OpenCL: Portable programming at the right or the wrong level?

Yngve Sneen Lindal
European Organization for Nuclear Research (CERN), Geneva, Switzerland

Second International Workshop for Future Challenges in Tracking and Trigger Concepts,
CERN
July 7th–8th, 2011

OpenCL

- Standard for heterogeneous computing, set by the Khronos Group

..and many more

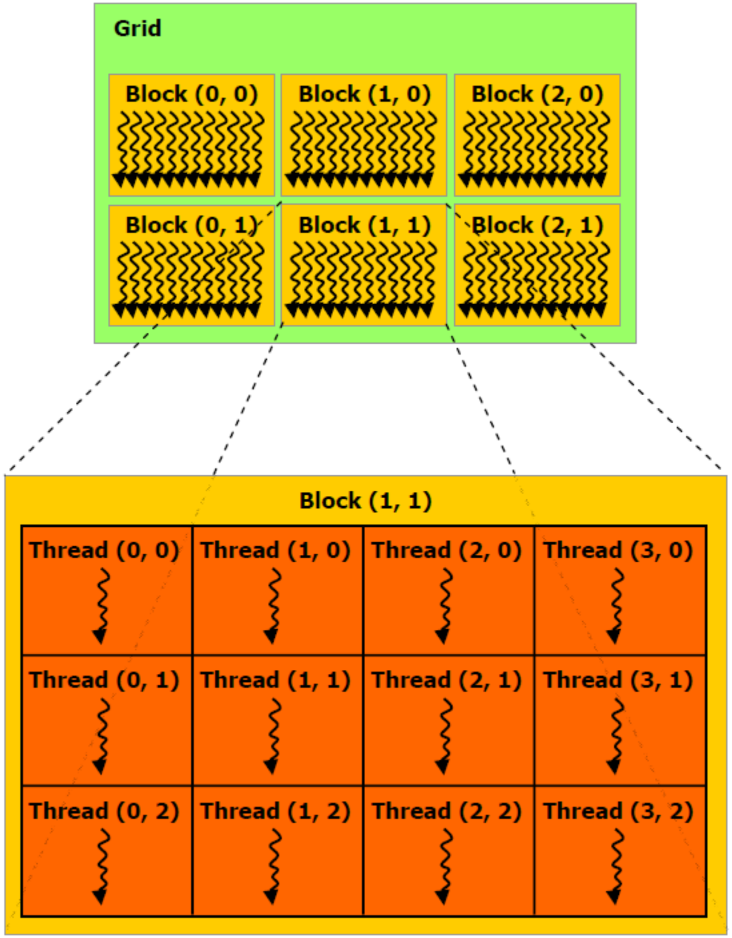
OpenCL

- Idea: implicit data-parallel code executed in «kernels», portable across different devices/vendors

```
void evaluatePdfGaussian(const double mu, const double sigma, const double* data,
  double* results, const int N)
{
  #pragma omp parallel for
  for(int i = 0; i < N; i++)
  {
    double temp = (data[i]-mu)/sigma;
    temp *= temp;
    results[i] = exp(-0.5*temp);
  }
}
```

```
__kernel void evaluatePdfGaussian(__const double mu, __const double sigma, __global
 const double *data, __global double *results, __const int N)
{
  int i = get_global_id(0);
  if (i >= N) return;
  double x = data[i];
  double temp = (x-mu)/sigma;
  temp *= temp;
  results[i] = exp(-0.5*temp);
}
```

- A kernel represents a parallel execution on a grid of threads



(Illustration borrowed from NVIDIAs OpenCL programming guide)

http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf

OpenCL

- Goal: *To use this both for CPUs and GPUs with the same kernel code, and that this is performant*
- Paradigm suitable for GPU execution
- CPUs and GPUs differ largely in hardware implementation
- Strictly C (or a superset of), no C++ here
- Cannot call «host code» from OpenCL code, and vice versa
- A lot of compute intensive programs are written in C++
- Will this work (and be performant) on CPUs as well?

