Vectorization with Haswell and CilkPlus

August 2013

Author:
Fumero Alfonso, Juan José

Supervisor:
Nowak, Andrzej

CERN openlab Summer Student Report 2013

Project Specification

This project concerns the parallel computing and vectorization field for Physics Computing at CERN. The document summarises the results and experience from vectorization activities and an initial evaluation of the CilkPlus technology with two different benchmarks from CERN.

Abstract

With the release of the Intel Sandy Bridge processor, vectorization ceased to be a "nice to have" feature and became a necessity. This work is focused on optimization, running comparative measurements of available vectorization technologies currently under investigation by the CERN Concurrency Forum. In particular, the project involves an assessment of the limits of autovectorization in two compilers, an evaluation of CilkPlus as implemented in ICC/GCC and an evaluation of AVX/AVX2 benefits with respect to legacy SSE workloads.

Table of Contents

1 Introduction 3

2 Vectorization techniques 6

2.1 The Vc Library 8

2.2 CilkPlus 8

3 Evaluation setup and benchmarks 9

3.1 Experimental Platforms 10

3.2 Exploring vector instructions: Matrix Multiplication 11

3.3 CERN MLFit with 500'000 events 12

4 CilkPlus evaluation 14

4.1 Understanding Vectorization with CilkPlus 14

4.2 Geant Vector Prototype 18

4.2.1 First approach 20

4.2.2 Second approach 21

4.2.3 Third and fourth approach 22

4.2.4 Fifth approach 22

4.2.5 Preliminary results 22

4.2.6 Assembly code ... 23

4.2.7 Comparison of Perf (PMU based) statistics ... 24

4.2.8 An improved CilkPlus solution for the Geant Vector Prototype ... 25

5 Conclusions ... 27

6 Annex A: How to install the Geant Vector Prototype? ... 28

6.1 ROOT Installation ... 28

6.2 Geant Vector Prototype ... 28

6.3 How to run the Benchmark? ... 29

6.3.1 Example of execution ... 29

7 Annex B: Installation script for the Geant Vector Prototype ... 31

Acknowledgements ... 33

8 References ... 33

1 Introduction

Gordon Moore predicted in 1965 that the number of transistors on a chip would double each 18-24 months. This implies an increase of computational power. The Figure 1 shows the number of transistor evolution between 1971 and 2004. The dashed line shows the value of Moore prediction. The dots indicate the Intel processor model and the number of transistors. We can see that the last tendency is not very close to Moore prediction:

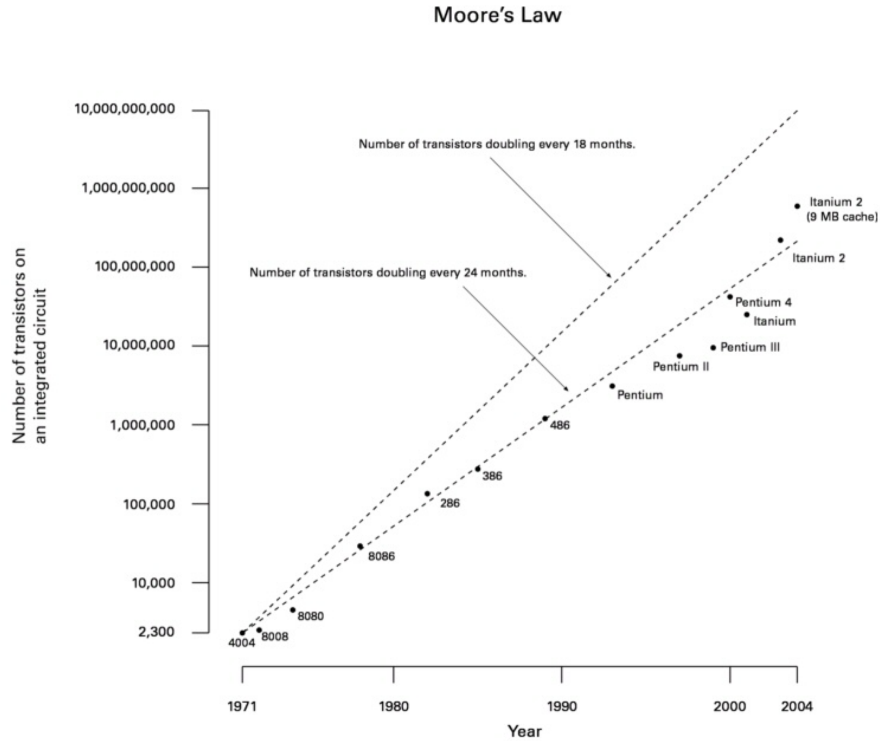

Figure 1: Increasing the number of transistors compared with the Moore's Law between 1971 and 2004

Nowadays, instead of using unique processors per chip (MCP), modern devices contain several processing units per socket and allow running several hardware threads simultaneously on the same socket. For each new generation of processor, manufacturers increase the numbers of cores and reduce the size of transistors, building faster and more integrated processors.

From the programming model view, this allows to exploit different levels of parallelism [2], from shared memory to distributed memory and heterogeneous systems, combining execution with any kind of accelerator such as GPUs or Intel MIC. It depends on the problem that we want to solve. At CERN the main computational problems are dedicated to Physics Computing and High Energy Physics. This kind of problem is data independent and it is usual to compute independent particles on independent cores. Exploiting vectorization could be an advantage to process more particles per instruction, in particular on the latest generations of CPUs, through the AVX and AVX2 instructions sets. This kind of parallel programming model is based on a Single Instruction Multiple Data model (SIMD). **This work is focused on this optimization area**.

With vectorization the programmer can operate with sets of data instead of scalar values in hardware. For instance, with AVX the programmer can multiply eight float elements in a vector in one processor instruction. Each core in the processor has special functional units to operate vector registers. Figure 2 shows the process inside a functional vector unit.

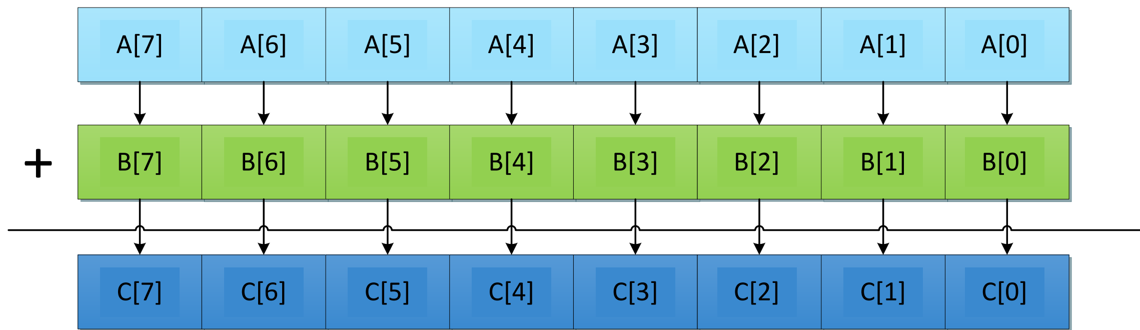

Figure 2: An Add operation in vectorization. In one hardware instruction it is possible add eight floats using AVX vector code on Sandy Bridge processors.

Advanced Vector Instructions (AVX) are extensions to the x86 instruction set architecture for Intel and AMD microprocessors. The first implementation of this instruction set was for Intel Sandy Bridge microarchitecture in 2011 and AMD Bulldozer several months after the Intel one. AVX is available in the last Intel microarchitectures: Sandy Bridge, Ivy Bridge and Haswell, but they do present some implementation and ISA differences, dependent on the architecture technology and the design of the processors. The theoretical FLOPS/Cycle, the number of floating point instructions per cycle, is 8 in the case of double precision on Sandy Bridge and Ivy Bridge and 16 on Haswell. In single precision the number of FLOPS/Cycle is 16 with Sandy Bridge and Ivy Bridge, and 32 in case of Haswell.

Haswell contains new instructions and new operations for FMA (Fused Multiply-Add) which double the core's peak FPU throughput [7]. Two FMA execution units were added to the microarchitecture. These instructions are able to execute operations such *as axb +c* with a single instruction. The L1 and L2 bandwidth have been doubled to ensure the execution units stay fed, and the integer and FPU register files have all been enlarged.

The code below is an example to show how the compiler generates vector instructions. The following loop can be vectorized with any vector instruction set.

```
for (int i = 0; i < n; i++) {
      c[i] = a[i] + b[i];
}
```

Compilers detect such cases and generate specific instructions sets such as SSE4, SSE, or AVX. The following snippet presents the assembly version of the previous C99/C++ code. Note the instructions start with "v", indicating that instructions are AVX and YMM registers are employed. Processors with the AVX instructions set contain 16 YMM registers (YMM0-YMM15). In the first line the vector elements are moved to the YMM0 register. Then an add operation is executed and the result is stored into the YMM1 register.

```
1   vmovups    .L_2il0floatpacket.8(%rip), %ymm0
2   vaddps     .L_2il0floatpacket.9(%rip), %ymm0, %ymm1
3   vmovups    %ymm1, 32(%rsp)
4   vmovups    %ymm1, 64(%rsp)
```

This approach can be combined with other models such as MPI or OpenMP. There are some modern programming imply vectorization. . For instance, when OpenCL is executed on the CPU vector instructions are normally used to execute the kernels.

Using vectorization the programmers can take advantage of the fastest registers in the core prepared for computation of a set of elements. Figure 3 shows the typical layout of modern multicore processors. A socket can contain several processing units. Each processing unit contains its own L1 Cache with Data and Instructions. The core is organized to execute microinstructions in a pipeline, an instruction runs on the functional units and the next instruction is decoding in the previous step of the pipeline [3] while the processor is looking for the next one in memory. The vector units are in the lowest level of this hierarchy and as a consequence are a fast facility that processes arrays and vectors.

