Evaluation of OpenCL for High Energy Physics Event Reconstruction

T. Hauth[*], D. Piparo[†] and V. Innocente[‡]
CERN

March 14, 2012

Abstract

The Open Computing Language (OpenCL) standard is evaluated in the context of a selected track reconstruction algorithm present in the software of the CMS experiment. The performance of the Intel, AMD and NVDIA OpenCL platforms on both GPU and multi-core CPU devices of different vendors are compared to a reference OpenMP implementation. In addition, the portability of OpenCL programs is investigated and the usability of the OpenCL API is assessed.

All the tested OpenCL platforms which run the benchmark compute kernel on the CPU show performance results which are comparable with the OpenMP reference implementation. The same kernel running on NVIDIA and AMD GPUs is faster than on the Intel and AMD CPUs, even when spawning a significant number of threads. The scheduling and data transfer overheads with OpenCL is measured and figures of merit about how it can be amortized are provided.
The performance of OpenCL is shown to be portable across CPU and GPU devices.
A significant improvement of the usability of the OpenCL technology was achieved via a C++ wrapper to make the access to the OpenCL API more convenient than the one natively offered by the standard.

[*]thomas.hauth@cern.ch
[†]danilo.piparo@cern.ch
[‡]vincenzo.innocente@cern.ch

Contents

Executive Summary

To cope with the increasing computing demands implied by the future Physics program of the Compact Muon Solenoid (CMS) experiment at the Large Hadron Collider (LHC), promising new technologies are evaluated on a regular basis. In this report the OpenCL standard is considered and the evaluation of its Intel, NVIDIA and AMD implementations are described in terms of overall compute performance, portability and usability.
OpenCL is a standard which defines a framework and a programming language for parallel computation on heterogeneous systems. It is supported by a large consortium of industry leaders in the software and hardware field and has been designed from the very beginning to support computations on Graphics Processing Units (GPU) and Central Processing Units (CPU).
An algorithm part of the track reconstruction chain of CMS events was selected as a standard candle for this report. To provide a reference using a well-established technology, the same algorithm was implemented using the OpenMP API.

The OpenMP and OpenCL measurements on the CPUs show comparable runtime performance and smoothly scale with the number of available cores in the CPU. If on the one hand the OpenCL platform adds a higher scheduling overhead with respect to OpenMP, on the other hand this cost is well amortized when an adequate workload, i.e. at least an input collection of 1,000 tracks, is provided to the single threads.
The benchmark algorithm could be run on all OpenCL platforms without modifications to its implementation. Taking advantage from the portability offered by the OpenCL technology, the algorithm has been tested on NVDIA and AMD GPUs. The time necessary to process exactly the same input data on the GPU was about the half of the one measured using all the cores of the CPU.
To reach a level of usability comparable to regular C++ programming, a consistent and easy-to-use C++ wrapper around the OpenCL API was developed. The OpenCL memory objects are encapsulated in compile-time type-checked C++ classes featuring an automated resource management. Therefore, the usage of the OpenCL technology becomes simpler and less error prone with respect to the one offered by the native OpenCL API.

Expressing parallelism with the OpenCL framework in selected CMS event reconstruction algorithms has proven to be possible. The performance scales properly with the number of available CPUs, provided that enough work per thread is supplied in order to amortize the scheduling overhead. The performance portability of OpenCL not only makes the usage of GPU resources transparent for the developer, but elegantly avoids vendor lock-in and allows the flexibility that a large and long-running project like CMS software needs. Therefore, the OpenCL technology is for CMS one appealing way to take advantage of herogeneous computing resources in the future many-core era.

1 Introduction

OpenCL is a standard which defines a framework, an API and a programming language for parallel computation on heterogeneous systems like client computer systems, high-performance computing servers as well as hand-held devices. The standard is maintained by the Khronos Group and supported by a large consortium of industry leaders including Apple, Intel, AMD, NVIDIA and ARM. Influenced by NVIDIA's CUDA from the GPU side and by OpenMP which originates from the classical CPU side, the open OpenCL standard is characterized by a formulation which is abstract enough to support both CPU and GPU computing resources. This is an ambitious goal, since providing an abstract interface together with a peak performance is a challenging task. OpenCL employs a strict isolation of the computation work into fundamental units, the kernels. These kernels can be developed in the OpenCL C programming language, a subset of the C99 language, with some additional OpenCL specific keywords. In general, these kernels are hardware independent and compiled by the OpenCL runtime when they are loaded. To be able to fully exploit the parallel execution of the kernel code, several kernel instances, the work items, are started to process a set of input values. The actual number of concurrently running work items is determined by the OpenCL system. How a concrete algorithm can be partitioned into work items has to be decided by the programmer.

The Compact Muon Solenoid (CMS) collaboration has foreseen a rich and ambitious Physics program based on the analysis of the data originating from the collisions delivered by the Large Hadron Collider (LHC). The reconstruction of the events recorded by CMS, is carried out with an object-oriented C++ software framework, CMSSW. With the increasing luminosity provided by the LHC and the future scenarios implied by the super-LHC collider, the treatment of the collision data will require extraordinary large computing resources, also in terms of CPU usage. The trivial event-based parallelism will not be enough to cope with this challenge. Therefore, new promising technologies in the field of computer hardware and software are evaluated on a regular basis within the CMS collaboration in order to assess their potential advantages in terms of improvement of the processing of the collision data. The purpose of this report is to summarize the findings gained in a first evaluation of the OpenCL technology.

