OpenCL programming

Piero Lanucara, SCAI

16 April 2014
Heterogeneous High Performance Programming framework

- [http://www.hpcwire.com/hpcwire/2012-02-28/opencl_gains_ground_on_cuda.html](http://www.hpcwire.com/hpcwire/2012-02-28/opencl_gains_ground_on_cuda.html)

"As the two major programming frameworks for GPU computing, OpenCL and CUDA have been competing for mindshare in the developer community for the past few years. Until recently, CUDA has attracted most of the attention from developers, especially in the high performance computing realm. But OpenCL software has now matured to the point where HPC practitioners are taking a second look.

Both OpenCL and CUDA provide a general-purpose model for data parallelism as well as low-level access to hardware, but only OpenCL provides an open, industry-standard framework. As such, it has garnered support from nearly all processor manufacturers including AMD, Intel, and NVIDIA, as well as others that serve the mobile and embedded computing markets. As a result, applications developed in OpenCL are now portable across a variety of GPUs and CPUs."

Heterogeneous High Performance Programming framework (2)

A modern computing platform includes:
• One or more CPUs
• One of more GPUs
• DSP processors
• Accelerators
• … other?

E.g. Samsung® Exynos 5:
• Dual core ARM A15
  1.7GHz, Mali T604 GPU

OpenCL lets Programmers write a single portable program that uses ALL resources in the heterogeneous platform
Microprocessor trends

Individual processors have many (possibly heterogeneous) cores.

The Heterogeneous many-core challenge:
How are we to build a software ecosystem for the Heterogeneous many core platform?
Industry Standards for Programming Heterogeneous Platforms

OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

CPUs
Multiple cores driving performance increases

Emerging Intersection

GPUs
Increasingly general purpose data-parallel computing

Multi-processor programming – e.g. OpenMP

Graphics APIs and Shading Languages

OpenCL
Heterogeneous Computing
OpenCL Timeline

- Launched Jun’08 … 6 months from “strawman” to OpenCL 1.0
- Rapid innovation to match pace of hardware innovation
  - 18 months from 1.0 to 1.1 and from 1.1 to 1.2
  - Goal: a new OpenCL every 18-24 months
  - Committed to backwards compatibility to protect software investments

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec08</td>
<td>OpenCL 1.0 released. Conformance tests released</td>
<td>Dec08</td>
</tr>
<tr>
<td>Jun10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nov11</td>
<td>OpenCL 1.2 Specification and conformance tests released</td>
<td></td>
</tr>
<tr>
<td>Jul13</td>
<td>OpenCL 2.0 Provisional Specification released for public review</td>
<td></td>
</tr>
</tbody>
</table>
OpenCL Working Group within Khronos

- Diverse industry participation
  - Processor vendors, system OEMs, middleware vendors, application developers.
- OpenCL became an important standard upon release by virtue of the market coverage of the companies behind it.
One **Host** and one or more **OpenCL Devices**
  - Each OpenCL Device is composed of one or more **Compute Units**
    - Each Compute Unit is divided into one or more **Processing Elements**
  - Memory divided into **host memory** and **device memory**
OpenCL Platform Example (One node, two CPU sockets, two GPUs)

**CPUs:**
- Treated as one OpenCL device
  - One CU per core
  - 1 PE per CU, or if PEs mapped to SIMD lanes, \( n \) PEs per CU, where \( n \) matches the SIMD width
- Remember:
  - the CPU will also have to be its own host!

**GPUs:**
- Each GPU is a separate OpenCL device
- One CU per Streaming Multiprocessor
- Can use CPU and all GPU devices concurrently through OpenCL

CU = Compute Unit; PE = Processing Element
The **BIG** idea behind OpenCL

- Replace loops with functions (a kernel) executing at each point in a problem domain
  - E.g., process a 1024x1024 image with one kernel invocation per pixel or 1024x1024=1,048,576 kernel executions

<table>
<thead>
<tr>
<th>Traditional loops</th>
<th>Data Parallel OpenCL</th>
</tr>
</thead>
</table>

```c
void mul(const int n,
         const float *a,
         const float *b,
         float *c)
{
    int i;
    for (i = 0; i < n; i++)
        c[i] = a[i] * b[i];
}
```

```c
__kernel void
mul(__global const float *a,
     __global const float *b,
     __global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}
```

// many instances of the kernel,
// called *work-items*, execute
// in parallel
An N-dimensional domain of work-items

- **Global Dimensions:**
  - 1024x1024 (whole problem space)
- **Local Dimensions:**
  - 128x128 (work-group, executes together)

Choose the dimensions that are “best” for your algorithm

Synchronization between work-items possible only within work-groups:

Cannot synchronize between work-groups within a kernel
OpenCL N Dimensional Range (NDRange)

- The problem we want to compute should have some dimensionality;
  - For example, compute a kernel on all points in a cube
- When we execute the kernel we specify up to 3 dimensions
- We also specify the total problem size in each dimension – this is called the global size
- We associate each point in the iteration space with a work-item
OpenCL N Dimensional Range (NDRange)

- Work-items are grouped into `work-groups`; work-items within a work-group can share `local memory` and can `synchronize`.
- We can specify the number of work-items in a work-group – this is called the `local` (work-group) size.
- Or the OpenCL run-time can choose the work-group size for you (usually not optimally).
OpenCL Memory model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a work-group
- **Global Memory / Constant Memory**
  - Visible to all work-groups
- **Host memory**
  - On the CPU

Memory management is **explicit**: You are responsible for moving data from host → global → local *and* back
Context and Command-Queues

- **Context:**
  - The environment within which kernels execute and in which synchronization and memory management is defined.

