Evaluation of OpenCL for High Energy Physics Event Reconstruction

Thomas Hauth
Danilo Piparo
Vincenzo Innocente

13-3-2012

For more details please have a look to the supporting document in the indico agenda!

Motivation

- Data processing software of HEP experiments:
  - Satisfy the needs of extremely ambitious Physics programs
  - Fit on the available computational resources (e.g. Tier0,1s,2s, trigger farms, laptops)
- New software technologies are crucial in this environment!
- CMS evaluates these innovations on a regular basis (e.g. compilers, allocators …)
- Today we will discuss one among them, OpenCL

The evaluation of this product goes along three lines: performance, portability and usability

Open Computing Language

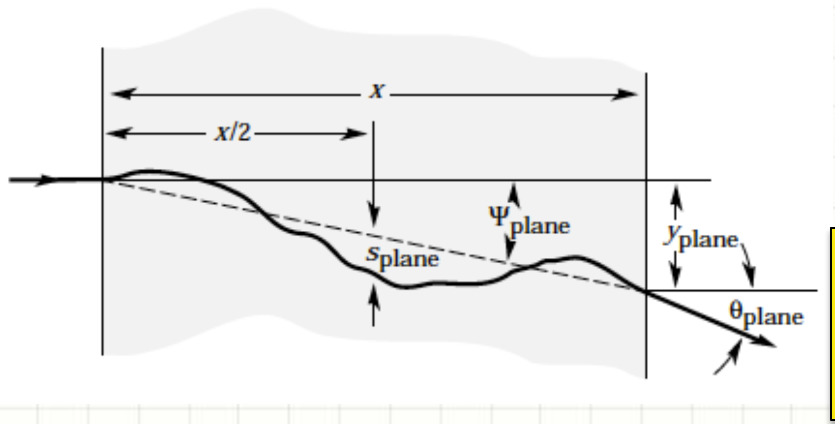
- Idea: allow programmers to write portable programs that use all resources in a heterogeneous platform (e.g. CPUs, (GP)GPUs, handheld devices, FPGAs).
- Mix data parallel and task parallel code in the same application.
- Maintained by the Khronos group and supported by many leading hardware and software vendors (Apple, NVIDIA, AMD, …)
- Open and Royalty-free
- OpenCL: a framework + a programming language (C99+limitations+additions)
  - IEEE 754 numerical accuracy for all fp operations available
- Abstracted memory and execution model:
  - Basic units of executable code, *kernels*, dispatched to the *Computing Units* (CUs)
  - Run the same code on CPUs and GPUs
- Explicit memory model (private, shared and global mem spaces)

OpenCL allows to run computations on heterogeneous platforms

See references and backup for more details

The "candle" used for this study

- Algorithm from the CMS tracking code (MultipleScatteringUpdator)
  - Calculate the maximum scattering angle of a particle passing through a material (silicon) layer.
  - Implementation of the Highland formula for multiple Coulomb scattering.
- Called several times for a single track
- Useful figure: 500-1000 tracks to be expected in an average LHC event in 2012
- In terms of mathematical operations:
  - Multiplications, divisions, sums and a logarithm.
  - About 40 lines of code, 1 branching
  - I/O: 4 double precision floating points in, 3 of them out.



An algorithm from the CMS software framework taken as candle

See references and backup for more details

Hardware used for the test

Intel CPU + NVIDIA graphics card:

1800 CHF

- Core i7-3930K @ 3.20GHz (AVX support)
  - 6 physical, 12 hyper-threaded cores
  - RAM: 16 GB
- NVIDIA GeForce GTX 560 – **250 CHF**
  - 336 CUDA compute cores*
  - 1.5 GB on-card RAM
  - NVIDIA Linux driver version 275.43
- Scientific Linux 6

AMD cpu + ATI graphics card:

1800 CHF

- AMD FX-8120 CPU – Bulldozer microarchitecture (AVX support)
  - 8 cores
  - RAM: 16 GB
- AMD Radeon HD 6970 – **300 CHF**
  - 1536 Stream Processors*
  - 2 GB on-card RAM
  - AMD Catalyst 11.11 Linux driver, revision 12.1
- Scientific Linux 6

* These numbers are **not** directly comparable!

The OpenCL SDKs used

- Intel SDK
  - Version 1.5 for 64-bit Linux
  - SSE and AVX instruction sets support
- NVIDIA SDK
  - NVIDIA Linux driver version 275.43
- AMD Accelerated Parallel Programming SDK:
  - Version 2.6 for 64-bit Linux
  - Supports both AMD CPUs and GPUs
  - No support for limiting the number of CPUs used
  - SSE and AVX instruction sets support

All these SDKs use the LLVM compiler infrastructure.

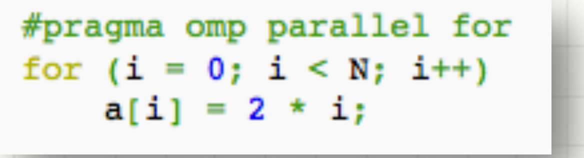
For the results presented in this report, the GPU has not been partitioned but considered on as one single compute entity.

Moreover no special optimisations were put in place: use OpenCL "out of the box".

Compute Performance and Performance Portability

Reference Implementation

- In order to have a reference for the benchmark a well-established technology was used:
  - MultipleScatteringUpdator OpenMP implementation
- Open Multi Processing:
  - C,C++,Fortran
  - Simple: annotation for parallel portions with pragmas
  - Good potential when coupled to recent compilers
  - No GPU support

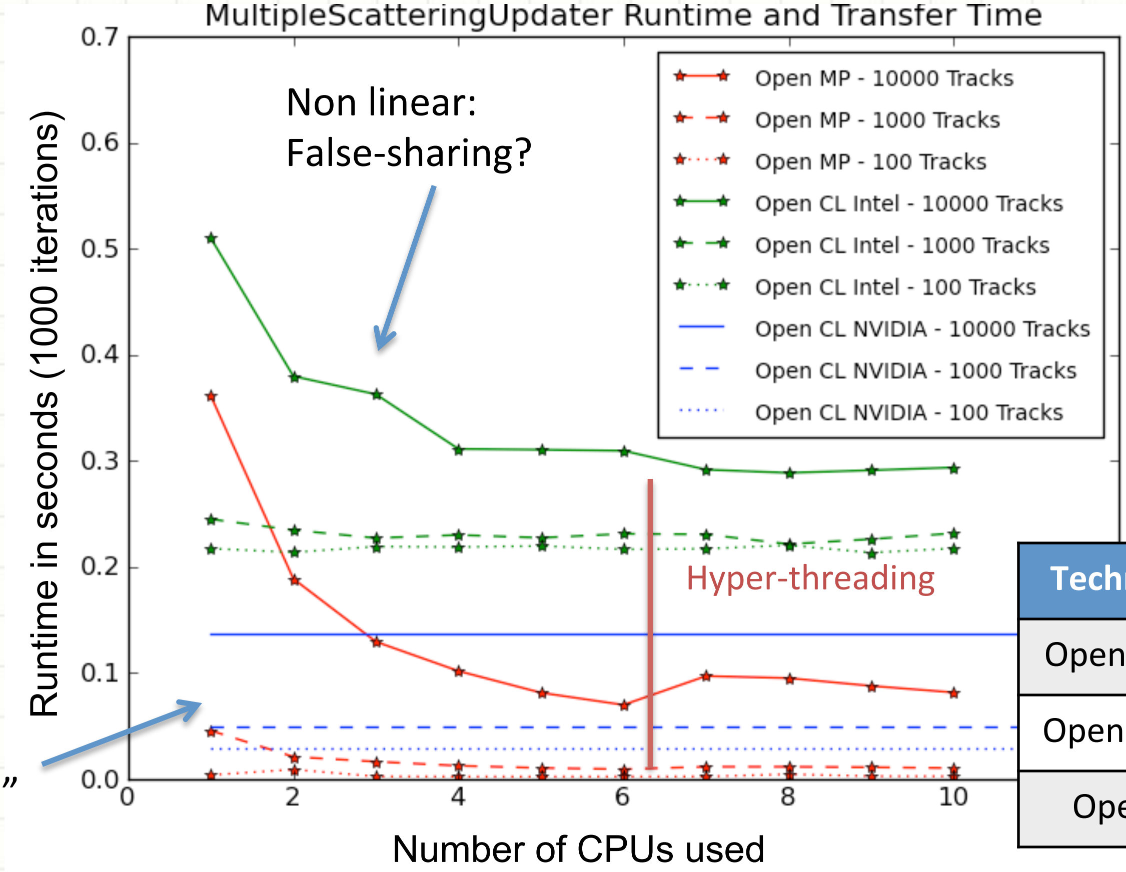


```
#pragma omp parallel for
for (i = 0; i < N; i++)
    a[i] = 2 * i;
```

Rely on a well established technology to assess the performance of OpenCL

Intel Box: Overall Performance 1

Times cited are intended for 6 cores, except for the GPU



Non linear: False-sharing?