2 Evaluated hardware and runtime environments

The specific implementations of the OpenCL runtime are provided by the hardware vendors. In this report, four different hardware architectures, coming with a different OpenCL runtimes, are evaluated.

2.1 Intel OpenCL SDK

As part of the OpenCL consortium, Intel played an important role in creating the OpenCL standard. The Intel OpenCL Software Development Kit (SDK) compiles OpenCL kernels in order to run them on x86-64 CPUs. Code vectorization is exploited to distribute the calculations of a kernel to the vector units of a CPU. The most recent release of the SDK (version 1.5), supports the SSE and AVX vector instruction sets.

The performance benchmarks described in this document are based on the Intel

OpenCL SDK 1.5 for 64-bit Linux. The hardware on which they are run is an Intel Core i7-3930K CPU at 3.20GHz machine, with 6 physical and thus 12 hyper-threaded cores. The RAM amounts to a total of 16 GB and the operating system is Scientific Linux 6 (SLC6).

2.2 NVIDIA OpenCL

One of the forerunners of the OpenCL standard is the CUDA system for GPUs by NVIDIA. CUDA also provides a framework and dedicated programming language to run compute kernels. Since 2009, the NVIDIA graphics driver also supports the OpenCL standard and can compile and run OpenCL kernels on the graphics card. For the evaluations described in this paper, the NVIDIA Linux driver version 275.43 was installed. The graphics card at disposal was a commodity one: the NVIDIA GeForce GTX 560 with 1.5 GB on-card RAM and 336 CUDA compute cores.

2.3 AMD APP OpenCL SDK

AMD is the first major vendor providing OpenCL support both for its own CPUs and GPUs with one runtime environment: the AMD Accelerated Parallel Processing (APP) SDK. The version used for the tests figuring in this document is the latest available version at the time of writing: the AMD APP SDK v2.6.

The machine used for the benchmarks presented is an AMD FX-8120 CPU which is based on the AMD Bulldozer microarchitecture and has 8 cores. As the machine equipped with Intel hardware, this test machine also has 16 GB of RAM and runs SLC6.

The graphic card mounted was the AMD Radeon HD 6970, and it was used to evaluate the GPU runtime of the AMD OpenCL SDK. This card is also aimed at the gaming market and mounts 2 GB on-card RAM and 1536 Stream Processors. Note that number of Stream Processors cannot directly be compared to the CUDA compute cores of the NVIDIA card, as these entities may differ in implemented functionality. To access the card with OpenCL the graphics driver AMD Catalyst 11.11 revision 12.1 for 64-bit Linux has been installed.

2.4 OpenMP

In order to have a relation of the OpenCL benchmark results with a more conservative and established technology, the benchmarked algorithms were also coded in C++ taking advantage of OpenMP. The adopted compiler was the most recent revision of the GCC 4.7 trunk.

3 The CMSSW use case: exploiting parallelism on track level

The reconstruction of a real-life HEP event consists in running a wide range of algorithms which include cluster finding, pattern recognition, linear algebra mathematics and minimizations. These algorithms elaborate very different types of data, organized in various categories of structures.

At the time of writing, the reconstruction of the charged particles tracks is by far the major contributor to the total runtime of the CMS data processing. This is due to the

complexity of the CMS silicon tracker detector and to the high number of charged particles originating from a proton-proton collision. Taking into account the expected LHC run conditions in the year 2012, the number of particle tracks in an average proton-proton collision is between 500 and 1,000 and this number is likely to increase even further in the future.

From a computational perspective, often the same algorithms are applied under similar conditions to all the tracks in an event. This behavior offers an opportunity to express data-parallelism exploiting vector units and multi-threading. One representative real-life use case from the actual CMS tracking reconstruction software is picked to perform the runtime and scaling tests.

3.1 The multiple scattering algorithm

The calculation of the multiple scattering of a charged particle in the detector material was chosen as the standard candle for the tests described in this document. The calculation is composed of about 40 mathematical operations on double precision floating point values and one call to the logarithm mathematical function. During the reconstruction process of one recorded event, this calculation has to be performed thousands of times for all tracks.

4 Multiple scattering runtime

In the following, the multiple scattering algorithm is applied to an input collection of 100, 1,000 and 10,000 tracks.

Both the OpenMP and the OpenCL implementations of the algorithm use the same C++ code fragment to perform the calculations. For OpenCL, this code is defined using the framework described in detail in section 5. The same OpenCL memory layout is used to benchmark the Intel OpenCL SDK, NVIDIA CUDA Driver and AMD APP OpenCL. For OpenMP, the computation is contained within a regular `for` loop and parallelized using the directive

```
#pragma omp parallel for
```

4.1 Intel machine: performance measurement

The calculation has been performed 1,000 times and figure 1 shows the overall runtime of all three implementations for 100, 1,000 and 10,000 tracks. The measurement has been performed for various numbers of threads for the Intel OpenCL and OpenMP implementations. As there is no comparable equivalent to CPU threads on the NVIDIA GPU, all available computing resources on the NVIDIA card have been used. The time interval from submitting the OpenCL compute kernel to the the arrival of the results to the application is reported.