- The **context** includes:
  - One or more devices
  - Device memory
  - One or more command-queues

- All **commands** for a device (kernel execution, synchronization, and memory transfer operations) are submitted through a **command-queue**.

- Each **command-queue** points to a single device within a context.
Execution model (kernels)

- OpenCL execution model … define a problem domain and execute an instance of a **kernel** for each point in the domain

```c
__kernel void times_two(
    __global float* input,
    __global float* output)
{
    int i = get_global_id(0);
    output[i] = 2.0f * input[i];
}
```

Input

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

Output

| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 | 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 | 48 | 50 |
Building Program Objects

- The **program object** encapsulates:
  - A context
  - The program kernel source or binary
  - List of target devices and build options
- The C API build process to create a program object:
  - `clCreateProgramWithSource()`
  - `clCreateProgramWithBinary()`

```c
__kernel void horizontal_reflect(read_only image2d_t src, write_only image2d_t dst)
{
    int x = get_global_id(0);  // x-coord
    int y = get_global_id(1);  // y-coord
    int width = get_image_width(src);
    float4 src_val = read_imagef(src, sampler, (int2)(width-1-x, y));
    write_imagef(dst, (int2)(x, y), src_val);
}
```

OpenCL uses **runtime compilation** … because in general you don’t know the details of the target device when you ship the program
Example: vector addition

- The “hello world” program of data parallel programming is a program to add two vectors

  \[ C[i] = A[i] + B[i] \text{ for } i=0 \text{ to } N-1 \]

- For the OpenCL solution, there are two parts
  - Kernel code
  - Host code
Vector Addition - Kernel

```c
__kernel void vadd(__global const float *a,
                    __global const float *b,
                    __global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```
Vector Addition – Host

- The **host program** is the code that runs on the host to:
  - Setup the environment for the OpenCL program
  - Create and manage kernels
- 5 simple steps in a basic host program:
  1. Define the **platform** … platform = devices+context+queues
  2. Create and Build the **program** (dynamic library for kernels)
  3. Setup **memory** objects
  4. Define the **kernel** (attach arguments to kernel functions)
  5. Submit **commands** … transfer memory objects and execute kernels

Please, refer to the reference card. This will help you get used to the reference card and how to pull information from the card and express it in code.
1. Define the platform

- Grab the first available platform:
  ```c
  err = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
  ```

- Use the first CPU device the platform provides:
  ```c
  err = clGetDeviceIDs(firstPlatformId, CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);
  ```

- Create a simple context with a single device:
  ```c
  context = clCreateContext(firstPlatformId, 1, &device_id, NULL, NULL, &err);
  ```

- Create a simple command-queue to feed our device:
  ```c
  commands = clCreateCommandQueue(context, device_id, 0, &err);
  ```
Command-Queues

- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Used to define independent streams of commands that don’t require synchronization
Command-Queue execution details

*Command queues* can be configured in different ways to control how commands execute:

- **In-order queues:**
  - Commands are enqueued and complete in the order they appear in the program (program-order).

- **Out-of-order queues:**
  - Commands are enqueued in program-order but can execute (and hence complete) in any order.

- Execution of commands in the command-queue are guaranteed to be completed at synchronization points.
2. Create and Build the program

- Define source code for the kernel-program as a string literal (great for toy programs) or read from a file (for real applications).

- Build the program object:

  ```
  program = clCreateProgramWithSource(context, 1
  (const char**) &KernelSource, NULL, &err);
  ```

- Compile the program to create a “dynamic library” from which specific kernels can be pulled:

  ```
  err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);
  ```
Error messages

• Fetch and print error messages:

```c
if (err != CL_SUCCESS) {
    size_t len;
    char buffer[2048];
    clGetProgramBuildInfo(program, device_id,
                          CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
    printf("%s\n", buffer);
}
```

• Important to do check all your OpenCL API error messages!

• Easier in C++ with try/catch
3. Setup Memory Objects

- For vector addition we need 3 memory objects, one each for input vectors A and B, and one for the output vector C.
- Create input vectors and assign values on the host:
  ```c
  float h_a[LENGTH], h_b[LENGTH], h_c[LENGTH];
  for (i = 0; i < length; i++) {
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
  }
  ```
- Define OpenCL memory objects:
  ```c
  d_a = clCreateBuffer(context, CL_MEM_READ_ONLY,
                      sizeof(float)*count, NULL, NULL);
  d_b = clCreateBuffer(context, CL_MEM_READ_ONLY,
                      sizeof(float)*count, NULL, NULL);
  d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                      sizeof(float)*count, NULL, NULL);
  ```
- Memory Objects:
  - A handle to a reference-counted region of global memory.
Creating and manipulating buffers

- Buffers are declared on the host as type: cl_mem

- Arrays in host memory hold your original host-side data:
  ```
  float h_a[LENGTH], h_b[LENGTH];
  ```

- Create the buffer (d_a), assign sizeof(float)*count bytes from “h_a” to the buffer and copy it into device memory:
  ```
  cl_mem d_a = clCreateBuffer(context,
  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  sizeof(float)*count, h_a, NULL);
  ```
Creating and manipulating buffers

• Other common memory flags include:
  CL_MEM_WRITE_ONLY, CL_MEM_READ_WRITE

• These are from the point of view of the device

• Submit command to copy the buffer back to host memory at “h_c”:
  – CL_TRUE = blocking, CL_FALSE = non-blocking

  clEnqueueReadBuffer(queue, d_c, CL_TRUE,
  sizeof(float)*count, h_c,
  NULL, NULL, NULL);
4. Define the kernel

- Create kernel object from the kernel function “vadd”:

  ```
  kernel = clCreateKernel(program, "vadd", &err);
  ```

- Attach arguments of the kernel function “vadd” to memory objects:

  ```
  err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
  err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
  err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
  err |= clSetKernelArg(kernel, 3, sizeof(unsigned int),
                       &count);
  ```
5. Enqueue commands

- Write Buffers from host into global memory (as non-blocking operations):

  ```
  err = clEnqueueWriteBuffer(commands, d_a, CL_FALSE, 
  0, sizeof(float)*count, h_a, 0, NULL, NULL);
  err = clEnqueueWriteBuffer(commands, d_b, CL_FALSE, 
  0, sizeof(float)*count, h_b, 0, NULL, NULL);
  ```

- Enqueue the kernel for execution (note: in-order so OK):

  ```
  err = clEnqueueNDRangeKernel(commands, kernel, 1, 
  NULL, &global, &local, 0, NULL, NULL);
  ```
5. Enqueue commands

- Read back result (as a blocking operation). We have an in-order queue which assures the previous commands are completed before the read can begin.

```c
err = clEnqueueReadBuffer(commands, d_c, CL_TRUE,
            sizeof(float)*count, h_c, 0, NULL, NULL);
```
Vector Addition – Host Program

It's complicated, but most of this is “boilerplate” and not as bad as it looks.
OpenCL C for Compute Kernels

- Derived from **ISO C99**
  - A few *restrictions*: no recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported (#include etc.)
- Built-in data types
  - Scalar and vector data types, pointers
  - Data-type conversion functions:
    - `convert_type<_sat>_<roundingmode>`
  - Image types:
    - `image2d_t`, `image3d_t` and `sampler_t`
OpenCL C for Compute Kernels

• Built-in functions — *mandatory*
  – Work-Item functions, math.h, read and write image
  – Relational, geometric functions, synchronization functions
  – printf (v1.2 only, so not currently for NVIDIA GPUs)

• Built-in functions — *optional* (called “extensions”)
  – Double precision, atomics to global and local memory
  – Selection of rounding mode, writes to image3d_t surface
OpenCL C Language
Highlights

• Function qualifiers
  – __kernel qualifier declares a function as a kernel
    • I.e. makes it visible to host code so it can be enqueued
  – Kernels can call other kernel-side functions

• Address space qualifiers
  – __global, __local, __constant, __private
  – Pointer kernel arguments must be declared with an address space qualifier

• Work-item functions
  – get_work_dim(), get_global_id(), get_local_id(), get_group_id()

• Synchronization functions
  – Barriers - all work-items within a work-group must execute the barrier function before any work-item can continue
  – Memory fences - provides ordering between memory operations
Host programs can be “ugly”

- OpenCL’s goal is extreme portability, so it exposes everything (i.e. it is quite verbose!).
- But most of the host code is the same from one application to the next – the re-use makes the verbosity a non-issue.
- You can package common API combinations into functions or even C++ or Python classes to make the reuse more convenient.
The C++ Interface

• Khronos has defined a common C++ header file containing a high level interface to OpenCL, cl.hpp
• This interface is dramatically easier to work with
• Key features:
  – Uses common defaults for the platform and command-queue, saving the programmer from extra coding for the most common use cases
  – Simplifies the basic API by bundling key parameters with the objects rather than requiring verbose and repetitive argument lists
  – Ability to “call” a kernel from the host, like a regular function
  – Error checking can be performed with C++ exceptions

1 especially for C++ programmers…
OpenCL Memory model

- **Private Memory**
  - Per work-item

- **Local Memory**
  - Shared within a work-group

- **Global/Constant Memory**
  - Visible to all work-groups

- **Host memory**
  - On the CPU

Memory management is *explicit*:
You are responsible for moving data from host → global → local *and* back
OpenCL Memory model

• Private Memory
  – Fastest & smallest: $O(10)$ words/WI

• Local Memory
  – Shared by all WI’s in a work-group
  – But not shared between work-groups!
  – $O(1-10)$ Kbytes per work-group

• Global/Constant Memory
  – $O(1-10)$ Gbytes of Global memory
  – $O(10-100)$ Kbytes of Constant memory

• Host memory
  – On the CPU - GBytes

Memory management is **explicit:**
$O(1-10)$ Gbytes/s bandwidth to discrete GPUs for Host $<->$ Global transfers
Private Memory

• Managing the memory hierarchy is one of the most important things to get right to achieve good performance.

• Private Memory:
  – A very scarce resource, only a few tens of 32-bit words per Work-Item at most.
  – If you use too much it spills to global memory or reduces the number of Work-Items that can be run at the same time, potentially harming performance.
  – Think of these like registers on the CPU.

* Occupancy on a GPU
Local Memory*

• Tens of KBytes per Compute Unit
  – As multiple Work-Groups will be running on each CU, this means only a fraction of the total Local Memory size is available to each Work-Group
