brings a slight gain in speedup. Especially for four threads and more, the speedup expected by the Amdahl’s Law prediction is slightly exceeded. On the overall scale, the parallel implementation performs very well considering the serial part, which is limiting the scalability with a larger number of cores.

An important aspect of parallel programming is the merging of the work performed by the independent threads to a final result list. If this merging step is too time-consuming, the algorithms’ performance will not be able to benefit from parallel execution as the gains in runtime will be spoiled. Therefore, the necessary time to perform the final merging of the triplet seeds result list was measured and compared to the overall runtime of the algorithm in Figure 4. This merging only takes 0.1% to 0.5% of the overall runtime, depending on the number of threads, and its impact on the overall runtime is therefore negligible.

To quantify the impact on the overall CMSSW application, Figure 5 displays the average runtime of the full reconstruction per event and the memory consumption running with a varying amount of threads. On average, adding one additional thread to the thread pool increases the memory consumption of the application by about 1 MB. This is very moderate compared to the overall memory consumption of the application and can therefore be referred to as lightweight scaling. The gain in runtime for even two threads is more than one expects from only the parallel
seeding step. By distributing the work on more than one core, the application can benefit from the caches of two CPU cores at the same time, resulting in fewer cache misses. In any case, to further quantify this positive effect on the overall runtime, more detailed analyses of the cache usage in various threading scenarios have to be performed.

![CMS Full Reconstruction Runtime & Memory](chart.png)

**Figure 5.** Average runtime per event and memory consumption of the full CMS reconstruction chain for a varying number of threads.

### 3.4. Possible gains using the Hyper-threading technology

The Intel® Hyper-threading technology enables two threads to run on the same CPU. A fast, hardware-based context switching between the two threads allows to process the operations of one thread while the other one is waiting, for example, for a memory access. To take full advantage of all cores of a compute node in data centers or on user workstations, as many single-threaded applications must be run as there are cores available. As the current configuration of the CMS reconstruction has a memory footprint of about 1 GB of during processing, this is not possible on most machine setups due to main memory limitations.

By having an application which features a lightweight scaling, almost no additional memory has to be allocated to benefit from additional cores. To support this statement, six instances of a single-threaded CMSSW application were run to reconstruct 50 events each. After this, six instances of the two-threaded CMSSW application were run and accessed the Hyper-threading mode of the CPU.

Comparing the overall runtime of both setups shows that two-threaded version is 4% faster than the serial version. Although the aforementioned cache benefit does not apply here, because the Hyper-threaded cores have no additional cache, the application still scales very well at this level of “overcommitting” of the machine.

The setup of running 12 single-threaded CMSSW applications to also use the Hyperthreaded cores could not be tested as a memory boundary of the test machine (16 GB of RAM) was encountered.
4. Conclusion

Parallel processing techniques were presented and their usefulness in the context of the CMS experiment was evaluated. Both, vectorization and multi-threading techniques delivered performance improvements and preserve the physics output at the same time.

Auto-vectorization techniques were shown to be profitable in the context of vertex reconstruction and a fast, general-purpose mathematical library. Speedups quantifiable by a factor of two could be achieved in this areas.

The most promising application of vectorized processing in the CMS experiment is to optimize compute intensive parts of the reconstruction software.

Algorithm-parallelism can be applied to a larger body of code. The challenge is to ensure a thread-safe concurrent data access and the reproducibility of the results. The TBB library offers a rich set of constructs to ensure a thread-safe data access and was used successfully within with the CMS Framework.

It was shown for the first time that algorithms within the CMS reconstruction chain can be run in parallel and scale well with the number of available threads. Furthermore, the additionally consumed memory per thread is minimal and number the of threads can be increased at almost no cost.

Acknowledgments

This work was supported by the German Doctoral Student Programme at CERN.

References

[1] Large Hadron Collider website. http://lhc.web.cern.ch/lhc/.
[2] Compact Muon Solenoid website. http://cms.web.cern.ch/.
[3] David Lange. The CMS reconstruction software. Technical Report CMS-CR-2011-002, CERN, Geneva, Jan 2011.
[4] Katja Klein. The upgrade of the CMS tracker for Super-LHC. Technical Report CMS-CR-2009-356, CERN, Geneva, Dec 2009.
[5] Michael J. Flynn. Some computer organizations and their effectiveness. Computers, IEEE Transactions on, C-21(9):948 –960, sept. 1972.
[6] The GNU Compiler Collection, 2012. http://gcc.gnu.org/.
[7] Libm. http://sourceware.org/newlib/libm.html.
[8] Stephen Moshier. Cephes C and C++ language special functions math library. http://www.moshier.net/#Cephes.
[9] Optimization options of GCC 4.7, 2012. http://gcc.gnu.org/onlinedocs/gcc-4.7.0/gcc/Optimize-Options.html#Optimize-Options.
[10] Intel Short Vector Mathematical Library. http://software.intel.com/sites/products/documentation/hpc/composerxe/enu/cpp/win/index.htm.
[11] Quake3 fast inverse square root. http://en.wikipedia.org/wiki/Fast_inverse_square_root.
[12] Deterministic Annealing for Vertex Finding at CMS, 2005.
[13] Susanna Cucciarelli, Marcin Konecki, Danek Kotlinski, and Teddy Todorov. Track reconstruction, primary vertex finding and seed generation with the pixel detector. Technical Report CMS-NOTE-2006-026, CERN, Geneva, Jan 2006.
[14] Intel Threading Building Blocks, 2012. http://threadingbuildingblocks.org/.
[15] Scientific Linux, 2012. http://www.scientificlinux.org/.
[16] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. AFIPS Conference Proceedings (30), pages 483 –485, 1967.