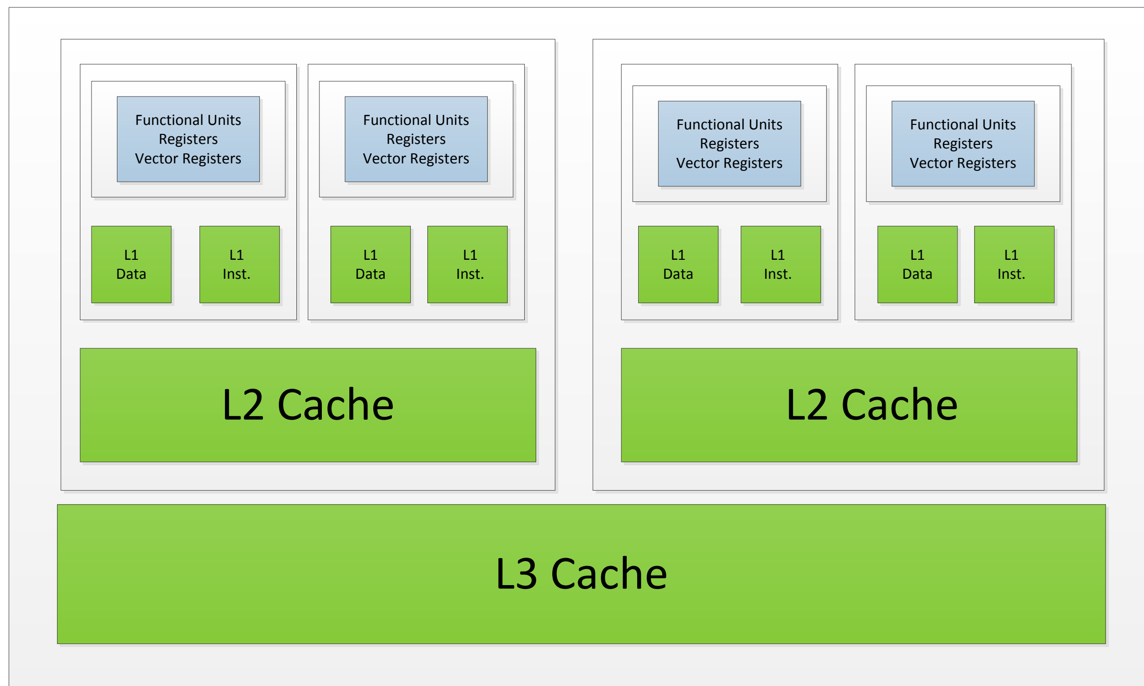

Figure 3: Cache Hierarchy in modern processors.

2 Vectorization techniques

There are several ways to develop vector code with GCC/Intel C/C++, from autovectorization, specific syntax to use vectorization, annotations in the code until *intrinsics* functions or assembler code. The compiler can help us to vectorize with specific options in compilation time, such as `–vec-report1` (in case of Intel Compiler) and detects the loops than can be vectorized and a briefly explanation about that. The compiler creates a dependency graph to detect is the loop is data independent and create vector code for SSE or AVX or other vector instruction set. There are several obstacles to creating vector code and easy loops are more likely to be converted to vector code rather than complex ones [1]. Some of the most relevant programming challenges are named as follows:

1. Countable loops. This means loop variable must be known at compilation time and does not depend on other conditions to break the loop.
2. No breaks inside the loops: The break statement terminates the vector code and parallelism.
3. No function calls. With Intel Compiler is it also possible call to MKL functions. Many of them are vector functions.
4. No data dependency between iterations. All iterations have to be completely independents to achieve vectorization. This case is similar to restrictions in OpenMP or Cuda/OpenCL.
5. Contiguous memory access: the vector optimization gets two, four or eight operands from memory (it depends on the vector instruction set such as SSE or AVX and the data type) and operates with this data. If the data is not contiguous, the processor has to look for the words in the different cache levels and vector code could not be efficient.

There are some general guidelines on how to write vectorizable code. As mentioned before, one should avoid data dependencies between loops and avoid read-after-write (RAW) dependencies. Also it is preferable help to compiler to take the decision on vectorization. To do that, it is better to use array annotations rather than using pointers. Sometimes, with Intel's compilers the code is not vectorizable although there is not any data dependency and pointers inside. This is because of compilers guard against possible data dependencies. It is up to the programmer to specify that that region is safe and vectorizable. In these cases, it is possible to assist the compiler through adding `#pragma vector always` and `#pragma ivdep` in the case of the Intel C compiler.

Another approach is to program with *instrinsics*. These are functions that the compiler replaces with the proper assembly instructions to use vector instructions sets. For instance, if to use AVX *intrinsics* one has to operate with a special data type and call a specific AVX function. The listing below shows a sketch of computation adding two vectors in AVX. In one AVX instruction it is possible to compute eight floats. Note that the loop is incremented by eight. Lines 3 and 4 load the eight floats elements into `a_i` and `b_i` variables. Line 7 computes the sum of these two vectors and line 10 stores the result in memory. In this case, the data type is float and we can package eight float numbers in an AVX register and operate on them (see lines 3-7).

```
1   for(int i = 0; i < N; i += 8) {
2       // Load 8 float to registers
3       __m256 a_i = _mm256_load_ps(&a[i]);
4       __m256 b_i = _mm256_load_ps(&b[i]);
5
6       // Compute out_i = a_i + b_i
7       __m256 out_i = _mm256_add_ps(a_i, b_i);
8
9       // Store the 8 floats into array
10      _mm256_store_ps(&out[i], out_i);
11  }
```

The compiler will replace the AVX called functions with inline assembly. This approach is very efficient but very low level. It is the responsibility of the programmer to write good code and to control any dependency in the loop. Otherwise, the result will be incorrect.

The lowest level approach is write assembly code inline and work with XMM and YMM registers directly and operate on them. This solution is the where the programmer can get the best performance but the ASM code could be hard to maintain and would be architecture dependent. There are other approaches to use vectorization, gaining portability and relatively high level of the programmability point of view such as specific libraries or compilers. This work is centered

on CilkPlus and autovectorization. The rest of the report is focused in these aspects. This work introduces a benchmark based on CilkPlus array notation and studies related issues, as evaluated at CERN.

2.1 The Vc Library

Vc is a free software library assisting with the vectorization of C++ code. This library is a high level API to use specific instruction sets such as SSE, SSE4 and AVX. The library is implemented using *intrinciscs* functions. The application programming interface of Vc is the most important feature allowing the efficient development of vectorized algorithms. The *intrinsics* functions are very low level of programming, but by working with the Vc Library, the programmer can abstract the lowest layer away and still use specific instructions to run his applications faster and take advantage of the SIMD programming model. Listing 1 shows a sketch of conversion from Cartesian 2D coordinates to polar coordinates with Vc Library. The full source code is available in the Vc webpage [9].

```
1  for (size_t i = 0; i < x_mem.vectorsCount(); i++) {
2    const float_v x = x_mem.vector(i);
3    const float_v y = y_mem.vector(i);
4    r_mem.vector(i) = Vc::sqrt(x * x + y* y);
5    float_v phi = Vc::atan2(y, x) * 57.295780181884765625f;
6    phi(phi < 0.f) += 360.f;
7    phi_mem.vector(i) = phi;
8  }
```

The objects x_mem and y_mem contain the Vc vectors. In the second and third line, x and y are copied to local variables. This can help to compiler with optimizations. The fourth line calls to a specific function of Vc (Vc::sqrt) and computes the r value to get the ratio in polar coordinates. In the same way, It is computed phi to get the appropriate angle. Finally the variable is saved into the vector. When the example is compiled the SSE implementation is used by default by default. It is possible to work with SSE2, SSE3, SSE4 and AVX.

2.2 CilkPlus

CilkPlus is an extension of the C99/C++ programming languages designed specifically for parallel computing through thread parallelism and vector parallelism. There are severalimplementations of this language. The most well-known implementation is that bundled with the Intel compiler. Recently, GCC has joined the CilkPlus effort and contains a branch of GCC from version 4.7 where the CilkPlus component is implemented. LLVM has implemented a CilkPlus but it is not completed and it only supports task parallelism and hyperobjects [4].

CilkPlus defines new tokens in the language to specify task parallelism (thread parallelism). There are three new tokens:

- `Cilk_spawn`: when this token is used the function that is called is meant to be executed in parallel. In this case, the runtime of CilkPlus creates a task dependency graph and if there is not any data dependency, CilkPlus creates a new thread to execute the function.
- `Cilk_sync.` previously with `cilk_spawn`.
- `Cilk_for`: this token is used to replace the "for" token and create a parallel loop. In this case it is the responsibility of the programmer to create vectorizable loops. Sometimes it is not possible because of data dependencies or complex loops.

In order to avoid deadlocks when `cilk_for` is used, Cilk Plus provides a set of reducers for the most common associative operations. All of these reductions functions are vectorizable and the user can implement new functions.

With the previous tokens it is possible express programs using thread parallelism. It is also possible create vectorizable code with CilkPlus. The language has some extensions very similar to OpenMP where the programmer can annotate his loops and help to the compiler to create the proper vector code.

```
1   #pragma simd
2   for (int i = 0; i < n; i++) {
3      v[i] = a[i] + b[i];
4   }
```

The compiler will optimize the above code to the following one:

```
1   for (int i = 0; i < n; i += 4) {
2      tmp_v[4];
3      tmp_v[0:4] = a[i:4] + b[i:4];
4      v[i:4] = tmp_v[0:4];
5   }
```

One of the main features of CilkPlus is the extension of C99 and C++ with special array notations. This notation expresses a set of high level vector parallel array operations. This is another way to create vector code and this helps the compiler to effectively vectorize the code. The array notation syntax is similar to the following:

```
array[lower_bound:length:stride]
```

The array is processed from the lower position until `lower + length`. The stride indicates the separation of the elements to be processed. This extension is particularly useful when one needs to start coding from scratch and obtain a fast implementation. If the code is clear and does not contain any data dependency, the compiler can create vector code for each array statement through array notation.

There are some additional annotations available to facilitate the compilation into vector code. One of them is `#pragma simd`. With this annotation the compiler enforces the creation of vector code, even if the generated assembly is not efficient. This pragma is useful when the programmer fully confident that the code is vectorizable.

Another approach that helps to compiler to vectorize the code is aligning the memory that we have allocated. The easiest way to write an annotation before the loop indicating to compiler that the next region aligns the memory - `#pragma vector aligned.`

3 Evaluation setup and benchmarks

In this report, two different Physics Computing benchmarks were evaluated. One of them is MLfit [8] that is presented in the next section and the other is a Geant Vector Prototype ported to CilkPlus. A preliminary evaluation with different instructions set across different Intel Microarchitectures is presented. The rest of work is focused on analysis two Benchmarks of physics of particles at CERN and a very simple case of study of Matrix Multiplication in order to

get a preliminary study of the different vector instructions set across different Intel microarchitectures.

3.1 Experimental Platforms

In order to evaluate different Intel microarchitectures with different vector instructions sets, three different platforms were used: two workstations and a server. Table 1 summarizes the different platforms. Moreover, MLFit was run with different vector instruction sets: SSE, SSE4.2 and AVX. In case of the Haswell microarchitecture, AVX2 and FMA were used. The benchmark was compiled with the Intel C++ Compiler 13.1 for each microarchitecture. Also some experiments were run with GCC 4.8.1 but we did not enough time to complete them. Each experiment took some hours if in case of GCC 4.8.1 in case of MLfit evaluation.

Server	Microarchitecture	Processor	Frequency (MHz)
olwork05	Sandy Bridge	Intel Xeon CPU E5-2690 (x2)	2900
opladev37	Ivy Bridge	Intel Xeon CPU E3-1265L v2	2500
opladev38	Haswell	Intel Xeon CPU E3-1285L V3	3100

Table 1: Michroarchitectures and features used in the experiments.