• Assume O(1-10) KBytes of Local Memory per Work-Group
  – Your kernels are responsible for transferring data between Local and Global/Constant memories … there are optimized library functions to help
• Use Local Memory to hold data that can be reused by all the work-items in a work-group
• Access patterns to Local Memory affect performance in a similar way to accessing Global Memory
  – Have to think about things like coalescence & bank conflicts

* Typical figures for a 2013 GPU
Local Memory

- Local Memory doesn’t always help…
  - CPUs don’t have special hardware for it
  - This can mean excessive use of Local Memory might slow down kernels on CPUs
  - GPUs now have effective on-chip caches which can provide much of the benefit of Local Memory but without programmer intervention
  - So, your mileage may vary!
The Memory Hierarchy

Bandwidths

- Private memory: $O(2-3)$ words/cycle/WI
- Local memory: $O(10)$ words/cycle/WG
- Global memory: $O(100-200)$ GBytes/s
- Host memory: $O(1-100)$ GBytes/s

Sizes

- Private memory: $O(10)$ words/WI
- Local memory: $O(1-10)$ KBytes/WG
- Global memory: $O(1-10)$ GBytes
- Host memory: $O(1-100)$ GBytes

Speeds and feeds approx. for a high-end discrete GPU, circa 2011
Memory Consistency

- OpenCL uses a **relaxed consistency** memory model; i.e.
  - The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.
- Within a work-item:
  - Memory has load/store consistency to the work-item’s private view of memory, i.e. it sees its own reads and writes correctly
- Within a work-group:
  - Local memory is consistent between work-items at a barrier.
- Global memory is consistent within a work-group at a barrier, **but not** guaranteed across different work-groups!!
  - This is a common source of bugs!
- Consistency of memory shared between **commands** (e.g. kernel invocations) is enforced by **synchronization** (barriers, events, in-order queue)
Consider N-dimensional domain of work-items

- **Global** Dimensions:
  - 1024x1024 (whole problem space)

- **Local** Dimensions:
  - 128x128 (**work-group**, executes together)

Synchronization: when multiple units of execution (e.g. work-items) are brought to a known point in their execution. Most common example is a barrier … i.e. all units of execution “in scope” arrive at the **barrier** before any proceed.

Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within a kernel
Work-Item Synchronization

• Within a work-group
  
  ```c
  void barrier()
  ```
  
  Takes optional flags
  
  - CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE
  
  - A work-item that encounters a barrier() will wait until ALL work-items in its work-group reach the barrier()
  
  - **Corollary**: If a barrier() is inside a branch, then the branch **must** be taken by either:
    
    • ALL work-items in the work-group, OR
    • NO work-item in the work-group

• Across work-groups
  
  - No guarantees as to where and when a particular work-group will be executed relative to another work-group
  
  - Cannot exchange data, or have barrier-like synchronization between two different work-groups! (Critical issue!)
  
  - **Only solution**: finish the kernel and start another

Ensure correct order of memory operations to local memory (with flushes or queuing a memory fence)! or global
• Targets a broader range of CPU-like and GPU-like devices than CUDA
  - Targets devices produced by multiple vendors
  - Many features of OpenCL are optional and may not be supported on all devices
• OpenCL codes must be prepared to deal with much greater hardware diversity
• A single OpenCL kernel will likely not achieve peak performance on all device types

Performance???
Portable performance in OpenCL

• Portable performance is always a challenge, more so when OpenCL devices can be so varied (CPUs, GPUs, …)

• But OpenCL provides a powerful framework for writing performance portable code

• The following slides are general advice on writing code that should work well on most OpenCL devices

Tremendous amount of computing power available

1170 GFLOPs peak

1070 GFLOPs peak
Optimization issues

• Efficient access to memory
  – Memory coalescing
    • Ideally get work-item i to access data[i] and work-item j to access data[j] at the same time etc.
  – Memory alignment
    • Padding arrays to keep everything aligned to multiples of 16, 32 or 64 bytes
• Number of work-items and work-group sizes
  – Ideally want at least 4 work-items per PE in a Compute Unit on GPUs
  – More is better, but diminishing returns, and there is an upper limit
    • Each work item consumes PE finite resources (registers etc)
• Work-item divergence
  – What happens when work-items branch?
  – Actually a SIMD data parallel model
  – Both paths (if-else) may need to be executed (branch divergence), avoid where possible (non-divergent branches are termed uniform)
Memory layout is critical to performance

• “Structure of Arrays vs. Array of Structures” problem:

```c
struct { float x, y, z, a; } Point;
```

• Structure of Arrays (SoA) suits memory coalescence on GPUs

```
x x x x ... y y y y ... z z z z ... a a a a a ...
```

Adjacent work-items like to access adjacent memory

• Array of Structures (AoS) may suit cache hierarchies on CPUs

```
x y z a ... x y z a ... x y z a ...
```

Individual work-items like to access adjacent memory
Advice for performance portability

• Optimal Work-Group sizes will differ between devices
  – E.g. CPUs tend to prefer 1 Work-Item per Work-Group, while GPUs prefer lots of Work-Items per Work-Group (usually a multiple of the number of PEs per Compute Unit, i.e. 32, 64 etc.)

• From OpenCL v1.1 you can discover the preferred Work-Group size multiple for a kernel once it’s been built for a specific device
  – Important to pad the total number of Work-Items to an exact multiple of this
  – Again, will be different per device

• The OpenCL run-time will have a go at choosing good EnqueueNDRangeKernel dimensions for you
  – With very variable results

• Your mileage will vary, the best strategy is to write adaptive code that makes decisions at run-time
Tuning Knobs

some general issues

- Tiling size (work-group sizes, dimensionality etc.)
  - For block-based algorithms (e.g. matrix multiplication)
  - Different devices might run faster on different block sizes
- Data layout
  - Array of Structures or Structure of Arrays (AoS vs. SoA)
  - Column or Row major
- Caching and prefetching
  - Use of local memory or not
  - Extra loads and stores assist hardware cache?
- Work-item / work-group data mapping
  - Related to data layout
  - Also how you parallelize the work
- Operation-specific tuning
  - Specific hardware differences
  - Built-in trig / special function hardware
  - Double vs. float (vs. half)
Auto tuning

• Q: How do you know what the best parameter values for your program are?
  – What is the best work-group size, for example

• A: Try them all! (Or a well chosen subset)

• This is where auto tuning comes in
  – Run through different combinations of parameter values and optimize the runtime (or another measure) of your program.
Hydro is a simplified version of RAMSES (CEA, France astrophysics code to study large scale structure and galaxy formation)

Hydro main features:
- regular cartesian mesh (no AMR)
- solves compressible Euler equations of hydrodynamics
- finite volume method, second order Godunov scheme
- it uses a Riemann solver numerical flux at the interfaces

How much fast? The Hydro benchmark
The Hydro benchmark

Hydro is about 1K lines of code and has been ported to different programming environment and architectures, including accelerators. In particular:
- initial Fortran branch including OpenMP, MPI, hybrid MPI+OpenMP
- C branch for CUDA, OpenCL, OpenACC, UPC
# Hydro run comparison

**performances of OpenCL code are very good (better than CUDA!)**

<table>
<thead>
<tr>
<th>Device/version</th>
<th>Elapsed time (sec.) without initialization</th>
<th>EfficiencyLoss (with respect to the best timing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA K20C</td>
<td>52.37</td>
<td>0.24</td>
</tr>
<tr>
<td>OpenCL K20C</td>
<td>42.09</td>
<td>0</td>
</tr>
<tr>
<td>MPI (1 process)</td>
<td>780.8</td>
<td>17.5</td>
</tr>
<tr>
<td>MPI+OpenMP (16 OpenMP threads)</td>
<td>109.7</td>
<td>1.60</td>
</tr>
<tr>
<td>MPI+OpenMP MIC (240 threads)</td>
<td>147.5</td>
<td>2.50</td>
</tr>
<tr>
<td>OpenACC (Pgi)</td>
<td>N.A.</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

*More than 16 Intel Xeon SandyBridge cores are needed to compare OpenCL 1 K20 device*

*Intel MIC preliminary run on CINECA prototype. 240 threads, vectorized code, KMP_AFFINITY=balanced*

*OpenAcc run it fails using Pgi compiler*
## Hydro OpenCL scaling

OpenCL+MPI run, varying the number of NVIDIA Tesla K20 devices, 4091x4091 domain, 100 iterations. Performances are good. Scalability is limited by domain size.

<table>
<thead>
<tr>
<th>Number of K20 devices</th>
<th>Elapsed time (sec.) without initialization</th>
<th>Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>23.5</td>
<td>1.7</td>
</tr>
<tr>
<td>4</td>
<td>12.2</td>
<td>3.4</td>
</tr>
<tr>
<td>8</td>
<td>8.56</td>
<td>4.9</td>
</tr>
<tr>
<td>16</td>
<td>5.70</td>
<td>7.3</td>
</tr>
</tbody>
</table>
How much fast? The EuroBen Benchmark

The EuroBen Benchmark Group provides benchmarks for the evaluation of the performance for scientific and technical computing on single processor cores and on parallel computers systems using standard parallel tool (OpenMP, MPI, …) but also emerging standard (OpenCL, Cilk, …)

- Programs are available in Fortran and C
- The benchmark codes range from measuring the performance of basic operations and mathematical functions to skeleton applications.
- Cineca started a new activity in the official PRACE framework to test and validate EuroBen benchmarks on Intel MIC architecture (V. Ruggiero-C.Cavazzoni).
MOD2F benchmark

16 OpenMP threads, size=$2^{22}$

Host: Intel Xeon SandyBridge cores
MOD2F benchmark

16 MPI process, size=$2^{19}$

Host: Intel Xeon SandyBridge cores
MOD2F benchmark

OpenCL, kernel only, size=$2^{23}$

Host: Intel Xeon SandyBridge cores
MOD2F benchmark

240 OpenMP threads, size=$2^{22}$

Native: Intel MIC, up to 240 hw threads
MOD2F benchmark

Native: Intel MIC, up to 240 hw threads

16 MPI process, size=$2^{23}$
MOD2F benchmark

OpenCL, kernel only, size=$2^{23}$

Native: Intel MIC, up to 240 hw threads
How much fast? The EuroBen Benchmark results

- OpenCL performances are quite good (better than expected) both on host and device. OpenCL test is faster than MPI for native runs.
- Host<->device memory transfers are a potential bottlenecks for OpenCL.
  - There is room for improvement…..OpenCL performance portability is always a challenge.
Eurora

- Eurora CINECA-Eurotech prototype
- 1 rack
- Two Intel SandyBridge and
- two NVIDIA K20 cards per node or:
- Two Intel MIC card per node
- Hot water cooling
- Energy efficiency record (up to 3210 MFLOPs/w)
- 100 TFLOPs sustained
Running environment

**NVIDIA Tesla K20**
- 13 Multiprocessors
- 2496 CUDA Cores
- 5 GB of global memory
- GPU clock rate 760MHz

**Intel MIC Xeon Phi**
- 236 compute units
- 8 GB of global memory
- CPU clock rate 1052 MHz
Setting up OpenCL on Europa

- Login on front-end.
Then:

```bash
module load profile/advanced
module load intel_opencl/none--intel--cs-xe-2013--binary
```

It defines:

```bash
INTEL_OPENCL_INCLUDE

and

INTEL_OPENCL_LIB
```

environmental variables that can be used:

```bash
cc -I$INTEL_OPENCL_INCLUDE -L$INTEL_OPENCL_LIB -lOpenCL vadd.c -o vadd
```
Running on Intel

Intel OpenCL platform found and 3 devices (cpu and Intel MIC card)

Intel MIC device was selected

Results are OK no matter what performances
Exercise 1

- **Goal:**
  - To inspect and verify that you can run an OpenCL kernel on Eurora machines

- **Procedure:**
  - Take the provided C `vadd.c` and `vadd.cl` source programs from VADD directory
  - Compile and link `vadd.c`
  - Run on NVIDIA or Intel platform.

- **Expected output:**
  - A message verifying that the vector addition completed successfully
  - Some useful info about OpenCL environment (Intel and NVIDIA)
Matrix multiplication: sequential code