Hyper-threading

GPU considered as a “single CPU”

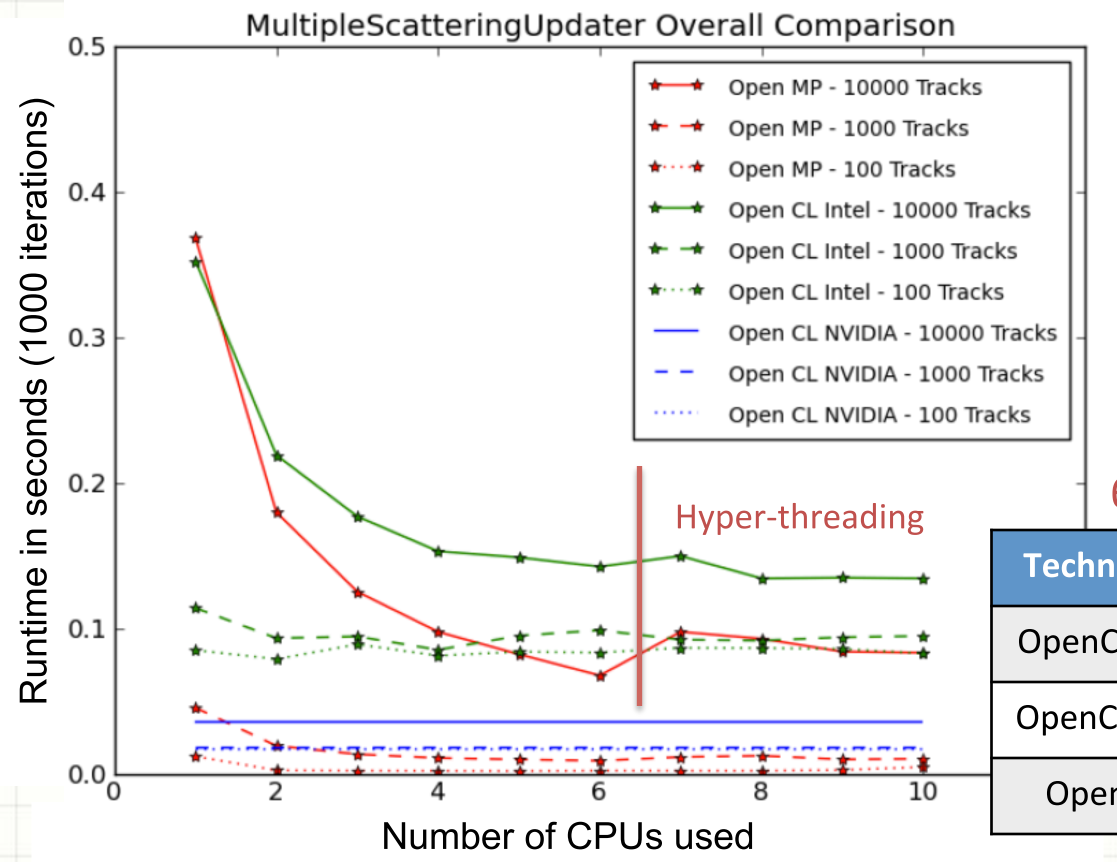
6 cores used

Technology	Time ms
OpenCL CPU	310
OpenCL GPU	130
OpenMP	70

- 10k,1k and 100 tracks considered – 1000 reiterations
  - 100 warm-up iterations: not accounted in the total time
- Allocation done once
- Transfers: 4 doubles sent to the device and 3 copied back
- Worst case scenario: a huge number of copies is done!