The OpenMP reference implementation has good performance and scales well up to about 4 threads.
The OpenCL NVIDIA implementation running on the GPU shows the best performance of the test for the 10,000 tracks case.
For what concerns the Intel OpenCL SDK, the single thread measurement shows about the same performance as the OpenMP case. The performance reasonably scales up to four

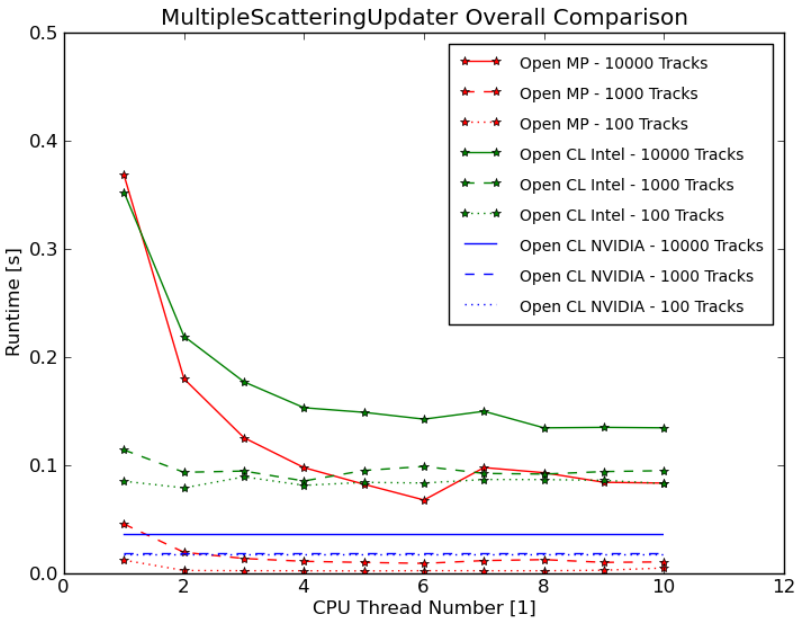


Figure 1: Overall runtime comparison for all three evaluated technologies for 100, 1,000 and 10,000 tracks on the Intel machine for 1000 repetitions of the process. A structure is clearly visible at 6 threads. This is due to the fact that the machine has 6 physical cores and after this number of threads the Hyperthreading technology comes into play.

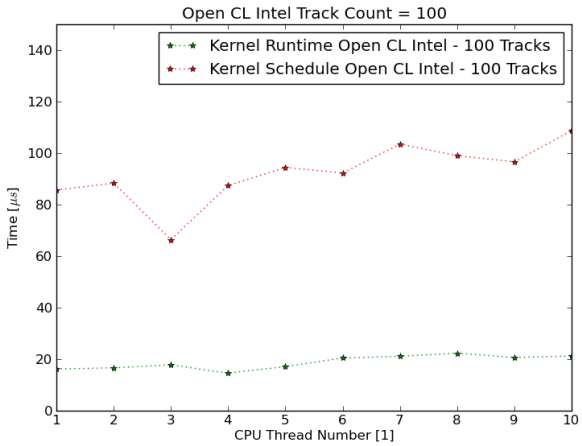
cores. After this point, no additional improvement is visible. In general, the OpenMP implementation can make better use of additional cores than the Intel OpenCL one.
For the 100 tracks case, multithreading basically does not bring any improvement. This behavior hints to a significant scheduling overhead. This assumption will be quantified in the next section.

4.2 A measurement of the OpenCL scheduling overhead

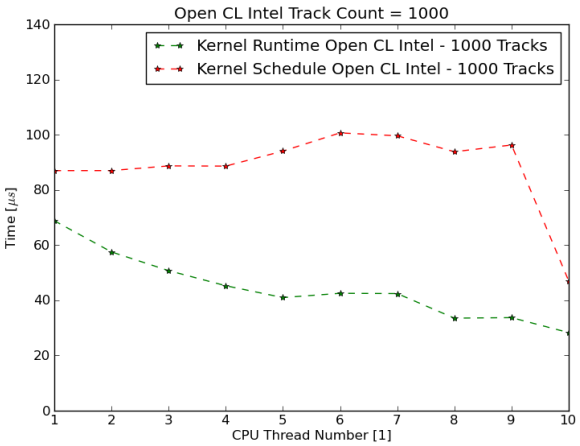
To be able to separate the scheduling overhead from the actual runtime of the single OpenCL kernels, OpenCL's `clGetEventProfilingInfo()` was exploited. The scheduling time and the actual runtime can be queried by this function after the kernels execution is complete. For this benchmark, the profiling has been enabled on the OpenCL platforms. The timings were gathered for 1,000 independent tests and their means are plotted.