```c
void mat_mul(int Mdim, int Ndim, int Pdim, float *A, float *B, float *C)
{
  int i, j, k;
  for (i = 0; i < Ndim; i++) {
    for (j = 0; j < Mdim; j++) {
      for (k = 0; k < Pdim; k++) {
        C[i * Ndim + j] += A[i * Ndim + k] * B[k * Pdim + j];
      }
    }
  }
}
```

We calculate $C=AB$, $\text{dim} A = (N \times P)$, $\text{dim} B = (P \times M)$, $\text{dim} C = (N \times M)$

Dot product of a row of $A$ and a column of $B$ for each element of $C$
Matrix multiplication: sequential code

```c
void mat_mul(int Mdim, int Ndim, int Pdim,
             float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < Ndim; i++) {
        for (j = 0; j < Mdim; j++) {
            for (k = 0; k < Pdim; k++) {
                // C(i, j) = sum(over k) A(i,k) * B(k,j)
                C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
            }
        }
    }
}
```

We turn this into an OpenCL kernel!
OpenCL mapping

- In OpenCL:

  - `get_global_size(0)`: Index Space
  - `get_local_size(0)`: Work Group (0,0)
  - `get_global_size(1)`: Work Group (0, 1)
  - `get_local_size(1)`: Work Item (0, 0)
OpenCL mapping (again)

**Kernels: Work-item and Work-group Example**

dimension: 2  
global size: 32x32=1024  
num of groups: 16

local id: (4,2)  
global id: (28,10)

workgroup id: (3,1)  
local size: 8x8=64
Matrix multiplication: OpenCL
kernel (1/2)

__kernel void mat_mul(
    const int Mdim, const int Ndim, const int Pdim,
    __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    for (i = 0; i < Ndim; i++) {
        for (j = 0; j < Mdim; j++) {
            // C(i, j) = sum(over k) A(i,k) * B(k,j)
            for (k = 0; k < Pdim; k++) {
                C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
            }
        }
    }
}
Matrix multiplication: OpenCL kernel (2/2)

__kernel void mat_mul(
    const int Mdim, const int Ndim, const int Pdim,
    __global float *A, __global float *B, __global float *C)
{
    int i, j, k;

    i = get_global_id(0);
    j = get_global_id(1);

    for (k = 0; k < Pdim; k++) {
        // C(i, j) = sum(over k) A(i,k) * B(k,j)
        C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
    }
}

Remove outer loops and set work-item co-ordinates
Matrix multiplication: OpenCL kernel

__kernel void mat_mul(
    const int Mdim, const int Ndim, const int Pdim,
    __global float *A, __global float *B, __global float *C)
{

    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);

    // C(i, j) = sum(over k) A(i,k) * B(k,j)
    for (k = 0; k < Pdim; k++) {
        C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
    }
}

Matrix multiplication: OpenCL kernel improved

Rearrange and use a local scalar for intermediate C element values (a common optimization in Matrix Multiplication functions)