The Sandy Bridge server contains two sockets with eight physical cores each, and 16 threads when HyperThreading is enabled. A Sandy Bridge system, for example, can run up to 32 threads simultaneously. Both the Ivy Bridge and Haswell systems contain one socket and four physical cores each, with eight threads when HyperThreading is enabled. For all experiments, Turbo Boost was disabled and results are scaled to the frequency of the Sandy Bridge processor. For each machine the `KMP_AFFINITY` variable was used to fill up all physical cores and then all virtual cores. The first core (core 0) has been filled as the last one, because it is also used by the operating system.

Table 2 shows cache sizes for each Intel microarchitecture. As mentioned before, the Sandy Bridge machine is a server type system which contains two sockets and 32 threads in total.

Machine	L1	L2	L3
Sandy Bridge	32 K	256K	20M
Ivy Bridge	32 K	256K	8M
Haswell	32 K	256K	8M

Table 2: Memory cache sizes for each microarchitecture used in the experiments.

Table 3 indicates the compiler options that were used with the Intel C++ Compiler. Notice that on Haswell it is possible to use the `core_avx2` option to generate instructions from the AVX2 vector instruction set.

Option	Description
-xsse	The compiler enables SSE3, SSE2 and SSE1 vector code
-xsse4.2	ICC may generate instructions from SSE to SSE4.1 and SSE4.2
-xavx	ICC generates instructions for AVX (256 bits) if the processor supports them.
-xcore_avx2	ICC generas AVX2 vector code, only enabled on the Haswell microarchitecture.
-no-fma	ICC enables FMA by default when AVX2 is used. This option is needed to disable FMA and compare AVX2 vector code with AVX

Table 3: Intel C/C++ Compiler options used on MLFit.

3.2 Exploring vector instructions: Matrix Multiplication

Matrix Multiplication (MxM) is a basic kernel frequently used to showcase peak performance. The program was compiled with the Intel C Compiler 13.1. The code was selected from Intel Benchmark for Matrix Multiplication. This benchmark is implemented with OpenMP.

The figure 4 shows the speedup for AVX for different microarchitectures. In this case, AVX2 on Haswell is around 2 times faster compared to AVX in Sandy Bridge. Ivy Bridge presents almost

the same speedup for AVX and AVX2 without FMA on Haswell. The big advantage is when AVX2 and FMA are used at the same time. All results are scaled by frequency and the matrix size is A(600,1200) * B(1200,2400). The benchmark multiplies the matrices 100 times and takes the total time.

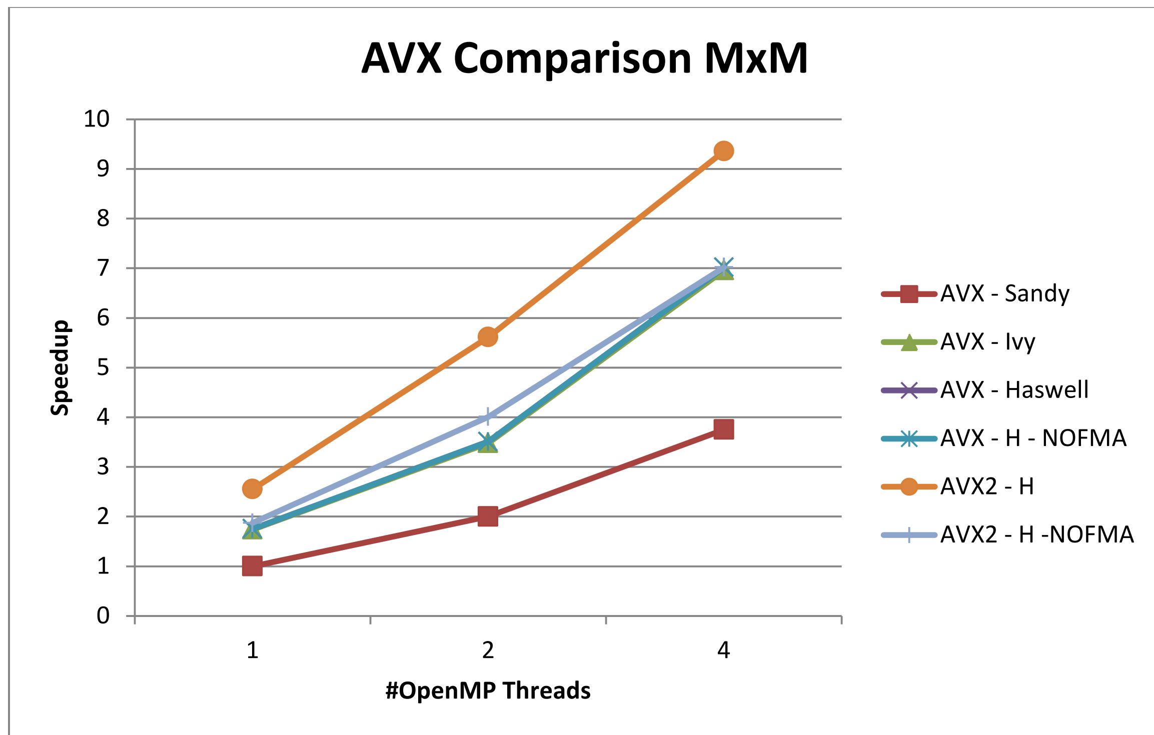

Figure 4: Comparison with AVX across different Intel microarchitectures for MxM problem.

3.3 CERN MLFit with 500’000 events

MLfit is a simplified version of the Roofit package (a component of the ROOT framework used in likehood based data analysis) at CERN. The MLfit benchmark is used at openlab as a representative of data analysis applications used in the High Energy Physics community. The Maximum Likelihood (ML) procedure is a popular statistical technique used to estimate parameters of a statistical model on a given numbers of events. Some previous publications provide specific details about the MLfit method computed in this benchmark [8], [10].

The benchmark is written in OpenMP and MPI [1] and can be executed on CPUs and on the Intel Xeon Phi accelerator. In this work a study of this benchmark is presented across different Intel Microarchitectures and different vector instruction sets: SSE, SSE4.2, AVX and AVX2.

The benchmark was run with 500,000 events for reference with previous measurements. Figures 5 and 6 show the speedup with different vector instruction sets across different Intel microarchitectures. The speedup for each variant is shown on Figure 5, where it is compared to one thread with SSE on Sandy Bridge. AVX in Sandy Bridge is a bit better than SSE and SSE4.2 for this numbers of events. Ivy Bridge scales to a factor of 1.5 with SSE and SSE4.2. In the case of Haswell, SSE and SSE4.2 scale by a factor of 1.7 and AVX/AVX2 by a factor of more than

2.5. On this benchmark, there are not many differences between AVX and AVX2, even when using the FMA instruction set. Figure 7 shows the relative speedup for each instruction set classified by microarchitecture. The graph shows that Haswell provides better potential for speedups. Major are noted when AVX and AVX2 are used. Overall, the speedup in the case of Haswell is 2.5 times higher than Sandy Bridge with one OpenMP thread.

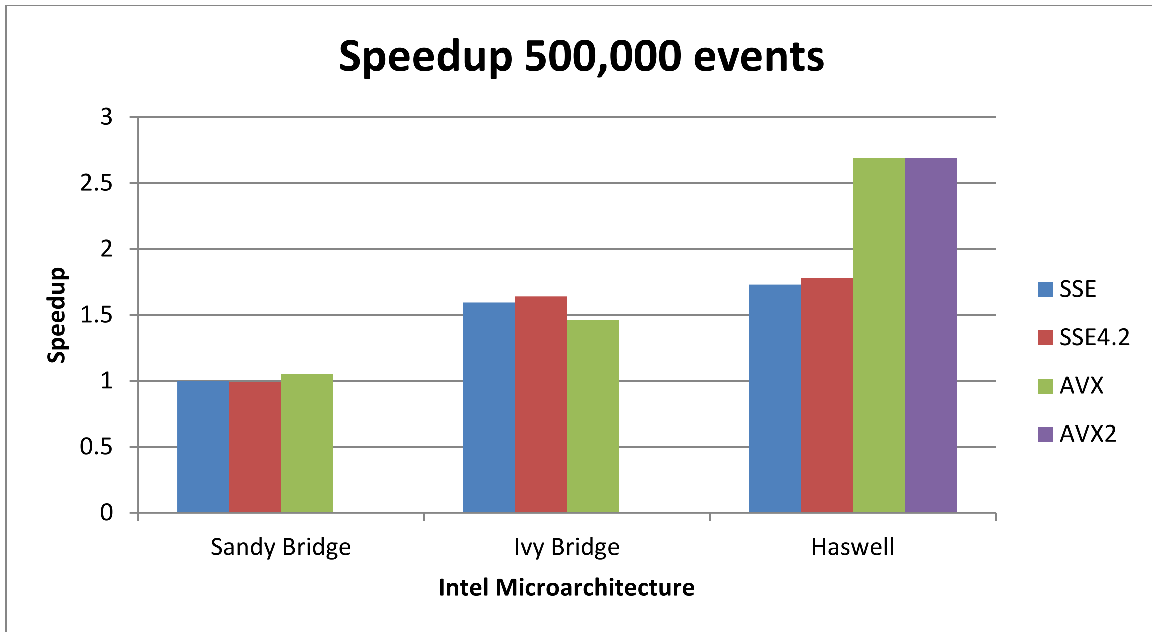

Figure 5: Speedup with 500,000 events for MLfit with different Intel microarchitectures.

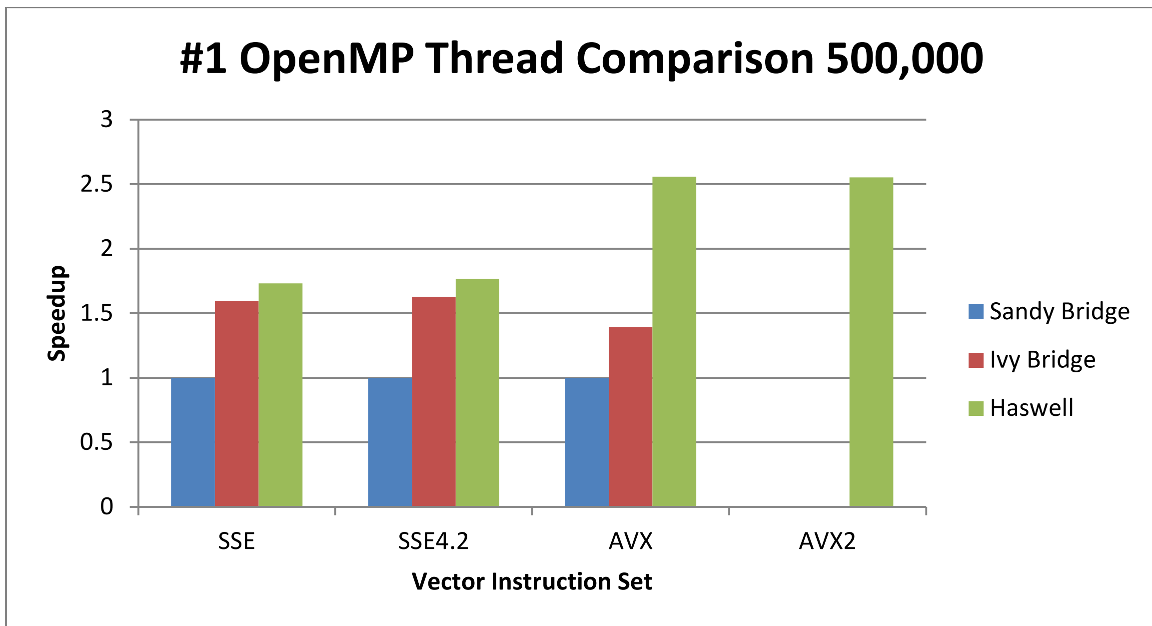

Figure 6: Relative speedup with different instruction sets over Intel microarchitectures.