As visible in figure 2, except for the 10,000 tracks case, the scheduling time of the kernel on the Intel platform is larger than the actual runtime of the kernel. Only for the case of 10,000 tracks, a significant improvement of the kernel runtime depending of the CPU thread number is visible. Independent of the number of threads or the number of input tracks, the scheduling overhead lies for this measurement in a region at around 90-100 μs.

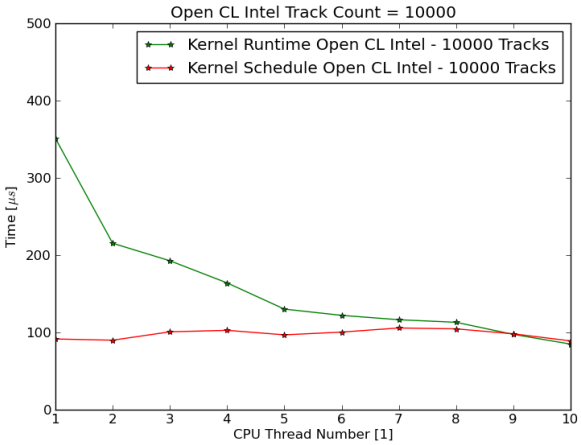
Figure 3 illustrates that the NVIDIA runtime also has a constant scheduling overhead when running the OpenCL kernel. This overhead is of about 4 μs and represents only the



(a) 100 Tracks



(b) 1,000 Tracks



(c) 10,000 Tracks

Figure 2: Mean elapsed scheduling and running time of the multiple scattering kernel running on the Intel OpenCL Platform. The mean was calculated on 1000 independent tests.

5% of the scheduling overhead observed with the Intel OpenCL SDK.

4.3 AMD machine: performance measurement

The same benchmarks described in section 4.1 were performed on the AMD machine. Figure 4 shows the timing results for the 1,000 and the 10,000 tracks case.

The Intel OpenCL SDK has also been installed on the AMD machine and the obtained timings are inserted into the plot.
The AMD APP SDK does not support running kernels only on a subset of the overall available processor cores. Therefore, all AMD APP measurements on the CPU imply the usage of all of the eight cores of the CPU. As it was the case for the NVIDIA GPU, the computing units of the AMD GPU are not limited in any way.
Compared to the Intel machine, the OpenMP performance is slightly degraded on the AMD machine. The performance of the Intel OpenCL runtime on the AMD machine is about 10% worse than the one measured on the Intel machine. On the other hand, the AMD OpenCL CPU kernel is faster than Intel OpenCL for the maximum number of available threads, even taking into account the difference in real cores per die.
The AMD OpenCL running on the AMD GPU is the fastest of all investigated implementations, including the NVIDIA GPU.

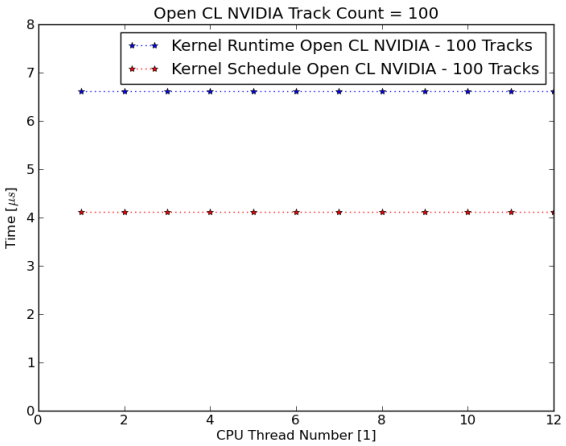
4.4 Data transfer overhead

One important aspect of the OpenCL system is its memory model. Depending on the actual hardware used, the memory accessible by the kernel is either located on the host's main memory or dedicated memory on the graphics card. In general, input values must be transferred to dedicated OpenCL memory banks before any kernel has access to them. After the kernel has performed its calculations, the resulting values stay in the OpenCL memory realm and subsequent kernels can be started. If the results must be available to the regular C/C++ program on the host side, an explicit memory copy operation has to be performed.

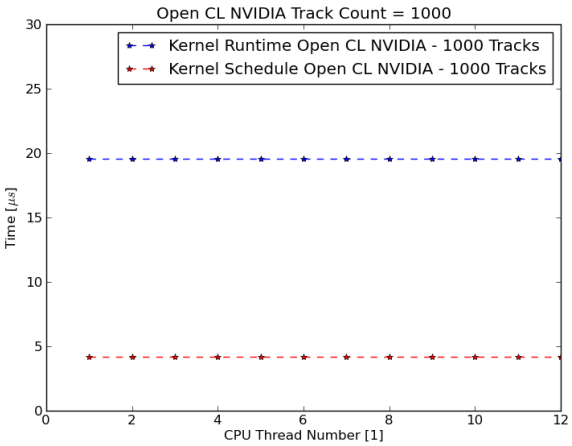
Up to now, the additional time necessary to perform this memory transfer from and to the OpenCL device has not been taken into account. The measured times reflected the speed if all necessary data structures are already in place. This is the most generic measurement which can be performed, as the complexity and frequency of memory transfers highly depends on the workflow of the underlying application. The main factors here are how many different kernels are run sequentially and how well data structures on the OpenCL device can be reused by the kernels. In this case, four doubles are shipped to the computing units and three doubles are taken back.