Intel Box: no transfer

Times cited are intended for 6 cores, except for the GPU

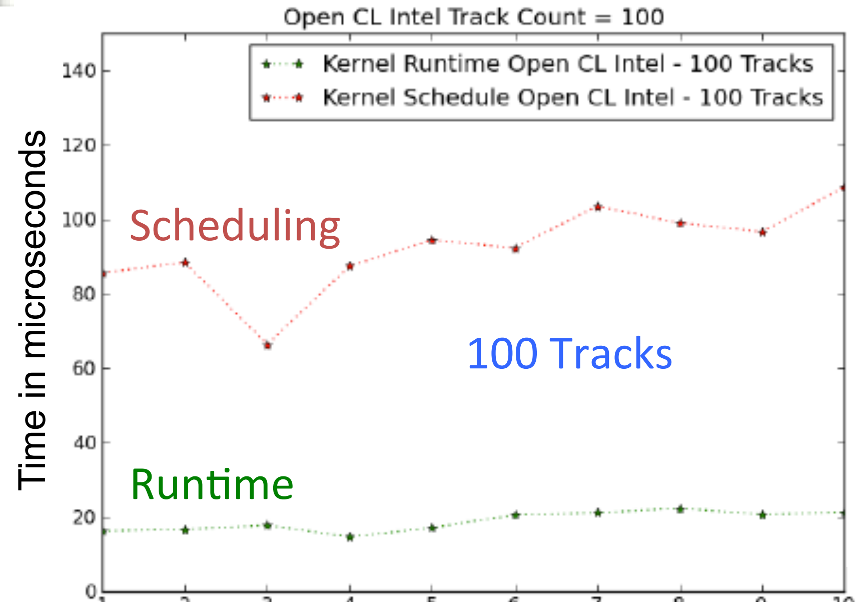


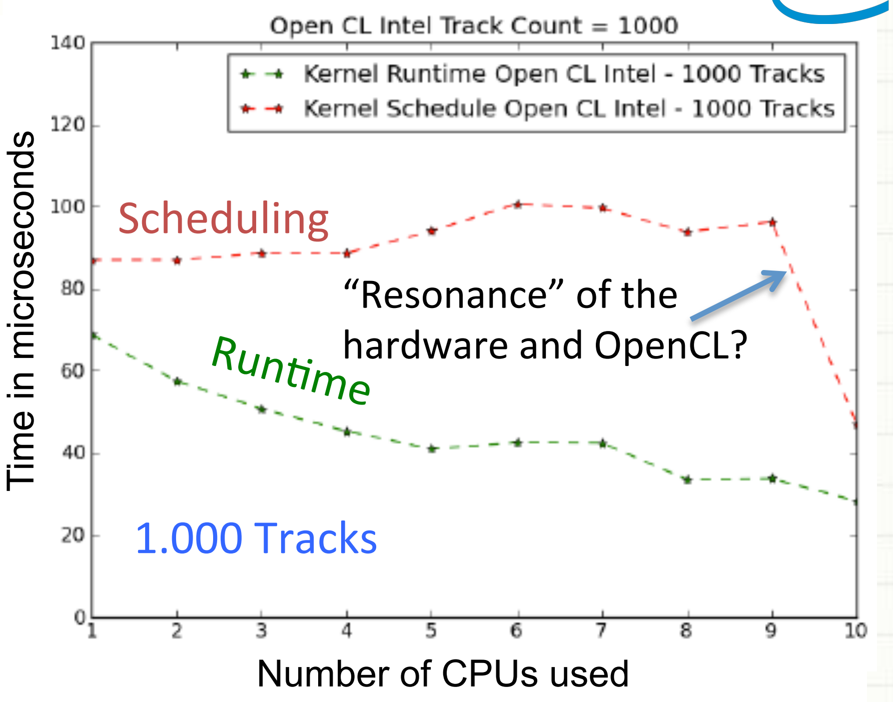
6 cores used

Technology	Time ms
OpenCL CPU	140 (310)
OpenCL GPU	**40** (130)
OpenMP	70

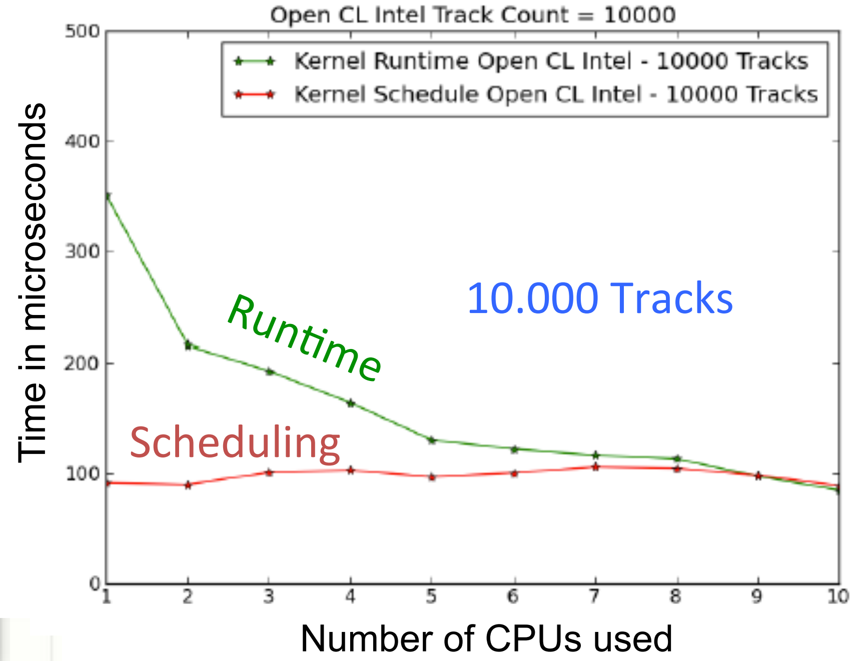
- Same conditions as in the previous slide
  - Transfers to/from memory not accounted
- 6 cores case CPU OpenCL: GPU OpenCL 3.8x – OpenMP 2x faster
- Within OpenCL, same hierarchy kept
- Transferring data from/to the device has an influence and this effect must be carefully considered

Intel Box: CPU Scheduling Overhead



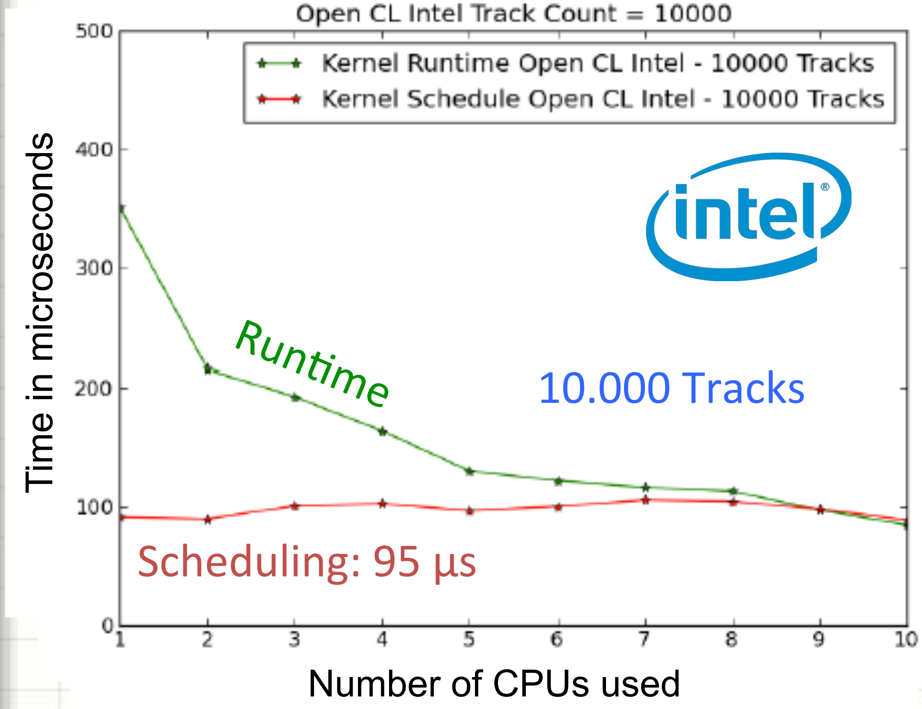


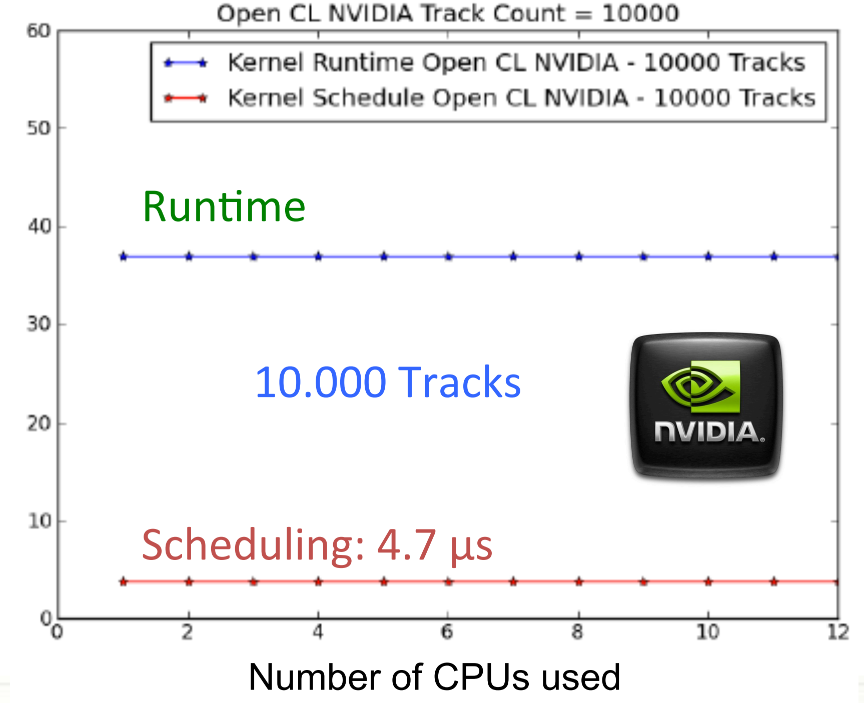
“Resonance” of the hardware and OpenCL?



- Scheduling is ~constant
- Small workload: Scheduling overhead > actual runtime
- Indication that in a real-life situation going across event boundaries might be needed