In case of AVX comparison (Figure 7), AVX2 does not seem to present any advantage over AVX. Also, for all results, the Ivy Bridge microarchitecture scaled almost the same eight threads working than Haswell microarchitecture.

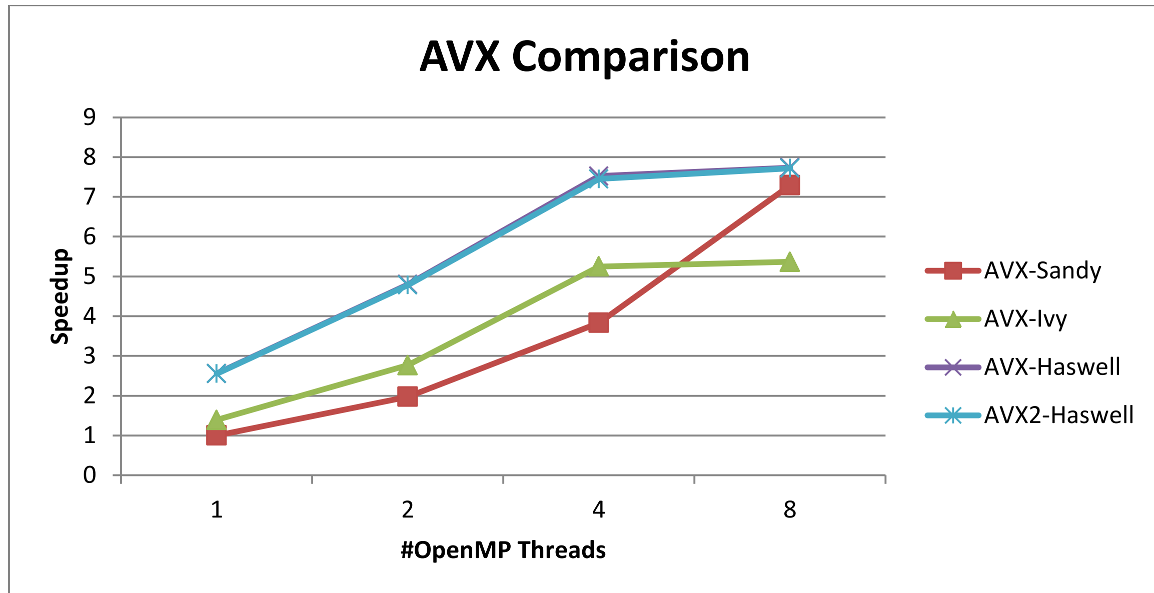

Figure 7: AVX comparison from 1 to 8 threads with different Intel microarchitectures.

4 CilkPlus evaluation

The ideal case to create vector code is through autovectorization. But the compiler has to be smart to create the proper vector code and sometimes it is not possible or the code is not as good as we expected. In these cases are needed new tools or libraries to create take advantage of the vector code. CilkPlus compiler with data parallelism has been evaluated. The classical MxM problem has been developed with different approach in CilkPlus and the Benchmark Geant Vector Prototype. All tests were executed on Haswell mircroarchitecture and were compiled with Intel 14.0. GCC 4.8.1 CilkPlus branch was not possible because of the compiler cannot parse some CilkPlus statements. The errors were reported to GCC CilkPlus team.

4.1 Understanding Vectorization with CilkPlus

In the section 2.2 it was mentioned that it is possible to create vector code through `#pragma simd` and CilkPlus array notations. But they present some differences and the code that the compiler generates is different for each strategy (CilkPlus array notation or `#pragma simd`). In this section, a study of different implementations of a short benchmark in CilkPlus is presented, in order to create and understand the vector code, the programmability of such syntax and how the compiler deals with source code of this type.

Below the classic algorithm for Matrix Multiplication is presented (MxM). The following code shows the sequential version of the MxM problem.

```
// result = a*b
void mxm_seq(double * restrict result, double *a, double *b, int m) {
    int i, j, k;
    for (i = 0; i < m; i++) {
        for (j = 0; j < m; j++) {
            for (k = 0; k < m; k++) {
                result[i*m+j] += a[i*m+k] * b[k*m+j];
            }
        }
    }
}
```

The `#pragma simd` version is identical to the sequential one. In this case, the compiler is given more information about how to create vector code. The snippet below shows the pragma-SIMD implementation.

```
void mxm_cilk_simd(double  * restrict result, double * a, double * b, int
n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            #pragma simd reduction(+:result[i*n+j])
            for (int k = 0; k < n; k++) {
                result[i*n+j] += a[i*n+k] * b[k*n+j];
            }
        }
    }
}
```

`#pragma simd` instructs the compiler to ignore every implication from the language standard, such as the data dependency, memory access, etc [5]. Besides, the efficiency heuristic is turned off by the compiler when this pragma is enabled. The loop has to be an ideal case to vectorize and it is the responsibility of the programmer to provide favourable conditions. Pragma SIMD should be written with a reduction operation. In the case of the Intel 13.1 compiler (ICC) the code is semantically correct and the result is correct as well. However, in case of ICC 14.0 release version the result is wrong. It is necessary to add the reduction operation (`reduction(+: result[i*n*j]`).

As a first implementation with CilkPlus Array Notation, the sequential version was translated to CilkPlus array notation. The code is shown below:

```
void mxm_array_notation_fail(double  * restrict result, double * a,
double * b, int n) {
    // Process four elements by four elements.
    // AVX can multiply 4 doubles in one instruction.
    int stride = 4;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k+= stride) {
                result[i*n+j:stride] += a[i*n+k:stride] *
b[k*n+j:stride:n];
            }
        }
    }
}
```

This snippet is almost the same as the sequential version, but four elements are multiplied by four elements with Array Notation. Again, the code is incorrect. The compiler unrolls the loop by 4 using array notations and the array “result” is invariant in the inner loop. Even so, the compiler does not return any error in compilation time or warnings. Using the array notation at the left hand side (result[i*n+j:stride]) does not work because this would imply that j columns of b would be processed at the same time. Because of 1. this is does not work. Two strides would be needed at the same time, one for k and one for j which is also not possible (kind of nested array notations).

This has been corrected above by using a reduction for the same (invariant) element result[i*n+j].

```
void mxm_array_notation(double  * restrict result, double * a, double *
b, int n) {
    // Process four elements by four elements.
    // AVX can multiply 4 doubles in one instruction.
    int stride = 4;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k+= stride) {
                result[i*n+j] += __sec_reduce_add(a[i*n+k:stride] *
b[k*n+j:stride:n]);
            }
        }
    }
}
```

In this case the result and the code are correct but the performance is not very good because vectorization is only used in the reduction operation. One of the possible optimizations is to interchange the loops j and k such as is shown in the code below.

```
void mxm_array_notation_interchange(double  * restrict result, double *
a, double * b, int n) {
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < n; k++) {
            for (int j = 0; j < n; j+= n) {
                result[i*n+j:n] += a[i*n+k] * b[k*n+j:n];
            }
        }
    }
}
```

The snippet is quite simple: the code is clearer than the previous one and it is also possible to express it with array notation, giving the compiler more information about the data distribution and possible vector code. The difference with #pragma simd is that in this case, array notation respects the language standard (aliasing, data dependencies, etc). The compiler is more conservative when array notation is used.

Figure 8 shows runtimes for each implementation, varying the matrix size, compiled for AVX vector instruction set. The MxM program with CilkPlus was executed on a Haswell machine with one thread (HyperThreading enabled and TurboBoost disabled) and compared with one CilkPlus thread using different approach to vectorization. The program was compiled with the beta release of the Intel C Compiler version 14.0. The best case is that where the OpenMP (thread

parallelism) version is used, with only 1 thread for comparison. The best CilkPlus variant is the one where two loops (j, k) are interchanged, and the performance is superior to the pragma simd version. On Figure 9 one can see in more detail the speedup for each CilkPlus version and OpenMP compiled for AVX2. This case is similar to the behaviour found in AVX. The speedup with pragma simd is less than one in both cases (AVX and AVX2). This means that the code is slower than the sequential code.

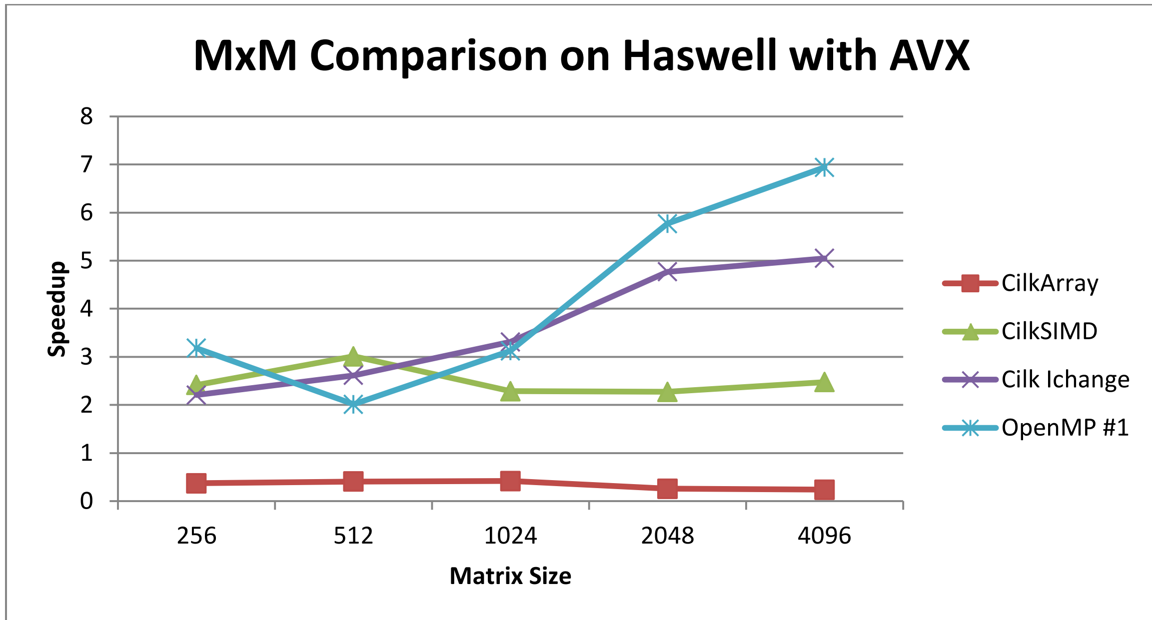

Figure 8: MxM comparison with CilkPlus AVX and OpenMP varying the data size.