OpenCL device abstractions

- Different hardware/SDKs/drivers are represented by different «platform» objects
- A platform object can have a range of devices (you must have them physically, of course)

An example

```
cl_platform platform;
cl_device device;
cl_context context;
cl_command_queue queue;
cl_int status;

clGetPlatformIDs(1, &platform, NULL);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);
queue = clCreateCommandQueue(context, device, 0, &status);
```

Kernel declaration/execution

❑ The Gaussian kernel, revisited

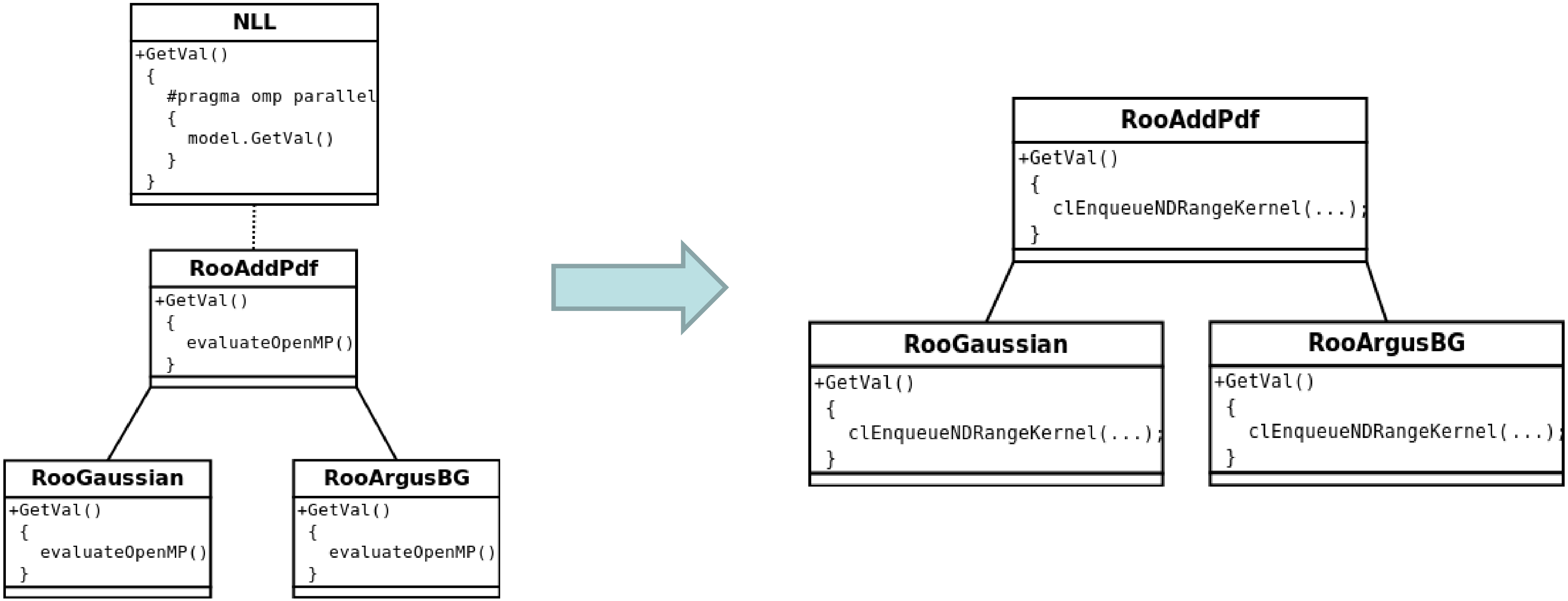
```
__kernel void evaluatePdfGaussian(__const double mu, __const double sigma, __global const double *data,
__global double *results, __const int N)
{
   int i = get_global_id(0);
   if (i >= N) return;
   double x = data[i];
   double temp = (x-mu)/sigma;
   temp *= temp;
   results[i] = exp(-0.5*temp);
}
```

❑ Executing a computational kernel

```
//Assume we have the required arguments and a kernel object for the Gaussian kernel above
clSetKernelArg(evaluatePdfGaussian, 0, sizeof(float), (void*)&mu);
clSetKernelArg(evaluatePdfGaussian, 1, sizeof(float), (void*)&sigma);
clSetKernelArg(evaluatePdfGaussian, 2, sizeof(cl_mem), (void*)&data);
clSetKernelArg(evaluatePdfGaussian, 3, sizeof(cl_mem), (void*)&results);
clSetKernelArg(evaluatePdfGaussian, 4, sizeof(int), (void*)&N);
size_t workGroupSize = 128;  //e.g.
size_t numWorkGroups = N % workGroupSize == 0 ? N/workGroupSize : N/workGroupSize + 1;
size_t total = workGroupSize * numWorkGroups;
clEnqueueNDRangeKernel(queue, evaluatePdfGaussian, 1, NULL, &total, &workGroupSize, 0, NULL, NULL);
```

An implementation example (RooFit)

- With OpenMP, each thread can evaluate a tree of PDFs top-down directly in fully parallel. Using OpenCL requires an explicit call to a kernel inside each PDF (see 2nd illustration), suggesting lower parallel efficiency.