Intel Box: GPU Scheduling Overhead

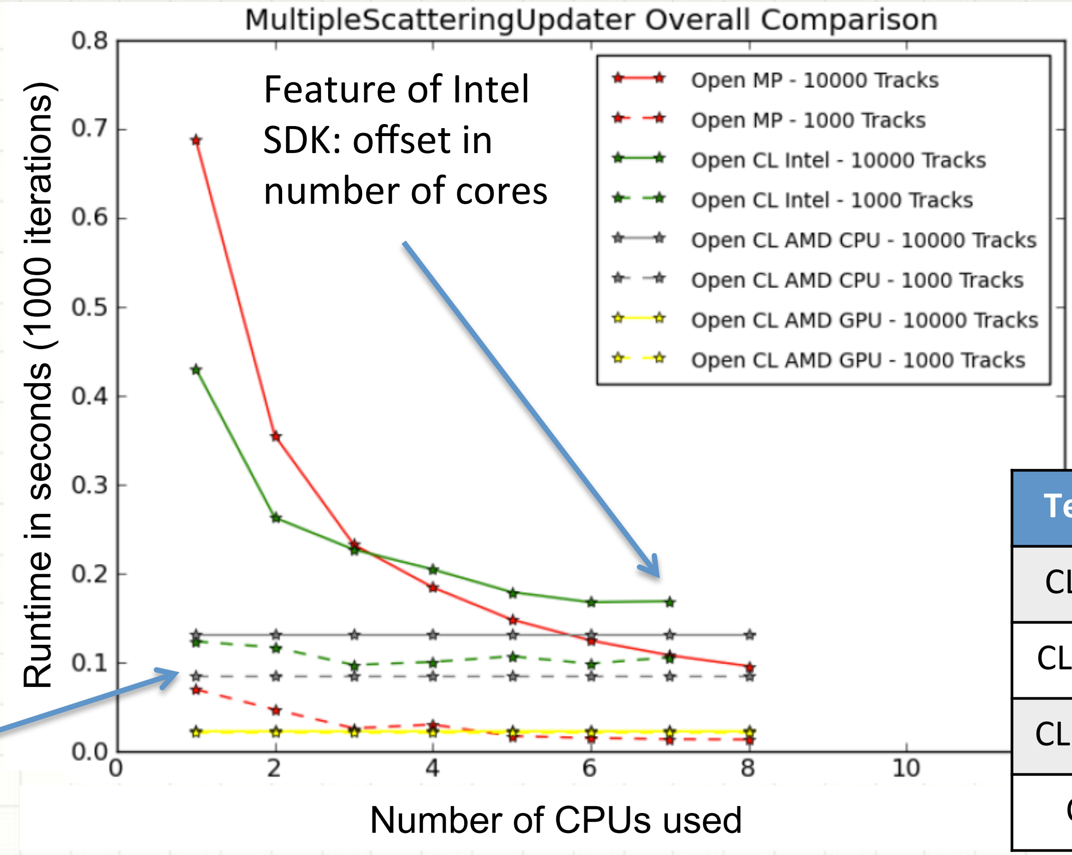




- GPU scheduling is 5% of the GPU one
  - 5 VS 95 µs

AMD Box

Times cited are intended for 6 cores, except for the GPU and the AMD SDK (8 cores)



Feature of Intel SDK: offset in number of cores

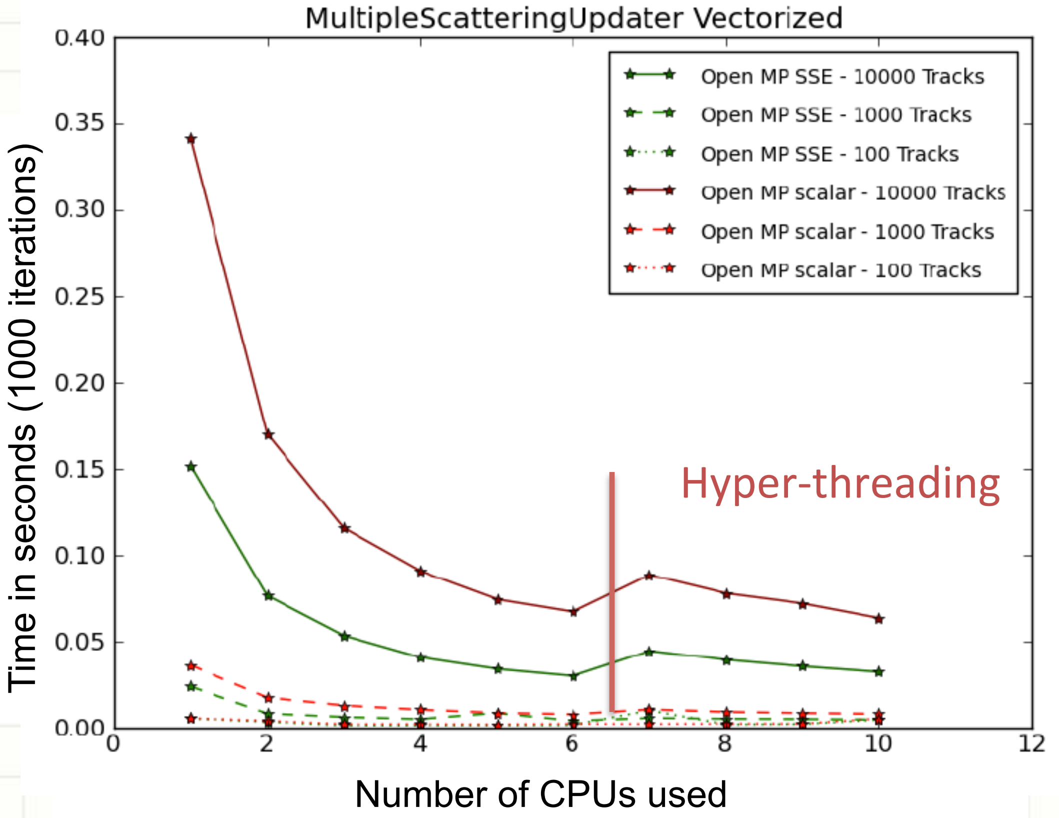
AMD SDK: no limitation on cores used possible

6 or 8 cores used

Technology	Time ms
CL IntelCPU	160
CL AMD CPU	140
CL AMD GPU	**25**
OpenMP	90

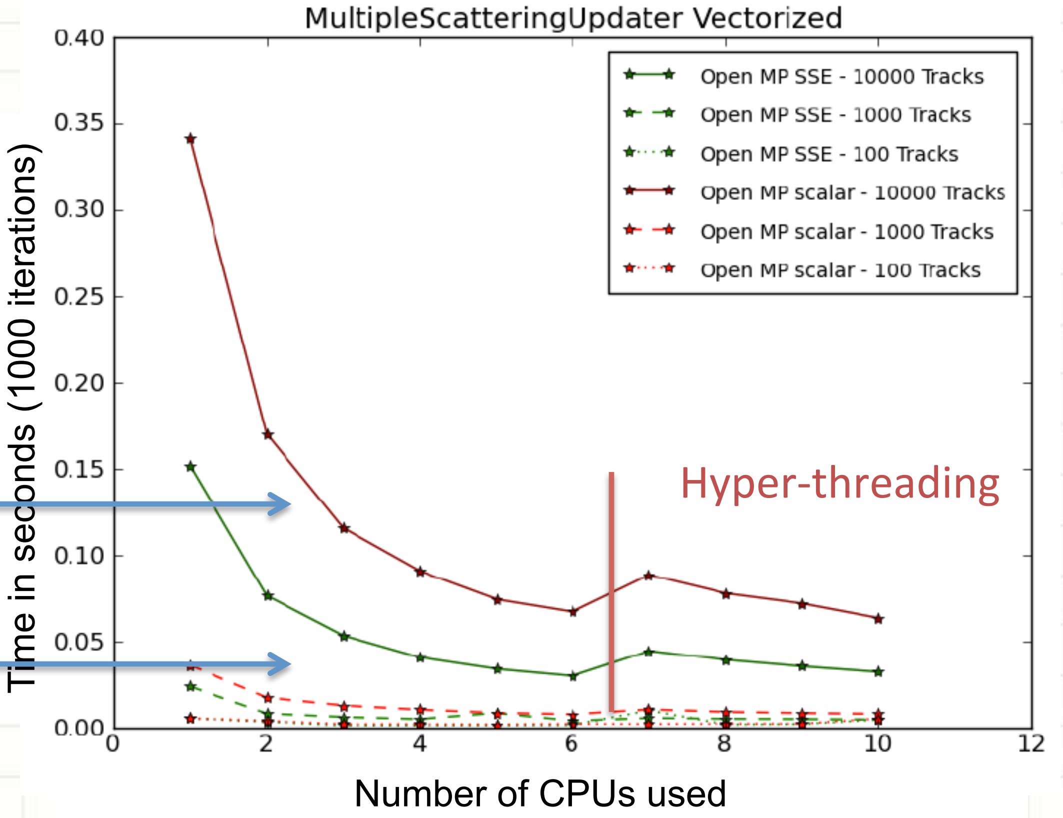
- 10k and 1000 tracks cases only (warm-up loop always present)
- No transfer from/to memory considered
- Comparison of CPU (Intel+AMD SDK), GPU and OpenMP
- OpenMP always slightly faster
- OpenCL CPU: AMD SDK faster but less flexible (all 8 cores used)
- AMD GPU (25 ms) faster than all CPUs and NVIDIA (40 ms)