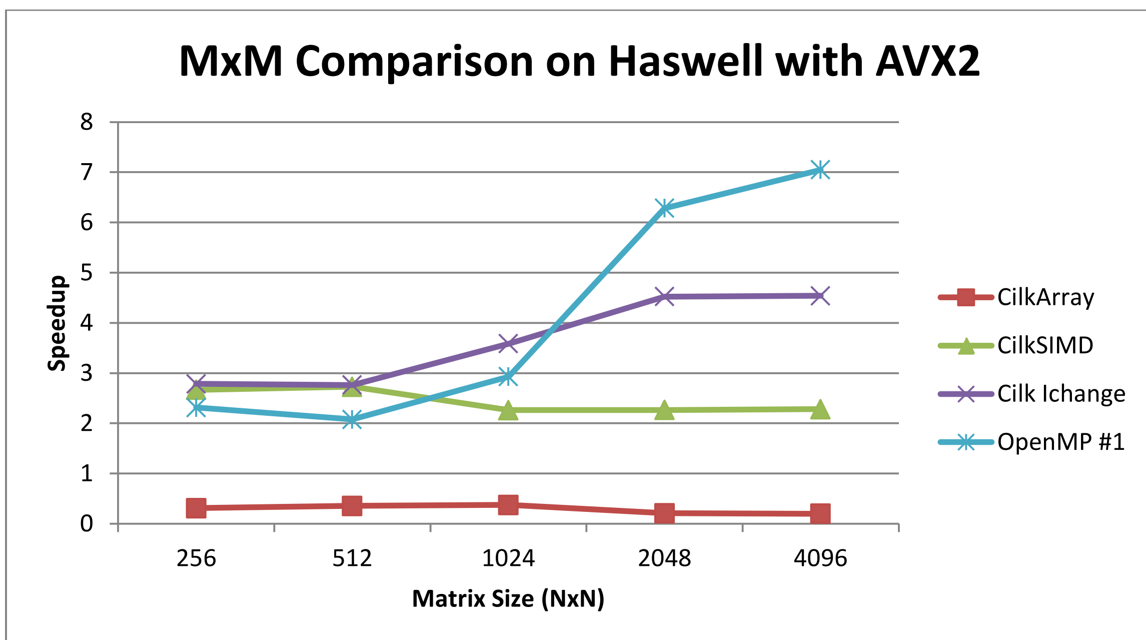

Figure 9: MxM comparison with CilkPlus and AVX varying the data size.

This analysis shows the differences between each implementation using AVX and AVX2 on the Haswell microarchitecture. Figure 10 shows these differences in speedup comparing AVX and AVX2 to the sequential version and a matrix size of 1024x1024. The bars titled CilkSIMD show the speedup for the pragma simd implementation. If the code is compiled for AVX2 the speedup is over 2.5 better than the sequential version. The central bars correspond to the CilkPlus array notation with j-k loops interchanged. The speedup in case of AVX is almost 4.5 and is about 20% faster than AVX. The last two bars show the speedup with OpenMP and 1 thread.

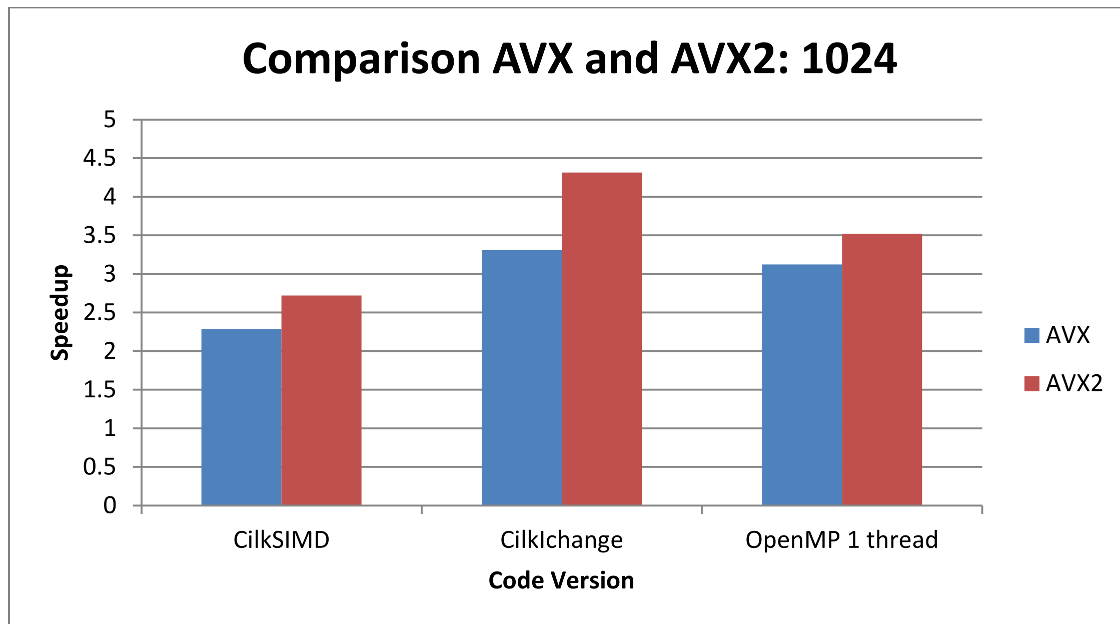

Figure 10: Speedup compared with the sequential version in Haswell. The results correspond to a matrix size of 1024x1024. The OpenMP version was run with 1 thread.

4.2 Geant Vector Prototype

Detector simulation is one of the most CPU intensive tasks in modern High Energy Physics. Geant Vector Prototype represents a benchmark for these kinds of simulations. It is used a part of ROOT Framework and compares the last processors optimization and advantage with the traditional model of ROOT [6].

At CERN openlab, a new part of this benchmark has been implemented in CilkPlus to study the differences with respect to other strategies for vectorization. The source code simulates a set of particles around a shape and computes the distance and direction for each particle. Figure 11 shows a representation of the calculation that is performed in the TGeoBBox simulation. In these tests, the original implementation was evaluated, where the compiler enables autovectorization and it was contrasted with new implementations using the CilkPlus array notation. Although CilkPlus contains specific constructs for a thread model, such as cilk_for or cilk_spawn, where the runtime could launch new tasks in parallel, our work is focused on the array notation. Each particle is completely independent of the other particles, as in other simulations of this type. Reading the array notation the compiler is expected to know more about how the data is

organized and it should therefore generate vector instructions. The overall advantage of vectorization here is that is possible to process more particles per instruction.

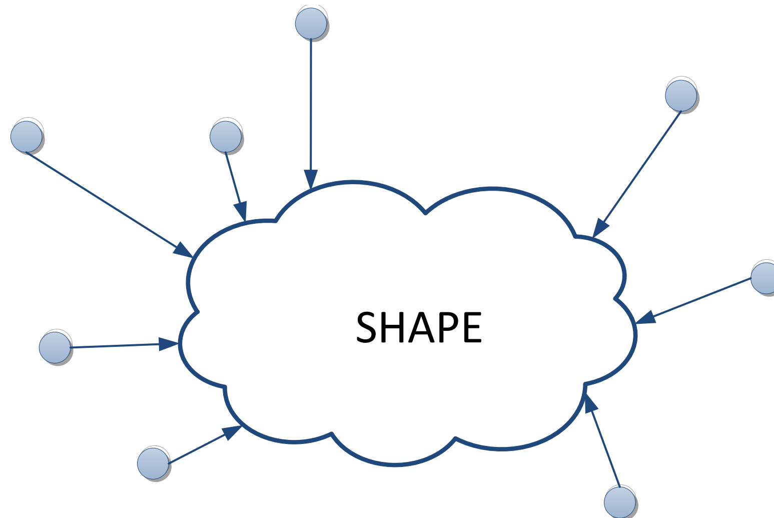

Figure 11: Overview diagram of the Geant Vector Prototype simulation for TGeoBBox code. The distance for all particles around the shape is computed.

The original version of this benchmark contains an outer loop to process all particles. For each particle, its distance x,y,z from the shape is computed. The Listing below shows pseudo code of the computation implemented in the function DistanceFromOutside:

```
for (k = 0; k < np; k++) {
        Compute_x();
        Compute_y();
        Compute_z();
        Distance[k] = factor*combiation(x,y, z);
}
```

This loop is called 14 times because there are 14 sets of particles, varying in size from 1, 2, 4… to 8192. For each number of particles, there are 500 repetitions.

The next listing shows a sketch of the original code. It consists of a main loop to process each particle (x, y, z) around the shape. First, the code computes the position of the particle relative to the origin. Second, the final distance of each particle having the values x, y, z is updated. Overall, five different approaches have been implemented, each aiming for better performance than the original version.

```
for(unsigned int k=0;k<np;++k) // @EXPECTVEC
    {
      Bool_t in;
      Double_t saf[3];
      Double_t newpt[3];

      Double_t factor=1.;
      Double_t infactor;
```

```
        newpt[0] = p[0][k] - origin[0];
        saf[0] = TMath::Abs(newpt[0])-par[0];
        factor = (saf[0]>=stepmax[k]) ? TGeoShape::Big() : 1.; // this
might be done at the end
        in = (saf[0]<0);

        newpt[1] = p[1][k] - origin[1];
        saf[1] = TMath::Abs(newpt[1])-par[1];
        factor *= (saf[1]>=stepmax[k]) ? TGeoShape::Big() : 1.; // this
might be done at the end
        in = in & (saf[1]<0);

        newpt[2] = p[2][k] - origin[2];
        saf[2] = TMath::Abs(newpt[2])-par[2];
        factor *= (saf[2]>=stepmax[k]) ? TGeoShape::Big() : 1.; // this
might be done at the end
        in = in & (saf[2]<0);

        infactor = (double) !in;

        // i=0
        Int_t hit0=0;
        if ( saf[0] > 0 & newpt[0]*d[0][k] < 0 ) // if out and right
direction
         {
           snxt[0] = saf[0]/TMath::Abs(d[0][k]); // distance to y-z face
           double coord1=newpt[1]+snxt[0]*d[1][k]; // calculate new y and
z coordinate
           double coord2=newpt[2]+snxt[0]*d[2][k];
           hit0 = (TMath::Abs(coord1)>par[1] | TMath::Abs(coord2)>par[2])?
0 : 1; // 0 means miss, 1 means hit
         }
```