To give an estimate of the expected overhead, we examine the worst-case scenario. All track-data is copied to the OpenCL memory space before each kernel run and is copied back to host memory once the computation is complete. We repeat the measurement done in Section 4.1 and include the transfer overhead in the measured runtime. The results can be seen in Figure 5 and show the expected increase in the runtime of the OpenCL implementations compared to the previous measurement (Figure 1). For the 10,000 tracks case, the OpenCL runtime on the NVDIA GPU was more than doubled, the runtime of the Intel OpenCL implementation increases by 25% for the single core case.

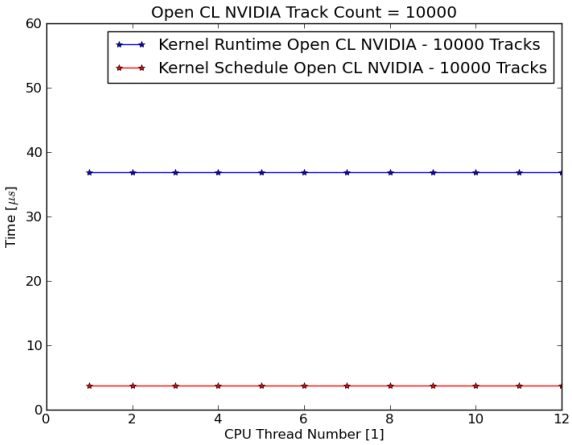
This measurement shows, that it is essential to keep the data structures as long as possible in the OpenCL memory realm, apply as much computation as possible and only copy the final results back to host.



(a) 100 Tracks



(b) 1,000 Tracks



(c) 10,000 Tracks

Figure 3: Mean elapsed scheduling and runtime time of the multiple scattering kernel running on the NVIDIA OpenCL Platform. The mean was calculated on 1000 independent tests.

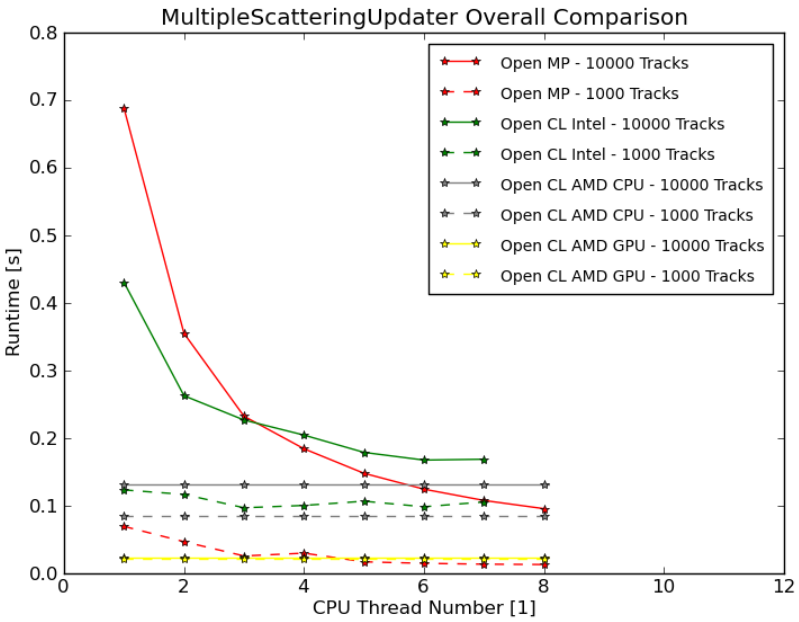


Figure 4: Overall Runtime comparison of all the available technologies for 1,000 and 10,000 tracks on the AMD machine.

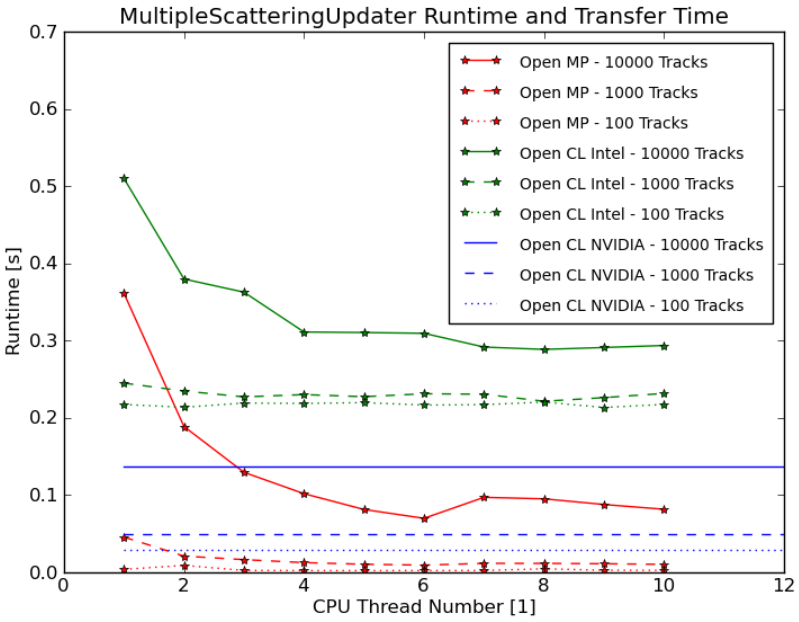


Figure 5: The Runtime including the transfer overhead for all three evaluated technologies for 100, 1,000 and 10,000 tracks on the Intel machine for 1000 repetitions of the process.