OpenMP + Vectorisation



- GCC 4.7 + OpenMP: well established technology
- Autovectorisation enabled (SSE2)
  - CMS autovectorisable logarithm implementation used (faster per se than libm)
- An overall factor 2 in speed wrt scalar version
- Faster than all OpenCL CPU implementations
- OpenCL on GPU (both AMD and NVIDIA) is still faster

OpenMP + Vectorisation



OpenCL CPU (6)

OpenCL GPU

Hyper-threading

- GCC 4.7 + OpenMP: well established technology
- Autovectorisation enabled (SSE2)
  - CMS autovectorisable logarithm implementation used (faster per se than libm)
- An overall factor 2 in speed wrt scalar version
- Faster than all OpenCL CPU implementations
- OpenCL on GPU (both AMD and NVIDIA) is still faster

Usability

Developing with OpenCL

- Many insights gained from the applications programmer's point of view
- Some features of programming with OpenCL are for some use cases less than optimal
  - Especially in the context of a sw project maintained by a plethora of users with non-uniform computer skills
- Examples:
  - Kernel code passed as string
  - No syntax check for kernels at compile time
  - Explicit memory management: alloc, dealloc

OpenCL is a powerful tool but not easy to use "as is"

A possible simplification

openclam
OpenCL C++ wrapper

Start from openclam:

- Kernels defined and interleaved with “regular” C++ code
- Compile-time syntax and type checking

  → No surprises during kernels re-compilation at runtime

On the top of that, a convenient C++ layer was built:

- Huge increase in usability:
  - Simplify kernel creation with various numbers of parameters (all OpenCL supported types)
  - Convenient data structures wrappers for vectors and matrices
  - Automated memory management

See references and backup for more details

Conclusions

Conclusions

More details in the supporting document in the indico agenda!

Performance

- A "real-life" standard candle was used to test AMD, Intel and Nvidia hardware and OpenCL runtimes.
- OpenMP and OpenCL performance is comparable
  - OpenMP still faster when considering CPUs
- Absolute performances: AMD GPU > Nvidia GPU > AMD CPU* ˜ Intel CPU*

  * Normalised according to the number of cores
- The scheduling overhead must be seriously considered
  - To ammortise it, elements from subsequent events might need to be lumped together
  - NVIDIA GPU scheduling overhead was 5% of the Intel CPU one
- The data transfer overhead was not a significant penalty for the GPU

Portability

- The promises of OpenCL are maintained: the same kernels run smoothly and without any modifications to the source code on [CG]PUs. No-vendor lock-in!

Usability

- The bare OpenCL API might result cumbersome
- An appropriate wrapper was developed on top of openclam
  - This is a promising strategy to take advantage of the power of OpenCL

Thanks to Benedikt Hegner
for the fruitful discussions!

Next Steps

Up to now: real-life simple algorithm in an artificial environment

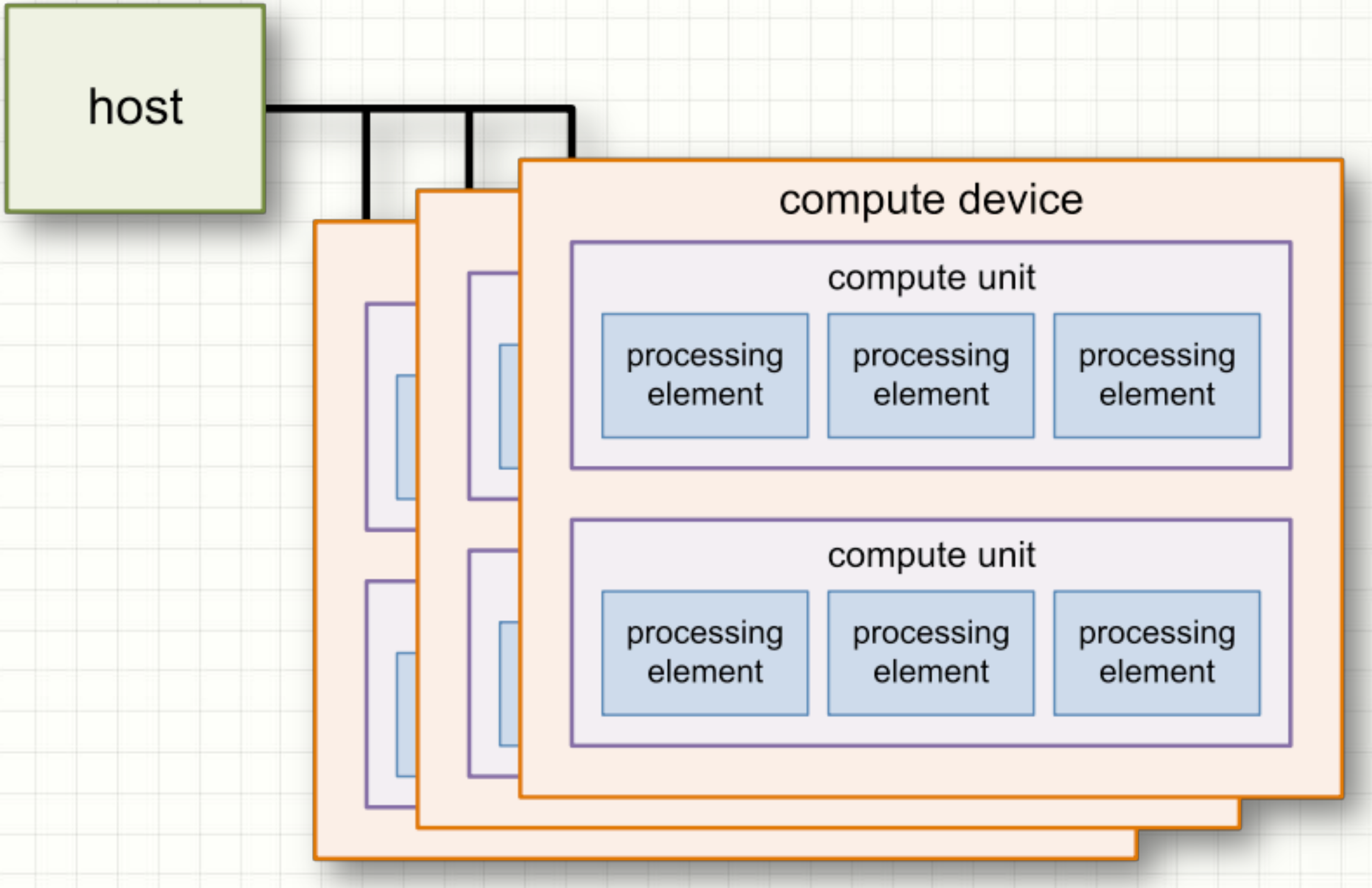
- Consider a more complex algorithm: Deterministic Annealing for vertices reconstruction (DAClusterizerInZ)
- Make it callable from a CMSSW component
- Answer these questions:
  - How much "work" shall we extract from a typical data processing chain to profit from a CPU->GPU offload?
  - How does this scale with one to N processes on a K cores machine?
  - Which should be the total cost of ownership of a (more) graphics card(s) per blade at a trigger/Tier0,1,2 farm for this idea to be profitable?
  - Get an idea about how the situation can change with another kind of accelerator (e.g. MIC)?