4.2.1 First approach

If there is not any data dependent between iterations, the vectors in the code could be expressed with array notation. The vectors contain the information points (x,y,z) for each particle. This means that there are very small vectors

```
double point_aux[3]    = { p[0][k], p[1][k], p[2][k] };
newpt[0:3] = point_aux[0:3] - origin[0:3];
saf[:] = abs(newpt[:])-par[:];
if ( saf[:] >= stepmax[k] )
    #pragma vector aligned
    factor[:] = geobig;
else
#pragma vector aligned
factor[:] = 1;
factor_scalar = __sec_reduce_mul(factor[:]);
if (saf[:] < 0)
           in = false;
infactor = (double) !in;
```

the listing above shows the first part, where each component of each particle is computed in the original code. The major difference is that the code is much smaller than the original version. The code is much clearer and this also can improve the productivity. However, the Intel compiler cannot vectorize some of these statements with array notation. For instance, in the second line the compiler returns a message that the loop is complex. The same happens on the third line, in which

a call to an external function (abs) to is needed to compute the absolute value of each element before doing the final operation. In any case, the if statement was vectorized as well as the reduction operation.

```
//#pragma vector aligned
if ( saf[:] > 0) {
    if  ( newpt[:]*distance_aux[:] < 0 ) {
        snxt[:] = saf[:]/abs(distance_aux[:]);
        coord1_vector[:] = newpt[aux_newpt[:]]
+snxt[:]*d[aux_newpt[:]][k]; // calculate new y and z coordinate
        coord2_vector[:] = newpt[aux2_newpt[:]]+snxt[:]*d[aux_d[:]][k];
        hit_vector[:] = (abs(coord1_vector[:]) > par_auxa[:] |
abs(coord2_vector[:]) > par_auxb[:])? 0 : 1; // 0 means miss, 1 means hit
        }
        }
        distance[k] = ( hit_vector )?
factor_scalar*infactor*(__sec_reduce_add(hit_vector[:] * snxt[:])) :
infactor*geobig;
     }
```

This listing shows the last part of the original loop. In this case, we need to compute the final distance for the particle located in (x,y, z) but each component has to be computed having the rest of the components in mind. Note that an additional vector was used as a second index to the data vectors. This kind of expression can reduce the code and the number of conditional statements for each particle's components The problem (again) is that the compiler is not vectorizing this code.

4.2.2 Second approach

In the first approach, very small vectors were processed and the data size did not exceed three elements.. It is possible to remove the main loop of the function, but temporary variables and arrays are needed to process all particles in one CilkPlus statement. Actually all x, all y and all z values of all sets of particles are computed:

```
    Double_t *newpt_x = (Double_t *) _mm_malloc(sizeof(Double_t)*np,
ALIGN_AVX);
    Double_t *newpt_y = (Double_t *) _mm_malloc(sizeof(Double_t)*np,
ALIGN_AVX);
    Double_t *newpt_z = (Double_t *) _mm_malloc(sizeof(Double_t)*np,
ALIGN_AVX);
```

The following Listing shows how x values are computed with CilkPlus.

```
// All x points for all particles
 newpt_x[0:np] = p[0][0:np] - origin[0];
 saf_vector_x[0:np] = abs(newpt_x[0:np]) - par[0];
 if (saf_vector_x[0:np] >= stepmax[0:np])
     factor[0:np] = big;
 else
     factor[0:np] = 1;
 if (saf_vector_x[0:np] < 0)
     in_vector[0:np] = true;
 else
     in_vector[0:np] = false;
```

The advantage of using this approach is that the source code is much clearer than the rest of the implementation and also the number of lines of code is reduced. The main problem is that the compiler does not create the vector code for the first and second statements.

4.2.3 Third and fourth approach

It has been mentioned that we need additional memory to process all values (set of particles). In order to reduce this, two additional versions were implemented: one of them allocates arrays of medium sizes (called tiles) and processes them. The second one is allocating the memory in advance and processes the tiled data. The problem with these two last approaches is that, in the best case, the time is the double of that in the version mentioned before (the second approach where all particles are processed through CilkPlus array notation).

4.2.4 Fifth approach

The last idea was to process the elements in the loop in order to compute four particles each iteration. Additional vectors are needed like in the second version, but much smaller (12 elements to process four particles each iteration). The following listing shows a portion of the code discussed. Note that this approach is on a lower level compared to the rest of implementations. But, in this case, the Intel compiler detects all CilkPlus array notation as vectorizable statements, even the arrays that in the first implementation were considered to be "complex structures".

```
#pragma ivdep
#pragma vector always
#pragma vector aligned
for(unsigned int k = 0; k < np; k += 4) {
    #pragma vector aligned
    factor[:] = 1;
    double point_aux[12];
    unsigned int j = k;

    #pragma simd
    for (unsigned int i = 0; i < (LENVEC*3); i+=3) {
        point_aux[i] = p[0][j];
        point_aux[i+1] = p[1][j];
        point_aux[i+2] = p[2][j];
        j++;
    }

    #pragma ivdep
    #pragma vector always
    #pragma vector aligned
    newpt[0:3] = point_aux[0:3] - origin[0:3];
    newpt[4:3] = point_aux[4:3] - origin[0:3];
    newpt[7:3] = point_aux[9:3] - origin[0:3];
    newpt[10:3] = point_aux[10:3] - origin[0:3];
    saf[:] = abs(newpt[:])-par[:];
    if ( saf[:] >= stepmax[k] )
        factor[:] = geobig;
```

4.2.5 Preliminary results

Table 4 shows the preliminary results for each CilkPlus version of the Geant Vector Prototype Benchmark. The total time and the computational time per particle have been measured. The Benchmark is run to process from 1 particle to 8192 particles. The times and analysis showed in the Table 4 is for 8192 particles. The total time indicates how much takes to run it and the time

per particle is the total time divided into the number of particles. The last columns show wheter the compiler was able to vector all CilkPlus array notation in the code. The Root version indicates the original version which corresponds to the code in the Root framework.

Version	Total time	Time (s) per particle	Same result?	Vector CODE	Vector ALL
Root	0,276	2,730E-04	YES	-	-
Auto-optimized version	0,123	1,180E-04	YES	AVX	NO
Last-cilkplus	0,269	2,680E-04	YES	AVX	NO
First-cilkplus	0,274	2,730E-04	YES	AVX	NO
Second-cilkplus	0,377	3,540E-04	YES	AVX	NO
Third-cilkplus	0,407	3,730E-04	NO	AVX	NO
Fourth-cilkplus	0,650	8,040E-04	YES	AVX	YES

Table 4: Time comparison between different versions of the Geant Vector Prototype benchmark, including CilkPlus implementations.

Note that the best version is the original version, as optimized by the compiler. In this case there is no CilkPlus array notation and the compiler takes the its own decisions on how to optimize the code. The second best version is when small vectors are processed small vector and when they are processed vectors each 4 particles. In case of the third version, where the additional memory is allocated outside the main loop, the result is not correct.

4.2.6 Assembly code

Table 5 shows the comparison between the ASM (.S) file generated for each version of the benchmark. The second column indicates the total ASM lines of the function that is being vectorized. The second and third columns indicate the number of instructions that used XMM registers and YMM registers. There are some instructions, such as vinsertf128 and vextractf128, where both registers are used. This is shown in the fourth column. The vinsertf128 instruction replaces either the lower half or the upper half of a 256-bit YMM register with the value of a 128-bit source operand. The other half of the destination is unchanged. Correspondingly, vextracttf128 extracts either the lower half or the upper half of a 256-bit YMM register and copies the value to a 128-bit destination operand.

On the right side of the table the number of jump instructions is indicated, as well as the total time to execute the function for each particle. Notice that the best version, as mentioned before, is the automatically optimized one. The statistics for this version are shown in the first row. Other rows indicate the values for each CilkPlus version. The last version correspondswhen CilkPlus detects all array notation are vectorized. In this case the number of total lines is lower than the original version, but contains less XMM and instructions. The first version is the simple translation to CilkPlus in order to use small vectors each iteration. In this case the total of assembly lines is very low and also contains almost 50% of instructions with XMM registers, but in this case is not used any YMM operation.

Code	#Lines	XMM Instructions	YMM instructions	V*tf128	Jump	Time
Auto-optimized version	729	434	135	33	28	0,150
Last-cilkplus	507	342	154	34	10	0,281
First-cilkplus	289	133	0	0	11	0,266
Second-cilkplus	707	275	136	28	12	0,431
Third-cilkplus	707	275	136	28	12	0,462
Fourth-cilkplus	1010	406	217	102	44	0,745

Tabla 5: ASM Comparison for each CilkPlus version and the auto optimized version.

4.2.7 Comparison of Perf (PMU based) statistics

To better understand the performance results, each version was measured with linux-perf. Tables 6 and 7 summarize the results. All results presented are the total statistics.

Version	Cycles	Insns Per Cycle	Branches	Time
Auto-optimized version	2.678.846.562	1,17	481.479.014	0,000118
Last-cilkplus	3.340.918.582	1,12	545.952.203	0,000268
First-cilkplus	3.356.677.160	1,24	640.323.963	0,000273
Second-cilkplus	3.870.739.549	1,07	634.312.969	0,000354
Third-cilkplus	3.950.685.971	1,08	651.628.305	0,000373
Fourth-cilkplus	4.715.796.328	1,19	645.794.406	0,000804

Table 6: Perf comparison for each CilkPlus version and the auto optimized version

The original version executes 1.17 instructions per cycle and the total cache misses is the lowest value compared with the rest of implementations with 7,5% of total references. The last version and the first version show very similar results. The major difference is between the numbers of instructions per cycle being better in the first implementation with 1.24. The three last implementations are not doing very well. Following the results in table 7, the last version, where all CilkPlus statements with array notation are vectorized by the Intel compiler and the first version (small vectors) present a similar ratio of cache misses, about 27%.

Version	Cache Misses	Cache references	% Cache Misses	Time
Auto-optimized version	327.094	4.509.255	7,254	0,000118
Last-cilkplus	894.158	6.879.908	12,997	0,000268
First-cilkplus	887.427	6.699.531	13,246	0,000273
Second-cilkplus	3.169.916	13.064.301	24,264	0,000354
Third-cilkplus	3.396.769	13.423.264	25,305	0,000373
Fourth-cilkplus	3.057.942	14.076.222	21,724	0,000804

Table 7: Perf Comparison. Cache misses and reference for each version.

4.2.8 An improved CilkPlus solution for the Geant Vector Prototype

With receiving feedback from Intel Support, a new CilkPlus version of the benchmark was proposed. This one is similar to the last version where the Intel compiler detects all array notation statements as vectorizable portions of code. The time per particle of this version is 59.23 us.. This is about 50% faster than the autovectorizable version of the Geant Vector Prototype.