```
__kernel void mat_mul(
    const int Mdim,
    const int Ndim,
    const int Pdim,
    __global float *A,
    __global float *B,
    __global float *C)
{
    int k;
    int i = get_global_id(0);
    int j = get_global_id(1);
    float tmp = 0.0f;
    for (k = 0; k < Pdim; k++)
        tmp += A[i*Ndim+k]*B[k*Pdim+j];
    C[i*Ndim+j] += tmp;
}
```
Exercise 2

- **Goal:**
  - To inspect and verify that you can run the first mat_mul OpenCL kernels

- **Procedure:**
  - Take the provided C `mat_mul.c.1` program from SOURCES dir.
  - Look at the host code and identify the API calls in the host code. Compare them against the vadd source code.
  - Try to understand regions where timing the execution, profiling and error checks (ifdefs within C code).
  - Rename it `mat_mul.c`, compile and link on front-end.
  - Run both `mat_mul.cl.1` and `mat_mul.cl.1.1` kernels.
  - Compare results

- **Expected output:**
  - A message verifying that the matrix multiply succeeded, GFLOPS and time results (if activated)
Optimizing matrix multiplication

- MM cost determined by FLOPS and memory movement:
  - $2n^3 = O(n^3)$ FLOPS
  - Operates on $3n^2 = O(n^2)$ numbers

- To optimize matrix multiplication, we must ensure that for every memory access we execute as many FLOPS as possible.

- Outer product algorithms are faster, but for pedagogical reasons, let’s stick to the simple dot-product algorithm.

$$C(i,j) = C(i,j) + A(i,:) \times B(:,j)$$

- Dot product of a row of A and a column of B for each element of C

- We will work with work-item/work-group sizes and the memory model to optimize matrix multiplication
Optimizing matrix multiplication

- There may be significant overhead to manage work-items and work-groups.
- So let’s have each work-item compute a full row of $C$

$C(i,j) = \sum A(i,:) \times B(:,j)$

Dot product of a row of $A$ and a column of $B$ for each element of $C$
An N-dimension domain of work-items

- **Global** Dimensions: 1024 (1D)
  Whole problem space (index space)
- **Local** Dimensions: 16 (work-items per work-group)
  \[1024 / 64 = 64\] work-groups in total
__kernel void mmul(
const int Mdim, const int Ndim, const int Pdim,
__global float *A, __global float *B, __global float *C)
{
    Do a whole row of C per work-item
    int i = get_global_id(0);
    float tmp;
    for (j = 0; j < Mdim; j++) {
        // Mdim is width of rows in C
        tmp = 0.0f;
        for (k = 0; k < Pdim; k++)
            tmp += A[i*Ndim+k] * B[k*Pdim+j];
        C[i*Ndim+j] += tmp;
    }
}
Exercise 3

- **Goal:**
  - To inspect and verify that you can run the second mat_mul OpenCL kernel

- **Procedure:**
  - Take the provided *C mat_mul.c.0* host program from SOURCES dir, rename it *mat_mul.c* and compare against the previous mat_mul source code.
  - Take the *mat_mul.cl.0* kernel from SKEL directory
  - Insert the correct call, compile and link.
  - Run on NVIDIA and Intel plaform.

- **Expected output:**
  - A message verifying that the mat_mul completed successfully
Optimizing matrix multiplication

- Notice that, in one row of C, each element reuses the same row of A.
- Let’s copy that row of A into private memory of the work-item that’s (exclusively) using it to avoid the overhead of loading it from global memory for each C(i,j) computation.
Matrix multiplication: OpenCL kernel (3/3)

__kernel void mat_mul(
    const int Mdim,
    const int Ndim,
    const int Pdim,
    __global float *A,
    __global float *B,
    __global float *C)
{
    int k, j;
    int i = get_global_id(0);

    float Awrk[1024];
    float tmp;

    for (k = 0; k < Pdim; k++)
        Awrk[k] = A[i*Ndim+k];

    for (j = 0; j < Mdim; j++) {
        tmp = 0.0f;
        for (k = 0; k < Pdim; k++)
            tmp += Awrk[k]*B[k*Pdim+j];
        C[i*Ndim+j] += tmp;
    }

    Setup a work array for A in private memory and copy into it from global memory before we start with the matrix multiplications.

    (Actually, this is using far more private memory than we'll have and so Awrk[] will be spilled to global memory)
Exercise 4

- **Goal:**
  - To inspect and verify that you can run the third mat_mul OpenCL kernel

- **Procedure:**
  - Take the provided C `mat_mul.c.00` host program from SOURCES dir and rename it `mat_mul.c`.
  - Take `the mat_mul.cl.00` kernel from SKEL directory.
  - Insert the correct calls, compile and link.
  - Run on NVIDIA and Intel platform.

- **Expected output:**
  - A message verifying that the mat_mul completed successfully
Matrix Multiplication: using Local Memory

1. Move a block from A
2. Move a block from B
3. Calculate block * block
4. If no finished, goto Step 1.
#define BLOCK_SIZE 16
__kernel void mat_mul(
    const int wA, const int wB, const int hA,
    __global float *A, __global float *B, __global float *C)
{
    // Block index
    int tx = get_local_id(0);
    int ty = get_local_id(1);
    int bx = get_group_id(0);
    int by = get_group_id(1);

    // Index of the first sub-matrix of A processed
    // by the block
    //Coalesced
    int indexA=by*BLOCK_SIZE*wA+ty*wA+tx;
    int indexB=bx*BLOCK_SIZE+ty*wB+tx;
    float Csub=0.0f;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int m =0;m<wA/BLOCK_SIZE;m++)
    {
        // Declaration of the local memory array As
        // used to store the sub-matrix of A
        __local float As[BLOCK_SIZE][BLOCK_SIZE];

        // Declaration of the local memory array Bs
        // used to store the sub-matrix of B
        __local float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory
        // to local memory; each thread loads
        // one element of each matrix
        As[ty][tx] = A[indexA];
        Bs[ty][tx] = B[indexB];
        indexA+=BLOCK_SIZE;
        indexB+=wB*BLOCK_SIZE;

        // Synchronize to make sure the matrices
        // are loaded
        barrier(CLK_LOCAL_MEM_FENCE);

        // MM using Local Memory (2)
        1. Move a block from A.
        2. Move a block from B.
Matrix Multiplication: using Local Memory (3)

// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += As[ty][k] * Bs[k][tx];
// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of A and B in the next iteration
 barrier(CLK_LOCAL_MEM_FENCE);
}
// Write the block sub-matrix to device memory;
// each thread writes one element
int
    indexC=wB*BLOCK_SIZE*by+BLOCK_SIZE*bx+wB*ty+tx;
C[indexC] = Csub;

3. Calculate block * block.
Exercise 5

• **Goal:**
  – To inspect and verify that you can run the fourth mat_mul OpenCL kernel

• **Procedure:**
  – Take the provided C *mat_mul.c.3* host program from SOURCES dir, rename it *mat_mul.c*.
  – Take the *mat_mul.cl.3* kernel from SKEL directory.
  – Insert the correct calls, compile and link.
  – Run on NVIDIA and Intel platform.

• **Expected output:**
  – A message verifying that the mat_mul completed successfully
Exercise 5 (again)

• Additional Goal:
  – To inspect and verify the performances of this mat_mul OpenCL kernel

• Procedure:
  – Take the `mat_mul.cl.3uncoalesced` kernel from SOURCES directory.
  – Compile and link.
  – Run on NVIDIA and Intel platform.

• Expected output:
  – You should verify performance degradation. Why?
From work-group to hw threads

- OpenCL kernels are structured into work groups that map to device compute units.
- Compute units on GPUs consist of SIMT processing elements.
- Hw creates **wavefronts (warps)** grouping threads of a WG (dimension zero first).
- All threads in a wavefront execute the same instruction.
- Thread mapping determine which thread access which data!
  - Proper mapping can align to hw (greater performances)
  - Improper mapping can be disastrous
Thread mapping

• To ensure coalesced accesses consecutive threads in a warp should be mapped to columns of B and C.

```c
__kernel void mat_mul(
    const int Mdim, const int Ndim, const int Pdim,
    __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
    for (k = 0; k < Pdim; k++) {
        C[i][j] += A[i][k] * B[k][j];
    }
}
```

Consecutive threads (i) are mapped to different rows of matrix C

Highly inefficient memory access
Thread mapping

- To ensure coalesced accesses consecutive threads in a warp should be mapped to columns of B and C.