References:

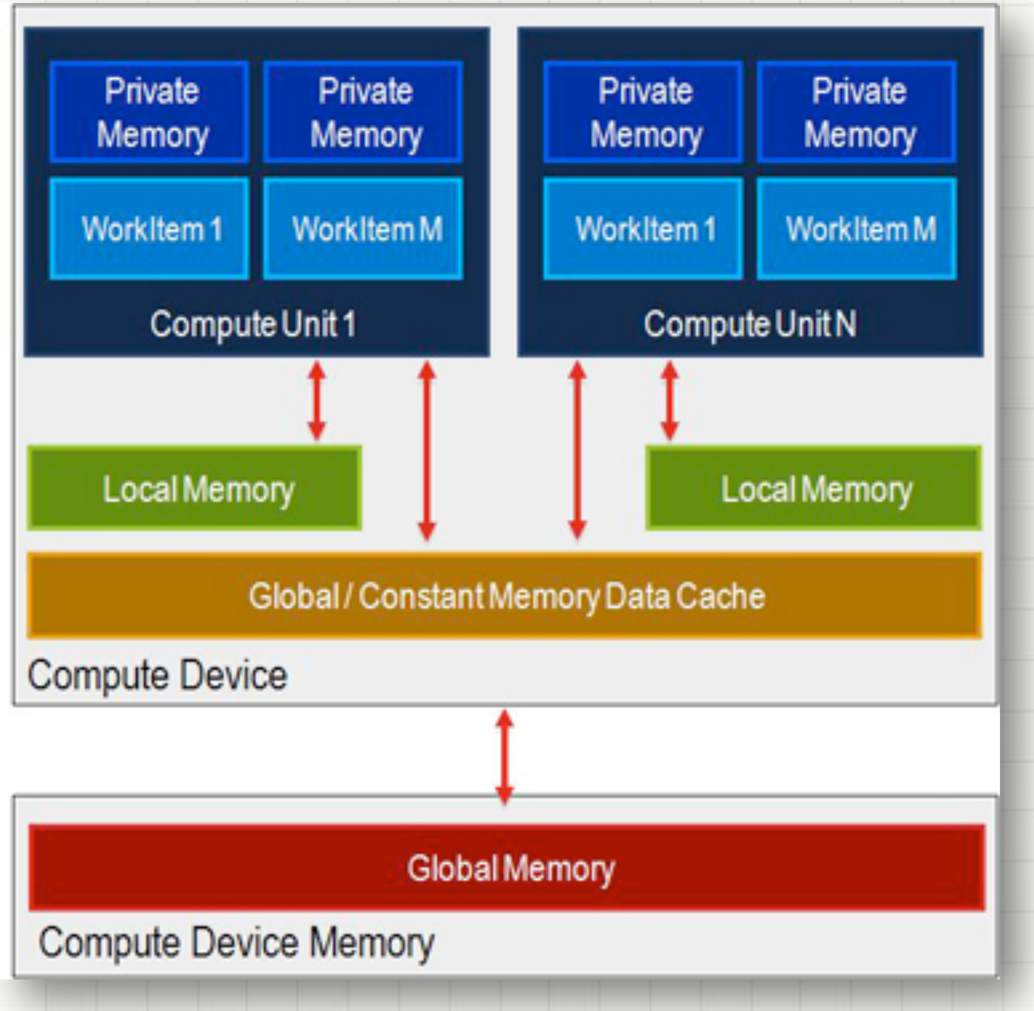
- http://www.khronos.org/opencl/
- Passage of particles through matter: http://pdg.lbl.gov/2011/reviews/rpp2011-rev-passage-particles-matter.pdf
- Multiple scattering updator in CMSSW: http://cmslxr.fnal.gov/lxr/source/TrackingTools/MaterialEffects/src/MultipleScatteringUpdator.cc?v=CMSSW_5_2_0#014
- Intel OpenCL SDK: http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/
- AMD OpenCL SDK: http://developer.amd.com/pages/default.aspx
- LLVM: http://llvm.org/
- Openclam: http://code.google.com/p/openclam/

Backup

OpenCL abstract resource layout



OpenCL Memory Model

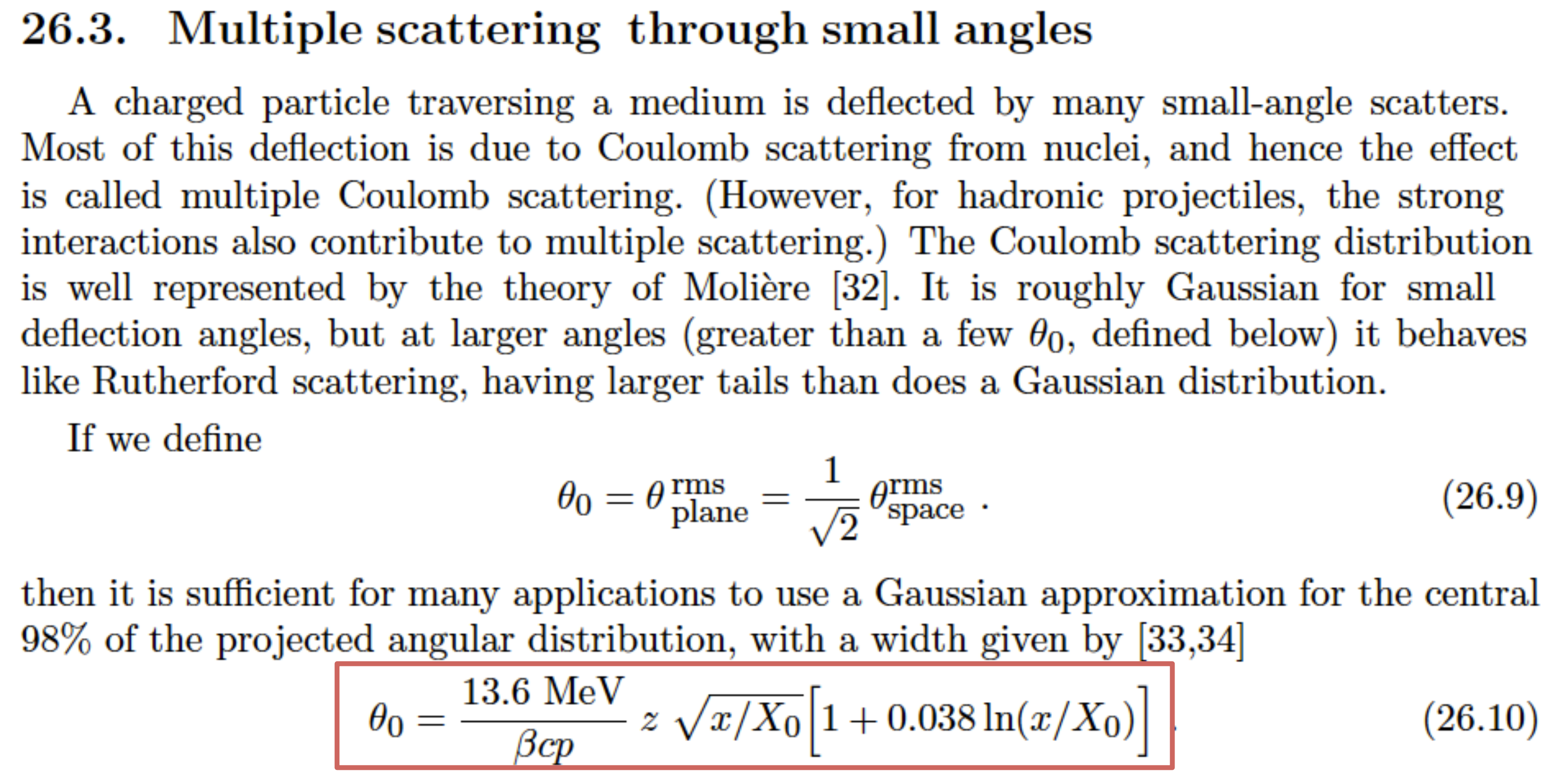


OpenCL C99

C-based language kernels:

- Derived from ISO C99
- Few restrictions, e.g. recursion, function pointers
- Short vector types e.g., float4, short2, int16
- Built-in functions: math (e.g., sin), geometric, common (e.g., min, clamp)

Passage of particles through matter



26.3. Multiple scattering through small angles

A charged particle traversing a medium is deflected by many small-angle scatters. Most of this deflection is due to Coulomb scattering from nuclei, and hence the effect is called multiple Coulomb scattering. (However, for hadronic projectiles, the strong interactions also contribute to multiple scattering.) The Coulomb scattering distribution is well represented by the theory of Molière [32]. It is roughly Gaussian for small deflection angles, but at larger angles (greater than a few θ_0, defined below) it behaves like Rutherford scattering, having larger tails than does a Gaussian distribution.