The main difference with the rest of approaches is that in this case a set of 4 particles is computed in parallel. This fixed number is the number of doubles that can be processed in one AVX instruction. The listing below shows a part of this solution. Note that explicit memory alignment is needed before processing the elements with array notation. All memory for extra arrays is aligned at the beginning of the loop. Also, the majority of arrays are not allocated. Instead of allocations, a set of pointers is used. This allows to save memory and gives more information to the compiler about the data distribution through constant variables.

```
for (int i = 0; i < np-tailsize; i+= 4)
  {
     const double (*vx)[4]       = (const double (*)[4]) &(point.x[i]);
     const double (*vy)[4]       = (const double (*)[4]) &(point.y[i]);
     const double (*vz)[4]       = (const double (*)[4]) &(point.z[i]);
     const double (*vdirx)[4]    = (const double (*)[4]) &(dir.x[i]);
     const double (*vdiry)[4]    = (const double (*)[4]) &(dir.y[i]);
     const double (*vdirz)[4]    = (const double (*)[4]) &(dir.z[i]);
     const double (*vstepmax)[4] = (const double (*)[4]) &(stepmax[i]);
     double (*vdistance)[4]      = (double (*)[4]) &(distance[i]);

     __assume_aligned(vx,32);
     __assume_aligned(vy,32);
     __assume_aligned(vz,32);
     __assume_aligned(vdirx,32);
     __assume_aligned(vdiry,32);
     __assume_aligned(vdirz,32);
     __assume_aligned(vstepmax,32);
     __assume_aligned(vdistance,32);
     __assume_aligned(origin,32);

     double newptx[4] __attribute__((aligned(32)));
     double newpty[4] __attribute__((aligned(32)));
     double newptz[4] __attribute__((aligned(32)));

     double safx[4] __attribute__((aligned(32)));
     double safy[4] __attribute__((aligned(32)));
     double safz[4] __attribute__((aligned(32)));

     double in[4] __attribute__((aligned(32)));

     double snxtx[4] __attribute__((aligned(32)));
     double snxty[4] __attribute__((aligned(32)));
     double snxtz[4] __attribute__((aligned(32)));

     double hit0[4] __attribute__((aligned(32)));
     double hit1[4] __attribute__((aligned(32)));
     double hit2[4] __attribute__((aligned(32)));

     double mask1[4] __attribute__((aligned(32)));
     double mask2[4] __attribute__((aligned(32)));

     newptx[:] = vx[0][:] - origin[0];
     newpty[:] = vy[0][:] - origin[1];
     newptz[:] = vz[0][:] - origin[2];
```

```
      #pragma ivdep
      safx[:] = std::abs(newptx[:]) - dx;
      safy[:] = std::abs(newpty[:]) - dy;
      safz[:] = std::abs(newptz[:]) - dz;

      vdistance[0][:] = big;

      mask1[:] = (safx[:] >= vstepmax[0][:] || safy[:] >= vstepmax[0][:]
|| safz[:] >= vstepmax[0][:]) ? 1.0 : 0.0;
      int faraway = __sec_reduce_any_nonzero(mask1[:]);
      if (faraway) return;

      in[:] = ((safx[:] < 0.0) && (safy[:] < 0.0) && (safz[:] < 0.0)) ?
0.0 : 1.0;

      snxtx[:] = safx[:]/std::abs(vdirx[0][:]+tiny);
      mask1[:] = std::abs(newpty[:]+snxtx[:]*vdiry[0][:]) - dy;
      mask2[:] = std::abs(newptz[:]+snxtx[:]*vdirz[0][:]) - dz;
      hit0[:] = (safx[:] > 0.0 && newptx[:]*vdirx[0][:] < 0.0 &&
(mask1[:] <= 0.0  && mask2[:] <= 0.0)) ? 1.0 : 0.0;

      snxty[:] = safy[:]/std::abs(vdiry[0][:]+tiny);
      mask1[:] = std::abs(newptx[:]+snxty[:]*vdirx[0][:]) - dx;
      hit1[:] = (safy[:] > 0.0 && newpty[:]*vdiry[0][:] < 0.0 &&
(mask1[:] <= 0.0  && mask2[:] <= 0.0)) ? 1.0 : 0.0;

      snxtz[:] = safz[:]/std::abs(vdirz[0][:]+tiny);
      mask2[:] = std::abs(newpty[:]+snxtz[:]*vdiry[0][:]) - dy;
      hit2[:] = (safz[:] > 0.0 && newptz[:]*vdirz[0][:] < 0.0 &&
(mask1[:] <= 0.0  && mask2[:] <= 0.0)) ? 1.0 : 0.0;

      if ( hit0[:] > 0.0 || hit1[:] > 0.0 || hit2[:] > 0.0 )
      vdistance[0][:] = hit0[:]*snxtx[:] + hit1[:]*snxty[:] +
hit2[:]*snxtz[:];
      #pragma vector nontemporal
      vdistance[0][:] *= in[:];
   }

   // do the tail part for the moment, we just call the old static
version
   for(unsigned int i = 0; i < tailsize; ++i)
     {
       double p[3]={point.x[np-tailsize+i], point.y[np-tailsize+i],
point.z[np-tailsize+i]};
       double d[3]={dir.x[np-tailsize+i], dir.y[np-tailsize+i], dir.z[np-
tailsize+i]};
       distance[np-tailsize+i]=TGeoBBox_v::DistFromOutsideS(p, d, dx, dy,
dz, origin, stepmax[np-tailsize+i] );
     }
}
```

Some pragmas are still needed, like `#pragma ivdep` and `#pragma nontemporal`. If `#pragma ivdep` annotation is not added, the compiler returns a message indicating that in this piece of code there is data dependence and it does not generate vector code. In that case the time per particle is 0.000112 seconds, about 5% faster than the original but 45% slower if the annotation is not written.

```
#pragma ivdep
safx[:] = std::abs(newptx[:]) - dx;
safy[:] = std::abs(newpty[:]) - dy;
safz[:] = std::abs(newptz[:]) - dz;
```

`#pragma vector nontemporal` directs the compiler to use non-temporal stores on systems based on all supported architectures. If `#pragma vector nontemporal` is not added there is not difference in the runtime on Haswell and using the Intel C Compiler 14.0.

5 Conclusions

Since the Sandy Bridge microarchitecture, it is possible to work with wider vector registers. The Intel microarchitectures reviewed present a clear evolution of the vector instructions sets with AVX and AVX2. This evolution is also well visible in Intel Xeon Phi coprocessors. AVX2 is on the way to an extension to 512 bits of length, where it will be possible to compute eight numbers in double precision per processor instruction.

An evaluation of two different tools used in High Energy Physics at CERN was presented in this work. Also, several challenges of autovectorization and of how to write vector code with CilkPlus have been studied with different benchmarks.

From the point of view of this evaluation and production systems, CilkPlus array notation is a novel offering. In case of the GCC variant, several bugs have been reported to the GCC CilkPlus team. Because of its novelty, the CilkPlus implementation in the LLVM compiler is not equipped with support for data parallelism yet. This includes: `#pragma simd`, elemental functions and array notation.

With the high level of parallel programming and vectorization in these extensions, the compiler must still be relatively smart in order to analyse code and to generate proper vector code without semantic changes. The high level of abstraction at which many of the operations surveyed are used is sometimes penalized with poor performance, and it is necessary to provide the compiler with more information, often excessive from the point of view of code readability. These are low-level and architecture specific details, like memory alignment or specific pragmas to be sure that the data set is completely independent. Only then can good performance be achieved.

From the programmability point of view, CilkPlus is very easy to learn and brings benefits. The syntax for array notation is simple yet powerful. Some classic examples, like MxM with CilkPlus, were studied with different approaches. In summary, the more information the compiler has about

how the data is distributed, the better. At the same time, the more lower levels details are available, the better. Aligning memory properly or loop tiling are architecture-dependent (for instance when a stripped is expressed related with the number of instructions that can be processed in parallel with AVX) and the high level programmer has to know the details – despite using a promising high-level syntax. CilkPlus is a potential language for physicists, mathematicians and scientists who might not be computer architecture experts. That philosophy is opposite to the very interesting aim of CilkPlus.

CilkPlus could be a very interesting language in future generations of the compilers. Apart from abstaining from having to declare low levels details, there should be an acceptable way to express the data parallelism and to benefit from it. Working with other approaches like *intrinsics*, modern programs become completely dependent of the architecture, but with CilkPlus array notation a set of statements could end up directly on vector units.

Finally, some questions and bugs were reported to Intel and to GCC in the hope that following versions of Intel and GCC CilkPlus are fixed and improved.

6 Annex A: How to install the Geant Vector Prototype?

Geant Vector Prototype depends on ROOT framework. To install see the following steps.

6.1 ROOT Installation

```
$ cd ~
$ git clone http://root.cern.ch/git/root.git
$ git checkout -b v5-34-09 v5-34-08
$ cd root
$ ./configure
$ make –j 8
```

When the installation is finished is needed set all ROOT variables:

```
$ source bin/thisroot.sh
```

6.2 Geant Vector Prototype

First of all is needed the source code.

```
$ cd  ~
$ git clone https://git.cern.ch/pub/geant
$ git checkout cilkdev
```

An auto configuration script is given. It is necessary to change the first line of the script. It depends of the installation path of Intel Compiler, ROOT and Geant Vector Prototype.

```
 8 #########################################################
 9 # Parameters of installation
10 GEANT_DIRECTORY="`pwd`/Geant Vector Prototype"
11 ROOT_DIRECTORY="`pwd`/build/"
12 ICC_PATH="/opt/envhpc/intel/composerxe"
13 MODULE="/usr/local/Modules/3.2.9/bin/modulecmd bash"
```

```
14 OUTPUT="testing01"
15 ########################################################
```

The module variable is not used with Intel Compiler compilation. It is necessary to change or review the lines 10-14. The next Listing is showed an example of execution in order compile with ICC.

```
$ ./configure_geant.sh icc
Compiling with Intel Compiler

Building in directory: testing01

-- The C compiler identification is Intel
-- The CXX compiler identification is Intel
-- Check for working C compiler: /opt/envhpc/intel/composer_xe_2013.2.146/bin/intel64/icc
-- Check for working C compiler: /opt/envhpc/intel/composer_xe_2013.2.146/bin/intel64/icc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /opt/envhpc/intel/composer_xe_2013.2.146/bin/intel64/icpc
-- Check for working CXX compiler: /opt/envhpc/intel/composer_xe_2013.2.146/bin/intel64/icpc -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Found ROOT 5.34/08 in /home/juanjo/cern/experiments/build
-- Vc library not found; try to set a VCROOT environment variable to the base installation path or add -DVCROOT= to the cmake command
-- Found TBB library in /opt/envhpc/intel/composer_xe_2013.2.146/tbb/lib/intel64/libtbb.so
-- Configuring done
-- Generating done
CMake Warning:
…
```