As the kernels running on Intel OpenCL are executed on the CPU and have therefore

access to the whole memory of the host system, the OpenCL standard offers a feature to reuse the host memory in OpenCL kernels. Therein, pointers to regular C/C++ data structures, like a `std::vector` of doubles, can be registered with the OpenCL framework and kernels running on the CPU can directly access this memory. Therefore, by using this technique on the CPU, no explicit memory transfer is necessary and no overhead with moving data from host memory to the OpenCL memory realm occurs.

4.5 Compiled Performance Numbers

Table 1 shows all numbers gained during the studies in the previous sections. It is important to note, that although these numbers reflect a general trend in the system runtimes, other parameters discussed before, like scalability must also be taken into consideration.

Table 1: Overview of the runtime of all investigated systems for 1,000 iterations of the multiple scattering algorithm

System under test	Only Runtime [ms]	Runtime with Transfer [ms]
Intel Machine		
Intel OpenCL (6 CPU cores)	140	310
OpenMP (6 CPU cores)	70	-
NVIDIA OpenCL	40	130
AMD Machine		
Intel OpenCL (all CPU cores)	160	-
AMD OpenCL (all CPU cores)	140	-
OpenMP (all CPU cores)	90	-
AMD OpenCL AMD GPU	25	-

4.6 Auto-Vectorization with OpenMP

In the process of implementing the OpenMP reference version, the gains were investigated which can be achieved by using the auto-vectorization features of GCC.
Modern CPUs present several vector units, the capacity of which is growing steadily with the introduction of new processor generations. Most recent C++ compilers can take advantage of such innovations, either by explicit statements in the sources of the program or automatically adapting the generated machine instructions to the available hardware, without the need of modifying the existing code base.
Due to a call to the logarithm function, GCC is not able to auto-vectorize the existing multiple scattering kernel source code as-is. To overcome this, a different double precision logarithm implementation was used. This code is part of a collection of double and single precision inline autovectorizable mathematical functions developed within CMS starting from the well-known Cephes[1] library.

[1] `http://www.netlib.org/cephes`

Figure 6 shows the auto-vectorized version compiled to use the SSE2 instruction set compared to the scalar version. The measurements shown in this plots were performed on the Intel machine which is described in section 4.1. The benefit brought by the usage of the CPU vector units is clearly visible and is not affected in a negative way by the number of available threads.

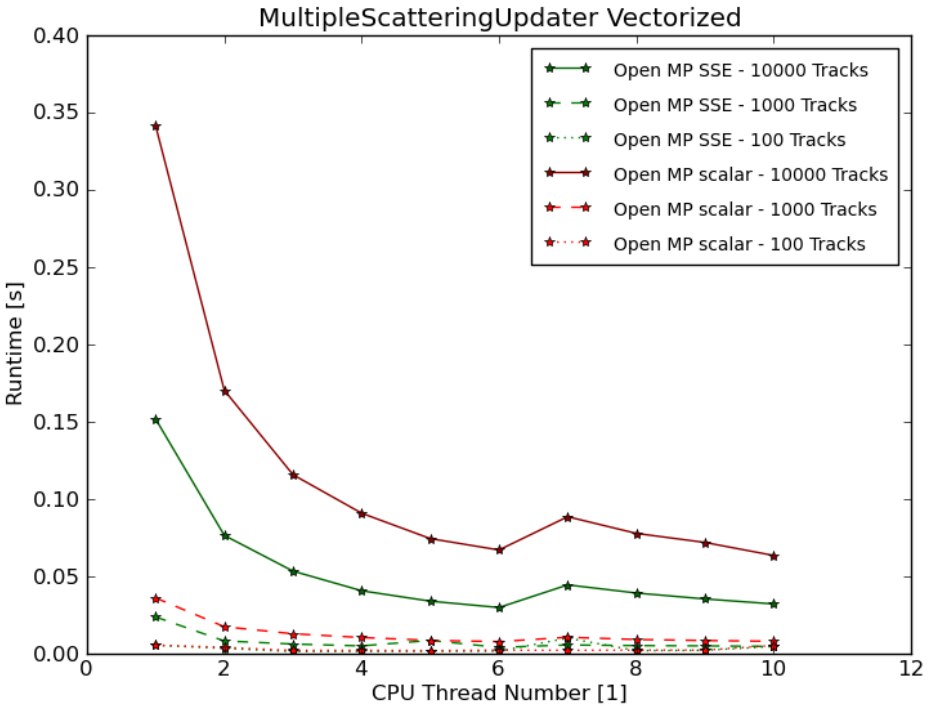


Figure 6: Auto-vectorized version of the OpenMP implementation compared to the scaler version on the Intel machine.