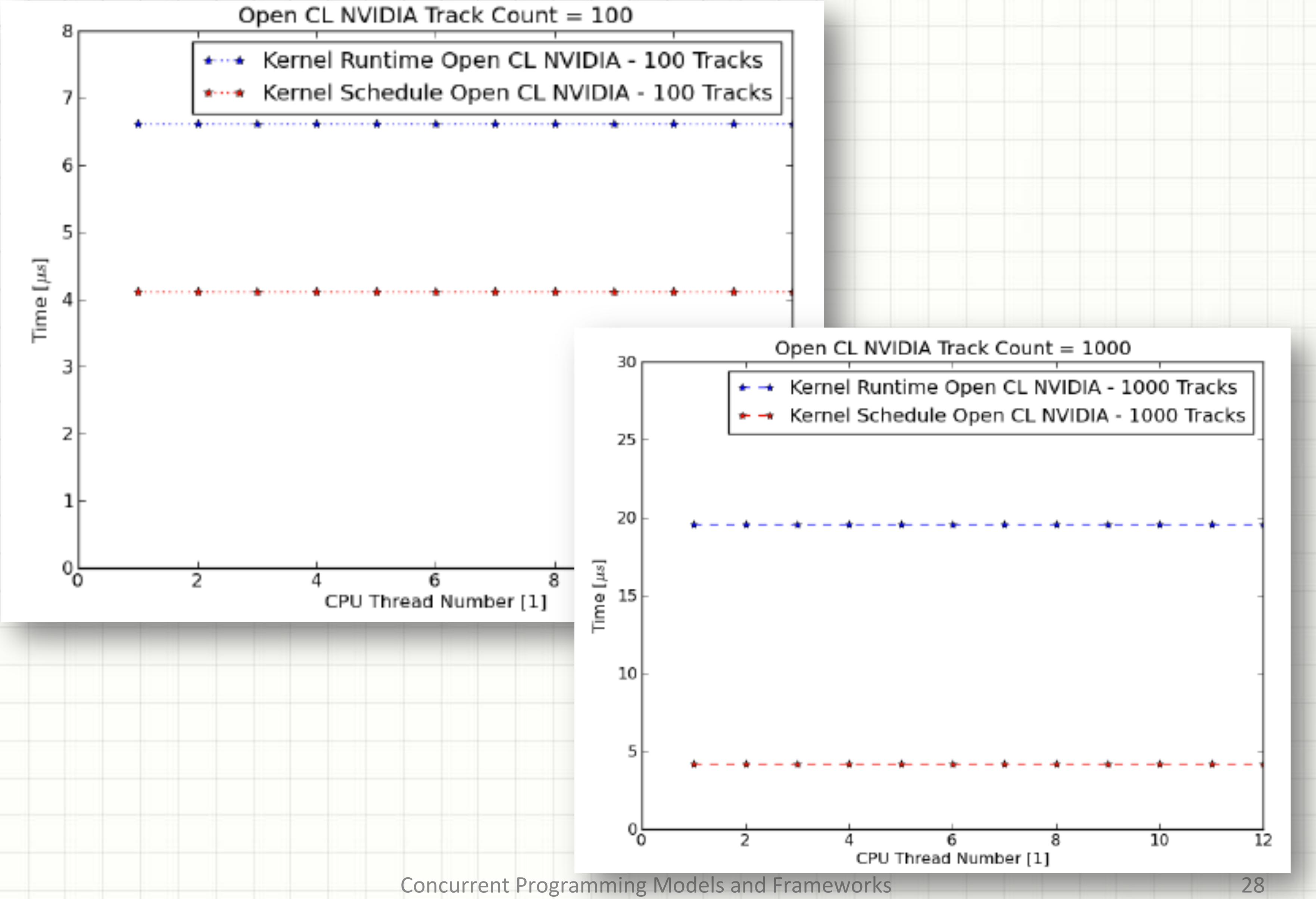
If we define

$$\theta_0 = \theta_{\text{plane}}^{\text{rms}} = \frac{1}{\sqrt{2}}\,\theta_{\text{space}}^{\text{rms}} \; . \qquad (26.9)$$

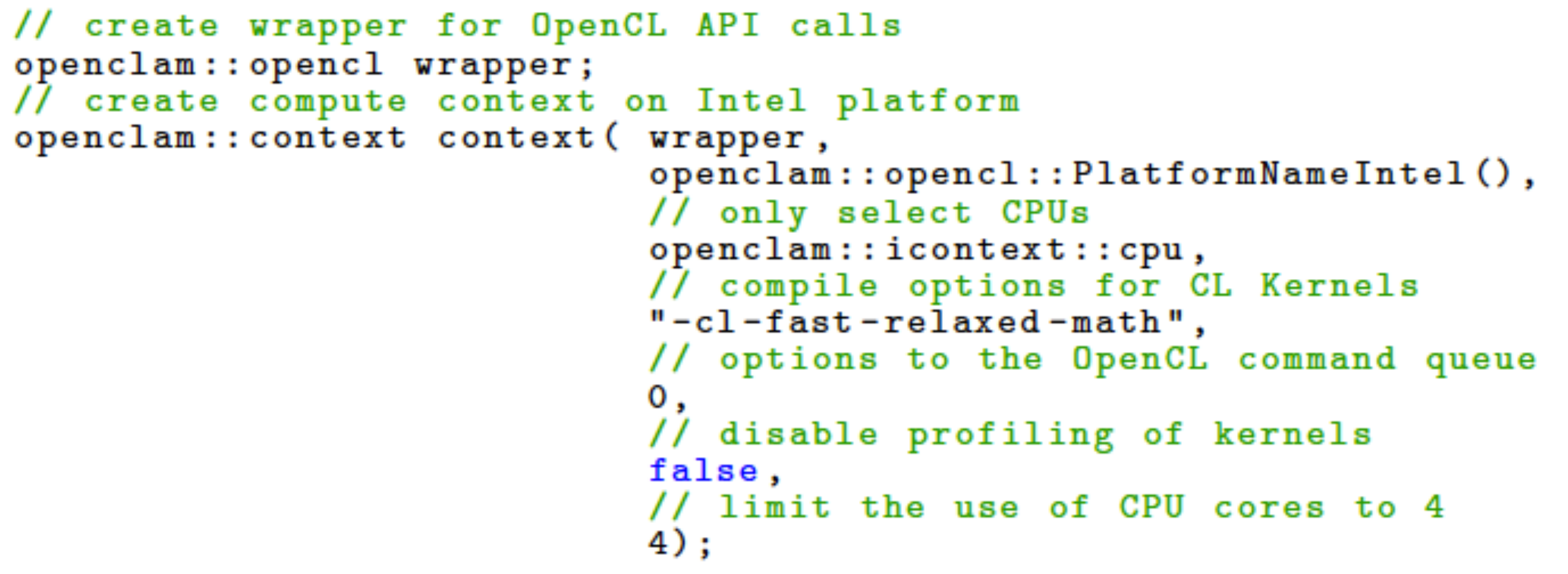
then it is sufficient for many applications to use a Gaussian approximation for the central 98% of the projected angular distribution, with a width given by [33,34]

$$\boxed{\theta_0 = \frac{13.6\text{ MeV}}{\beta c p}\, z\, \sqrt{x/X_0}\Big[1 + 0.038\ln(x/X_0)\Big]} \; . \qquad (26.10)$$