- Leads to larger serial fraction, many kernel calls and in general, stalls
- Remember, no C++ in OpenCL kernels

Experiences

- Introduces more expressive code when setting up environment and e.g. calling kernels.
- Using plain C++ for CPU and OpenCL for GPU, we get duplication of code since we now must use an OpenCL compiler in addition to the C/C++ compiler
- Neither Intel or AMDs x86 implementation (Linux) offers auto-vectorization per 01.07.2011
- Have to use vector types to achieve vectorization. But even then AMDs OpenCL compiler (for CPU) does not vectorize transcendentals for instance

Manual vectorization ☹

```
__kernel void evaluatePdfGaussian(__const double mu, __const double sigma, __global
  const double *data, __global double *results, __const int N)
{
  int i = get_global_id(0);
  if (i >= N) return;
  double x = data[i];
  double temp = (x-mu)/sigma;
  temp *= temp;
  results[i] = exp(-0.5*temp);
}
```

```
__kernel __attribute__((vec_type_hint(double2))) void evaluatePdfGaussian(__const double mu,
    __const double sigma, __global const double *data, __global double *results, __const int N
    )
{
  int i = get_global_id(0);
  if (i >= N/2) return;
  double2 x = vload2(i, data);
  double2 temp = (x-mu)/sigma;
  temp *= temp;
  double2 result = exp(-0.5*temp);
  vstore2(result, i, results);
}
```

Experiences cont.

- Hidden threading overhead. Necessary to do more work per OpenCL thread for performance (goes for both Intel and AMD)
- Have talked to Intel OpenCL expert. He says that Intel will support auto-vectorization in OpenCL
- It would of course be nice to have one piece of code for any device, but that seems like somewhat of a silver bullet so far…
- AMD APP SDK uses LLVM as backend for CPUs

Manual work partitioning ☹☹☹

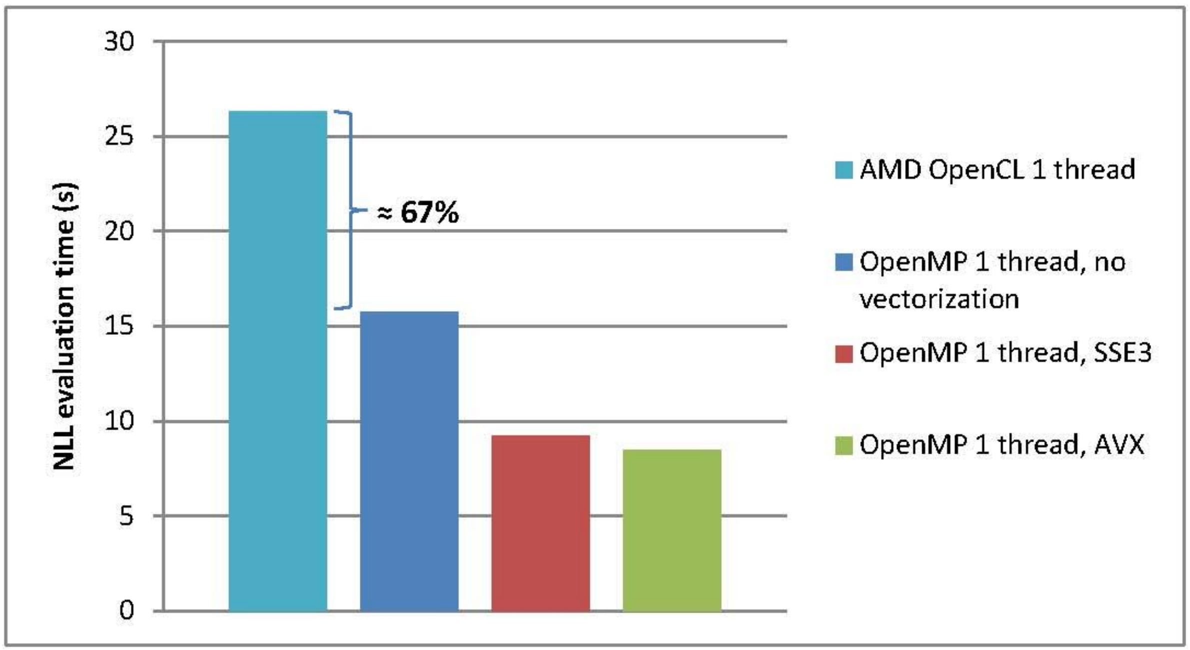
```
__kernel void evaluatePdfGaussian(__const double mu, __const double sigma, __global
 const double *data, __global double *results, __const int N)
{
  int i = get_global_id(0);
  if (i >= N) return;
  double x = data[i];
  double temp = (x-mu)/sigma;
  temp *= temp;
  results[i] = exp(-0.5*temp);
}
```