6.3 How to run the Benchmark?

One configured and compiled ROOT framework and Geant Vector Prototype is needed a set some environment variables. They are needed ICC, ROOT and some LD_LIBRARY_PATH to install_dir/lib. For instance:

```
$ cd testing01/bin
$ source ~/cern/experiments/root_gcc/bin/thisroot.sh
$ source /opt/envhpc/intel/bin/compilervars.sh intel64
$ export LD_LIBRARY_PATH=../lib/:$LD_LIBRARY_PATH
$ ./
```

6.3.1 Example of execution

```
$ ./BenchTGeoBBox_v
# TIMER OVERHEAD IS :2.29895e-08
 prepared data with 8192 points inside
 prepared data with 8192 points inside
 prepared data with 8192 points inside
 prepared data with 8192 points inside
 prepared data with 8192 points inside
 prepared data with 8192 points inside
 prepared data with 397 points inside
 prepared data with 0 points inside
```

```
 have 2676 points hitting
 have 2731 points hitting
Loops from 0 to 500 and 0 to 14
Elapsed sequential time: 0.315074 (s)
CilkPlus elapsed time per particle 0.000312 (s)

   VECTORSIZE  TIMECONTAINS       R.SPEEDUP     TIMESAFETY      R.SPEEDUP
TIMEDISTMIN     R.SPEEDUP   TIMEDISTOUT     R.SPEEDUP
            1   3.05725e-08              1   4.54025e-08              1
4.91745e-08            1   7.00585e-08            1
            2   2.80385e-08        2.18075   3.20345e-08         2.8346
8.79325e-08       1.11846   9.79425e-08       1.4306
            4   3.82225e-08        3.19942   3.91425e-08        4.63971
1.409e-07       1.39601   1.72776e-07       1.62195
            8   5.14805e-08        4.75093   6.14305e-08         5.9127
2.52336e-07       1.55901   3.40388e-07       1.64655
           16   9.35025e-08        5.23152   1.03128e-07        7.04403
4.7124e-07       1.66962   6.30192e-07       1.77872
           32    1.8278e-07        5.35243     1.874e-07        7.75281
9.29266e-07       1.69336   1.21564e-06       1.8442
           64   3.54198e-07        5.52413   3.89298e-07        7.46409
1.76296e-06       1.78516    2.4022e-06       1.86652
          128   6.99166e-07        5.59706   6.93584e-07        8.37896
3.48517e-06       1.80603   4.81845e-06       1.86107
          256   1.40463e-06        5.57198   1.37913e-06        8.42779
6.91519e-06       1.82044   9.66296e-06       1.85605
          512   2.78114e-06         5.6283   2.73169e-06        8.50977
1.37696e-05       1.82848   1.92421e-05       1.86414
         1024   5.58977e-06        5.60063   5.49868e-06        8.45514
2.73758e-05       1.83939   3.87529e-05       1.85121
         2048   1.10941e-05        5.64375   1.08922e-05        8.53679
5.46321e-05       1.84341   7.73969e-05       1.85382
         4096     2.234e-05         5.6054   2.16725e-05        8.58084
0.000109007       1.84776   0.000155169       1.84933
         8192   4.46212e-05         5.6128     4.312e-05        8.62562
0.00021842       1.84432   0.000314792       1.82317
Main loop:  0 - 500 & 0 - 14
CilkPlus elapsed time: 0.297455 (s)
CilkPlus elapsed time per particle 0.000294 (s)

   VECTORSIZE  TIMECONTAINS       R.SPEEDUP     TIMESAFETY      R.SPEEDUP
TIMEDISTMIN     R.SPEEDUP   TIMEDISTOUT     R.SPEEDUP
            1   3.78496e-07              1   1.07716e-07              1
7.51925e-08            1   2.25414e-07            1
            2   2.58886e-07        2.92403   5.26405e-08        4.09253
8.54105e-08       1.76073   1.86712e-07       2.41456
            4   2.95824e-07        5.11785   4.66705e-08        9.23208
1.27182e-07       2.36487   1.84472e-07       4.88776
            8   3.43968e-07        8.80305   4.78605e-08        18.0051
2.19228e-07        2.7439   3.35336e-07       5.37763
           16   4.63612e-07        13.0625   6.97925e-08        24.6941
4.38452e-07       2.74392   6.32702e-07       5.70036
           32   6.63832e-07        18.2454   9.75405e-08        35.3384
7.90096e-07        3.0454   1.21671e-06       5.92852
           64   1.16048e-06        20.8739   1.28728e-07        53.5535
1.56424e-06       3.07646   2.41882e-06       5.96427
          128   1.54983e-06          31.26   2.11776e-07         65.105
3.08445e-06       3.12037   4.62606e-06       6.23707
          256   2.90885e-06        33.3105    3.9925e-07         69.068
6.12921e-06       3.14058   9.17012e-06       6.29284
          512    5.6094e-06        34.5474   6.99888e-07        78.7995
1.22631e-05       3.13937   1.82339e-05       6.32955
```

```
            1024    1.14482e-05         33.8552    1.33441e-06         82.6597
2.45009e-05        3.14262     3.633e-05        6.35355
            2048    2.33089e-05          33.256    2.66165e-06         82.8822
4.8965e-05        3.14498    7.27336e-05        6.34712
            4096    4.57848e-05          33.861    5.45109e-06         80.9391
9.78391e-05        3.14791    0.000145978        6.32489
            8192    9.18368e-05         33.7625    1.07172e-05         82.3359
0.000195598         3.1492    0.000296118        6.23601
```

7 Annex B: Installation script for the Geant Vector Prototype

```
#!/bin/bash

# #####################################################
# Author: Juan José Fumero | July 2013
# Email : juan.jose.fumero.alfonso@cern.ch
# #####################################################

#######################################################
# Parameters of installation
GEANT_DIRECTORY="`pwd`/geant5"
ROOT_DIRECTORY="`pwd`/build/"
ICC_PATH="/opt/envhpc/intel/composerxe"
MODULE="/usr/local/Modules/3.2.9/bin/modulecmd bash"
OUTPUT="testing01 "
#######################################################1

module () {
    eval `$MODULE $*`
}

function icc() {
    echo "Compiling with Intel Compiler"
    echo -e "\nBuilding in directory: $OUTPUT \n"
    source $ROOT_DIRECTORY/bin/thisroot.sh
    source $ICC_PATH/bin/compilervars.sh intel64
    export CC=icc
    export CXX=icpc
    dir_build=$OUTPUT
    mkdir $dir_build
    cd $dir_build
    export GEANT_OUTPUT_DIR=$PWD
    #module add vc
    export LDFLAGS="-lrt"
    if [[ $1 == "original" ]]
    then
        echo -e "\n CMAKE with original version  \n"
        cmake -DUSE_ICC=ON -DVECGEOBENCHMARKS=ON -DUSEAVX=ON -
DCMAKE_BUILD_TYPE=Release $GEANT_DIRECTORY
    else
        echo -e "\n CMAKE with original CILKPLUS VERSION  \n"
```

```
        cmake -DUSE_ICC=ON -DUSE_CILK=ON -DUSE_AVX=ON -
DVECGEOBENCHMARKS=ON -DCMAKE_BUILD_TYPE=Release $GEANT_DIRECTORY
    fi
    make
    export LD_LIBRARY_PATH="./lib/":$LD_LIBRARY_PATH
    echo "===================================="
    echo "Configured and compiled"
}

# GCC Configuration of Geant Vector Prototype
function gcc() {
    echo "Compiling with GCC"
    echo -e "\nBuilding in directory: $OUTPUT \n"
    source $ROOT_DIRECTORY/bin/thisroot.sh
    echo $MODULEPATH
    #module add cilkplus/4.8
    module add cilkplus/4.8-fixed
    #module  add vc
    module add tbb
    dir_build=$OUTPUT
    mkdir $dir_build
    cd $dir_build
    export GEANT_OUTPUT_DIR=$PWD
    export LDFLAGS="-lrt"
    if [[ $1 == "original" ]]
    then
        cmake -DUSE_GCC=ON -DVECGEOBENCHMARKS=ON -DUSEAVX=ON -
DCMAKE_BUILD_TYPE=Release $GEANT_DIRECTORY
    else
        cmake -DUSE_GCC=ON -DUSE_CILK=ON -DVECGEOBENCHMARKS=ON -
DUSEAVX=ON -DCMAKE_BUILD_TYPE=Release $GEANT_DIRECTORY
    fi
    make
    export LD_LIBRARY_PATH="./lib/":$LD_LIBRARY_PATH
    echo "===================================="
    echo "Configured and compiled"
}

compiler=$1
version=$2

if [[ $compiler == "gcc" ]]
then
    gcc $version
elif [[ $compiler == "icc" ]]
then
    icc $version
else
    echo "Usage: ./configure_geant.sh <gcc|icc> [original|cilk]"
fi
```

Acknowledgements

This work was carried out under the CERN openlab summer student program 2013 in collaboration with Intel, Intel CilkPlus Team. As part of Intel: Laurent Duhem, Jeff Arnorld, Klaus Dieter Oertel, Georg Zitzlsberger, Hans Pabst, Martyn John Carden, Barry M Tannenbaum, Pablo Halpern and John Pierer . Also we would like thank Balaji Iver from GCC CilkPlus Team as well. The Geant Vector Prototype was followed and advised by Sandro Wenzel, Federico Carminati and John Apostolakis from the Physics Department at CERN.

8 References

[1] Sverre Jarp, Afio Lazzaro, Julien Leduc, Andrzej Nowak, Liviu Valsan. *Report on parallelization of MLfit benchmarch using MPI and MPI.* CERN openlab July 2012.

[2] Sverre Jarp, Afio Lazzaro, Andrzej Nowak, Liviu Valsan. *Comparison os Software Technologies for Vectorization and Parallelization*. CERN OpenLAB 2012 – Whitepaper.

[3] John L. Hennesy, David A. Patterson. *Computer Architecture. Quantitative Approach*. Fifth Edition (The Morgan Kaufmann Series in Computer Architecture Design).

[4] LLVM CilkPlus: http://cilkplus.github.io/

[5] Martyn Corden. *Requirements for Vectorizing Loops with #pragma SIMD.* Intel Articles 2012.

[6] John Apostolakis, René Brun, Federico Carminati and Andrei Gheata. *Rethinking particle transport in the many-core era towards GEANT 5*. J. Phys.: Conf. Ser. 396 022014

[7] *Intel® Next Generation Microarchitecture. Codename Haswell: New Processor Innovations*. Intel Developer Forum 2012.

[8] Sverre Jarp, Alo Lazzaro, Julien Leduc, Andrzej Nowak and Felice Pantaleo. *Parallelization of maximum likelihood fits with OpenMP and CUDA*. CERN Openlab 2011-009.

[9] Vc Library website: http://code.compeng.uni-frankfurt.de/projects/vc

[10] Sverre Jarp, Alfio Lazzaro, Julian Leduc, Andrzej Nowak, Livio Valsan. *Report on parallelization of MLfit benchmark using OpenMP and MPI*. CERN openlab, July 2012.