5 A possible simplification of OpenCL development

A relevant parameter taken into account by CMS during the evaluation process of new software technologies is their usability. The responsibility of the development and maintenance of the CMS reconstruction code is spread among many developers from various Research Institutions and Universities and their programming skills are far from being homogeneous. Therefore, to ease the development process and to keep the source code consistent, the complexities linked to the employed multi-threading technology should be encapsulated as best as possible.

The API offered by OpenCL is a C-style one, which hands out handles and requires explicit memory management. Moreover, OpenCL kernels cannot be defined as part of the regular C++ code of a program but can only be contained in string variables or read from external files. Therefore, no syntax or type checking of the kernel program code is natively performed at compile time of the host application.

The openclam[2] project tries to mitigate these issues by encapsulating the OpenCL API in C++ classes. Taking advantage of smart C++ templating and preprocessor statements, OpenCL kernels can be defined interleaved with regular C/C++ program code and are syntax and type checked by the compiler, in our case GCC. Unfortunately, the activities linked to the openclam project showed a setback in the last year and the project is still more a proof-of-concept than a finished product. Nevertheless, openclam acted as a basis and has been heavily extended and refactored by the authors of this paper.

In the following, several aspects of the OpenCL experience are discussed from the programmer's point of view. The code snippets shown present the aforementioned OpenCL programming interface. Given the abundance of examples offered on the web, the equivalent OpenCL API code that would be necessary to achieve the same functionality is not shown.

5.1 Platform management and OpenCL compute context

To simplify the instantiation of the OpenCL runtime, the C++ classes `clam::opencl` and `clam::context` are available. If no parameters are given, the first available OpenCL runtime on the machine will be selected. Specific platforms can be selected by the vendor name or by defining whether a GPU or CPU should be used.
The release of all to the compute context associated resources is automatically performed once the C++ destructor of the `clam::context` object is called.
Figure 7 to Figure 9 show some examples of how the `clam::context` C++ class can be used to access various computing hardware.

```
// create wrapper for OpenCL API calls
openclam::opencl wrapper;
// create compute context on default platform
openclam::context context(wrapper);
```

Figure 7: Instantiation of an OpenCL Compute Context on the default Platform.

```
// create wrapper for OpenCL API
calls openclam::opencl wrapper;
// create compute context on default platform
openclam::context context( wrapper,
                           openclam::opencl::PlatformNameNVIDIA());
```

Figure 8: Instantiation of an OpenCL Compute Context on the NVIDIA Platform.

5.2 Memory buffer management and data transfer

The OpenCL concept of memory management is explicit. This means that memory regions for output and input values must be allocated by calls to the OpenCL API and

[2] `http://code.google.com/p/openclam`

```
// create wrapper for OpenCL API calls
openclam::opencl wrapper;
// create compute context on Intel platform
openclam::context context( wrapper,
                           openclam::opencl::PlatformNameIntel(),
                           // only select CPUs
                           openclam::icontext::cpu,
                           // compile options for CL Kernels
                           "-cl-fast-relaxed-math",
                           // options to the OpenCL command queue
                           0,
                           // disable profiling of kernels
                           false,
                           // limit the use of CPU cores to 4
                           4);
```

Figure 9: Instantiation of an OpenCL Compute Context on the Intel Platform and using 4 CPU cores.

memory from within the host C++ program's heap or stack must be transferred from and to the OpenCL memory regions explicitly.
This form of low-level memory management could remind of the C `malloc` and `free` functions. Unfortunately, this form of memory management is characterized by very similar caveats and pitfalls. No type safety during the access to OpenCL memory regions is enforced and the validity of the accessed range is not checked. This can lead to catastrophic memory violations or simply faulty calculations which are very hard to spot and debug. The manual transfer of input values might result cumbersome and is more or less a variation of a crude C-style `memcpy` command. Moreover, OpenCL buffers must be explicitly released to free the associated memory.

For large-scale applications like CMSSW, where OpenCL buffers could need to be allocated and used within different modules, a direct usage of the OpenCL memory management system would be very hard to implement and maintain. For this reason, a C++ template-based system to allocate, use and release OpenCL buffers was developed. The application developer is relieved of the low-level memory handling but can use templated and type-safe C++ classes to pass memory from and to the OpenCL compute kernels. The device memory associated with an OpenCL buffer is automatically released once the C++ object is destroyed. Therefore, no additional effort is necessary to ensure a consistent memory management except following the regular C++ principles.
For the tests described in this document, this mechanism was used to provide a vector and a matrix data type but this idea can be extended to every type of data. The code snippet 10 shows how to create a square 10-dimensional matrix of double precision numbers and to transfer it to an OpenCL compute context.

```
// define a Matrix datatype for our case
typedef openclam::matrix<double,10> Matrix;
// create a std::vector to have initial values
std::vector < double> arr( Matrix::value_elements, 1.0 );
// create one matrix with the intial values from the std::vector
Matrix m1 ( arr, 1, wrapper, context );
```

Figure 10: Allocation of a custom matrix data type on an OpenCL compute context.