⇓

```
__kernel __attribute__((vec_type_hint(double2))) void evaluatePdfGaussian(__const double mu,
    __const double sigma, __global const double *data, __global double *results, __const int N,
     __const int numComputeElements)
{
  int i = get_global_id(0);
  if (i >= N) return;
  int part = N/numComputeElements;
  for(int index = i*part; index < (i+1)*part - 1; index+=2)
  {
    double2 x = vload2(index/2, data);
    double2 temp = (x-mu)/sigma;
    temp *= temp;
    double2 result = exp(-0.5*temp);
    vstore2(result, index/2, results);
  }
}
```

Some CPU results

- Benchmark on a desktop system
  - CPU: Intel Sandy Bridge @ 3.40GHz: 4 cores – 8 potential hardware threads
  - Linux 64bit, Intel C++ compiler version 12.1



/Function /Call Stack	CPU Time
__exp_f64	40.4%
__OpenCL_evaluatePdfAdd_stub	12.3%
__OpenCL_normalizeResults_stub	12.2%
__OpenCL_evaluatePdfProd_stub	8.9%
__OpenCL_evaluatePdfPolynomial_stub	7.1%
__log_f64	4.8%
__OpenCL_evaluatePdfArgusBG_stub	3.8%
[libatiocl64.so]	3.0%

VS

/Function /Call Stack	CPU Time
__svml_exp2.N	47.2%
PdfArgusBG::evaluateOpenMP	8.1%
PdfPolynomial::evaluateOpenMP	6.9%
PdfProd::evaluateOpenMP	6.5%
PdfAdd::evaluateOpenMP	6.1%
PdfGaussian::evaluateOpenMP	5.6%
AbsPdf::GetVal	4.7%
__svml_log2.L	3.8%
NLL::GetVal	3.4%
PdfBifurGaussian::evaluateOpenMP	1.9%

On GPUs

- Potential portability problem between NVIDIA and AMD/ATI; VLIW registers
- More difficult for AMD to exploit parallelism
- AMD Radeon series has 4 general stream cores and 1 special functional unit per scalar processor. We cannot use the functional unit (Geforce also has special functional units)
- We use transcendentals and double precision. Peak performance? Dream on...
- So, portability issue will in general arise only if doing simple math and not being memory-bound (typically, linear algebra)
- Of course, optimal work group size will differ between different models
- In our case, we are in general memory (latency) bound, so we don't experience any difference

Conclusion

- Reflect carefully before introducing OpenCL in your code
- Not ideal for CPU computations until code can be written the same way on the CPU as on the GPU and be performant. In essence this means:
  - Automatic vectorization for CPUs (both Intel and AMD supports SSE…)
  - Implicit effective thread-scheduling for most workloads
- No point in mixing OpenCL for CPUs and GPUs today, from a programmer's perspective (me). Atleast if you can play around with the Intel compiler

Conclusion

- OpenCL can be painful in legacy C++ programs. NVIDIA CUDA supports C++, but then we're bound to one specific vendor
- The main positive effect is code reuse between CPU and GPU
- Yes, it is portable, but it is not fully performance portable (there's a bunch of papers that states exactly this, also across GPU vendors)
- We are now focusing on hybrid (balancing) solutions with OpenMP and OpenCL, and they can co-exist fairly well