More Scheduling Time

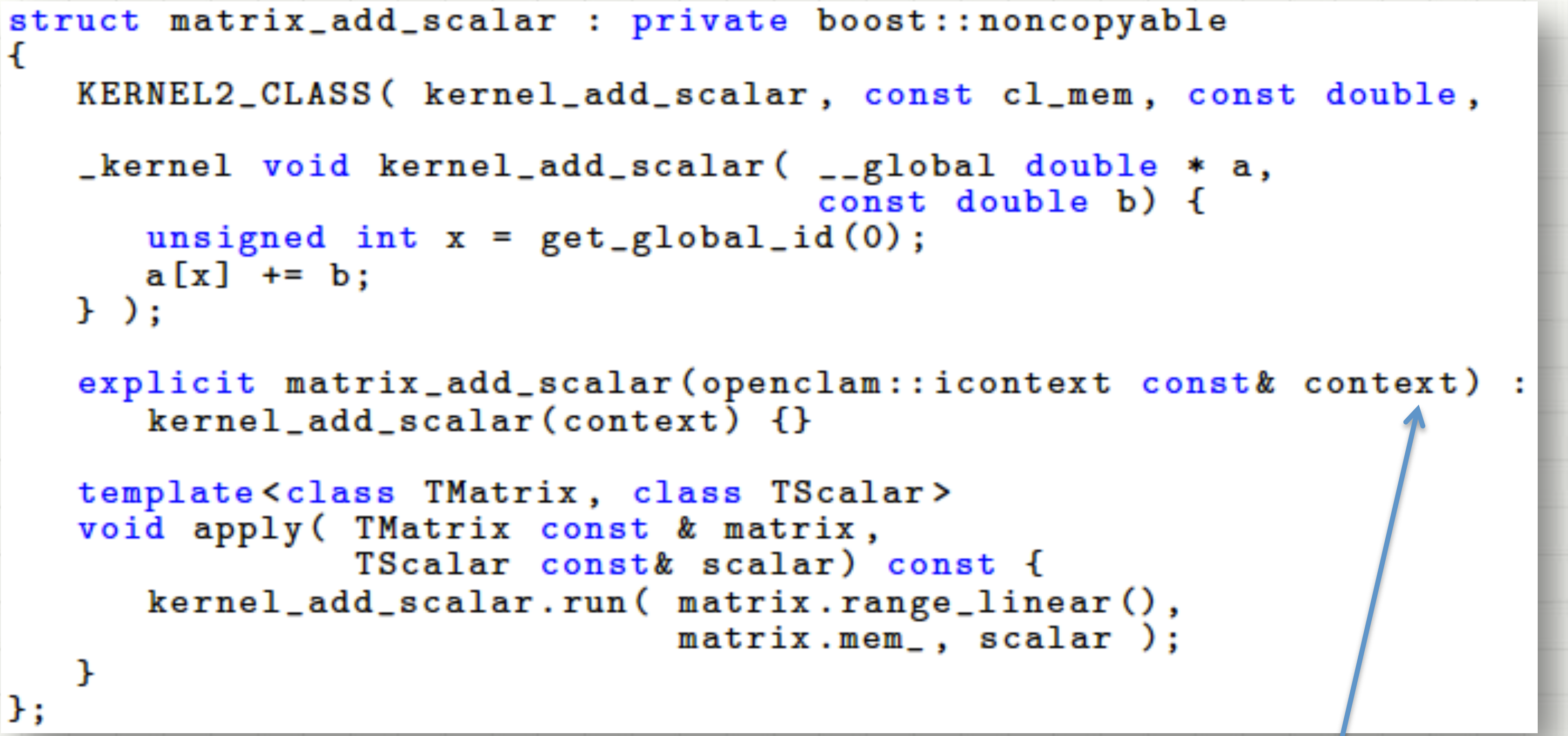


A “bare openclam” example



```
// 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);
```

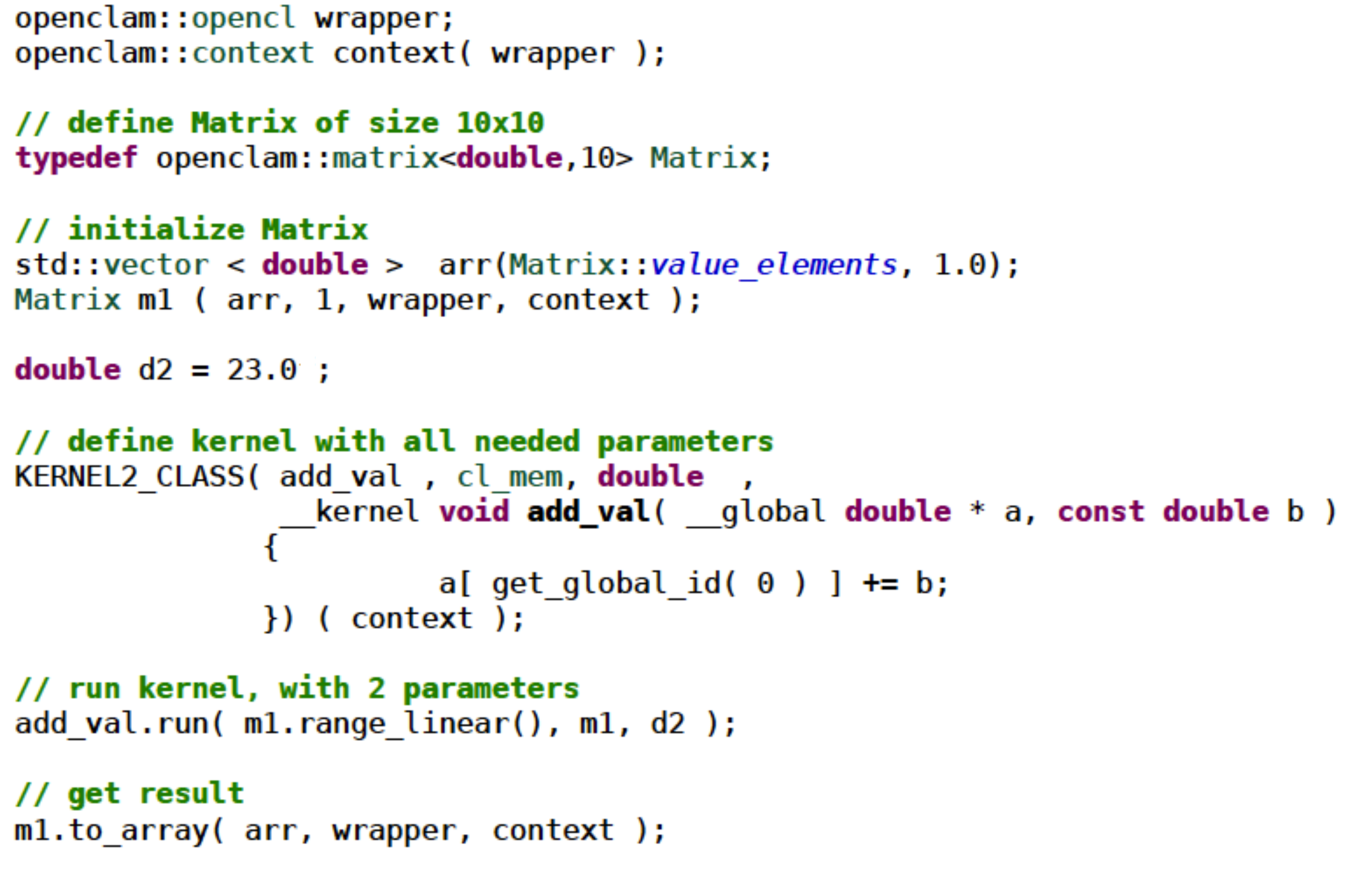
Full kernel code example



```
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 );
   }
};
```

Carries the information about the resources to be used and compiler flags.

Another Full Example



```
openclam::opencl wrapper;
openclam::context context( wrapper );

// define Matrix of size 10x10
typedef openclam::matrix<double,10> Matrix;

// initialize Matrix
std::vector < double >  arr(Matrix::value_elements, 1.0);
Matrix m1 ( arr, 1, wrapper, context );

double d2 = 23.0 ;

// define kernel with all needed parameters
KERNEL2_CLASS( add_val , cl_mem, double  ,
               __kernel void add_val( __global double * a, const double b )
              {
                        a[ get_global_id( 0 ) ] += b;
              }) ( context );

// run kernel, with 2 parameters
add_val.run( m1.range_linear(), m1, d2 );

// get result
m1.to_array( arr, wrapper, context );
```

Notes

Category