5.3 Defining OpenCL Compute Kernels

As mentioned before, at the time of writing, OpenCL kernels cannot be natively interleaved with the regular C++ code, but must be provided as external text files or as string variables. Furthermore, the parameters of a kernel call must be set one by one via the OpenCL API: depending on the amount of parameters this procedure can easily span several lines of C++ code.

Starting from the openclam's framework, we created a system to conveniently define OpenCL kernels. They are syntax and type checked during the compilation of the host program and can be called with all parameters in one single line of C++ code. Figure 11 shows how a simple kernel with two parameters can be defined.

```
KERNEL2_CLASS( kernel_add_scalar, const cl_mem, const double,
 __kernel void kernel_add_scalar( __global double * a,
                                  const double b)
{
   unsigned int x = get_global_id(0);
   a[x] += b;
} );
```

Figure 11: Definition of a simple kernel with two parameters. The kernel only adds a constant value `b` to the values contained in the input array `a`.

As soon as the C++ class containing this kernel definition is instantiated, the kernel code is compiled using the OpenCL runtime and registered as `kernel_add_scalar`.
More complex kernels with more parameters have been created using this scheme but are not presented here for the sake of compactness. These kernels include matrix multiplication algorithms or rather complex calculations (see below) with more than 100 instructions and up to 6 parameters.

5.4 Running Kernels

Running OpenCL kernels in the scheme described above is straightforward. The `run()` method of the defined kernel can be called. This method takes the parameters which are passed to the kernel and allows to specify the amount and size of the OpenCL work dimensions. Figure 12 shows how to call an already defined kernel.

```
kernel_add_scalar.run( matrix.range_linear(), matrix.mem_,
                       5 );
```

Figure 12: Calling an already defined kernel.

5.5 Full Kernel Example code

Figure 13 shows how to define and call a kernel. The C++ class `matrix_add_scalar` wraps the kernel definition. The `apply(...)` method can now be called by external code to run the kernel.

```
struct matrix_add_scalar : private boost::noncopyable
{
   KERNEL2_CLASS( kernel_add_scalar, const cl_mem, const double,

   _kernel void kernel_add_scalar( __global double * a,
                                   const double b) {
      unsigned int x = get_global_id(0);
      a[x] += b;
   } );



   explicit matrix_add_scalar(openclam::icontext const& context) :
      kernel_add_scalar(context) {}

   template<class TMatrix, class TScalar>
   void apply( TMatrix const & matrix,
               TScalar const& scalar) const {
      kernel_add_scalar.run( matrix.range_linear(),
                             matrix.mem_, scalar );
   }
};
```

Figure 13: Full example of defining and calling a kernel.

6 Conclusions

6.1 Performance

As the benchmarks in section 4 show, all OpenCL implementations exhibited a scheduling overhead. The Intel OpenCL runtime performance reaches the results of the OpenMP reference implementation for the single core case, but does not scale as well with the number of threads. The scheduling overhead is a relevant factor especially for the Intel's OpenCL.
The NVIDIA OpenCL implementation also shows some scheduling overhead, but far less then the Intel version (5 μs with respect to 90-100 μs in the tests performed). Overall, the NVIDIA implementation shows a better performance compared to the OpenMP reference implementation.
The AMD APP OpenCL SDK on the AMD machine is also well behaved in terms of performance. The AMD OpenCL GPU benchmarks show a very small scheduling overhead and the kernel run on the AMD GPU reaches the lowest runtime of all the tests.

The current benchmark scenario used various numbers of input particle tracks to evaluate the runtime of the implementations. For both, the well-established OpenMP and the newer OpenCL, it becomes clear that a sufficient amount of input must be available to allow a profitable track-based parallelism. The work-chunks which can be handed to individual cores must be large enough to allow for a distribution of the load to all available cores. If this work-chunks are to small, the whole working set cannot be parallelized beyond two or three cores.

Section 4.4 showed that, depending on the memory usage pattern of the application, an overhead in transfering data from and to the OpenCL devices can be observed. Moreover, as described in section 4.2, OpenCL imposes a scheduling overhead which is associated with starting the kernels. This overhead is not linked to the number of input elements and therefore is constant. This scheduling overhead can be amortized with processing of several input elements with one kernel run.

Therefore, if the OpenCL or OpenMP technology would be used to speed up the CMS tracking reconstruction step, a sufficient amount of tracks must be gathered together to make the parallel processing profitable. One possible way to achieve this is to reconstruct the particle tracks from different events at once.

6.2 Portability

For the tests described throughout this document, the OpenCL technology has shown a remarkable portability. It can be considered as a powerful tool to implement algorithms to run both on CPUs and GPU resources transparently.

6.3 Usability

By providing a consistent and easy-to-use C++ wrapper around the OpenCL API, the level of simplicity of programming with regular C++ classes was almost reached. In particular, the encapsulation of the OpenCL memory buffers in templated C++ classes, fully type-checked and featuring an automatic releasing of the resources, allows to greatly reduce the complexities and potential mistakes dealing with OpenCL memory management. The definition of the OpenCL kernels within regular C++ code was demonstrated to be very convenient for the application developers. Further functionalities and data types can

be added to the currently developed C++ wrapper. Therefore, from the usability standpoint, there are no major obstacles in using OpenCL in the CMS event reconstruction code.