```c
__kernel void mat_mul(
const int Mdim, const int Ndim, const int Pdim,
__global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
    for (k = 0; k < Pdim; k++) {
        C[j][i] += A[j][k] * B[k][i];
    }
}
```

Consecutive threads (i) are mapped to consecutive rows of matrices B and C.

Highly efficient memory access
Making matrix multiplication really fast

- Tiled (or blocked) algorithm can improve data reuse on architectures where at least two memory hierarchies exist:
  1. **Slow and big memory**
  2. **Fast but small memory**
- To utilize it, try to maximize data reuse, so instead of producing one element of C, compute a block.
An 2-dimension domain of tiles

- For example: total number of tiles in N direction: 1024
- This number is equal to 16 (work-items per work-group size in one dimension) * 64 (TILE_SIZE_N parameter)
Consider $A,B,C$ to be divided into blocks (tilings) along the two dimensions.

```plaintext
for i = 0 to NUM_OF_TILES_M-1
    for j = 1 to NUM_OF_TILES_N-1
        C_BLOCK=ZERO_MATRIX(TILE_SIZE_M,TILE_SIZE_N);
        for k = 0 to size-1
            for ib = 0 to TILE_SIZE_M-1
                for jb = 0 to TILE_SIZE_N-1
                    C_BLOCK(jb,ib)=C_BLOCK(jb,ib)+A(k,i*TILE_SIZE_M+ib)*B(j*TILE_SIZE_N+jb,k)
                end for
            end for
        end for
    end for
end for
```

$TILE_SIZE_N*TILE_SIZE_M$ being the number of elements of matrix $C$ computed by one work-item.
Consider \( A, B, C \) to be divided into blocks (tilings) along the two dimensions.

\[
\begin{align*}
\text{for } i = 0 \text{ to } \text{NUM}_M - 1 \\
\text{for } j = 1 \text{ to } \text{NUM}_N - 1 \\
\text{C}_{\text{BLOCK}} = \text{ZERO}_{\text{MATRIX}}(\text{TILE}_M, \text{TILE}_N) ; \\
\text{for } k = 0 \text{ to } \text{size} - 1 \\
\text{for } \text{ib} = 0 \text{ to } \text{TILE}_M - 1 \\
\text{for } \text{jb} = 0 \text{ to } \text{TILE}_N - 1 \\
\text{C}_{\text{BLOCK}}(\text{jb}, \text{ib}) = \text{C}_{\text{BLOCK}}(\text{jb}, \text{ib}) + A(k, \text{i} \cdot \text{TILE}_M + \text{ib}) \times B(\text{j} \cdot \text{TILE}_N + \text{jb}, k) \\
\text{end for} \\
\text{end for} \\
\text{end for} \\
\text{end for} \\
\text{end for} \\
\text{end for} \\
\end{align*}
\]

\[
\begin{align*}
\text{for } \text{ib} = 0 \text{ to } \text{TILE}_M - 1 \\
\text{for } \text{jb} = 0 \text{ to } \text{TILE}_N - 1 \\
\text{C}(\text{j} \cdot \text{TILE}_M + \text{jb}, \text{i} \cdot \text{TILE}_N + \text{ib}) = \text{C}_{\text{BLOCK}}(\text{jb}, \text{ib}) \\
\text{end for} \\
\text{end for} \\
\end{align*}
\]

\( \text{TILE}_GROUP_N \) and \( \text{TILE}_GROUP_M \) being the zero and one dimension of local work-group size.

The local work-group size can be defined as:

\[
\text{int } A_{\text{ind}} = \ldots \\
\text{int } B_{\text{ind}} = \ldots \\
\text{Int } C_{\text{ind}} = \ldots \\
\text{size}/\text{TILE}_N \text{ and } \text{size}/\text{TILE}_M \text{ being the zero and one dimension of NDR range.}
\]
Exercise 6

- **Goal:**
  - To inspect and verify that you can run the fifth `mat_mul` OpenCL kernel

- **Procedure:**
  - Take the provided C `mat_mul.c.4` host program from SOURCES dir, rename it `mat_mul.c`.
  - Take the `mat_mul.cl.4` kernel from SKEL directory.
  - Insert the correct calls, compile and link.
  - Run on NVIDIA and Intel platform.

- **Expected output:**
  - A message verifying that the `mat_mul` completed successfully
Exercise 6 (again)

• Additional Goal:
  – To inspect and verify the performances of this mat_mul OpenCL kernel

• Procedure:
  – Play with TILE* parameters and matrix size.
  – Compile and link.
  – Run on NVIDIA and Intel platform.

• Expected output:
  – You should verify very exciting performance on Intel MIC. Why?
OpenCL on Intel MIC

- Tremendous computing power available (236 compute units…)
- More than 1 TFLOPs peak for double precision computation
- You should understand how to optimize OpenCL code in order to reach peak performances

Is Intel MIC, not a Pentium 4
Intel MIC architecture

- Intel MIC architecture targets highly parallel and computationally intensive applications. Three basic factors:
  1. Threading scalability (60+ Intel CPU cores in a single chip)
  2. Vectorization (wide vector registers and associated SIMD operations)
  3. Memory bandwidth utilization
OpenCL on Intel MIC

- Intel MIC combines many core onto a single chip. Each core runs exactly 4 **hardware threads**. In particular:
  1. **All cores/threads are a single OpenCL device**
  2. **Separate hardware threads are OpenCL CU.**
- In the end, you’ll have parallelism at the work-group level (vectorization) and parallelism between work-groups (threading).
OpenCL on Intel MIC

- To reach performances, the number of work-groups should be not less than `CL_DEVICE_MAX_COMPUTE_UNITS` parameter (more is better).
- Again, automatic vectorization module should be fully utilized. This module:
  - packs adjacent work-items (from dimension 0 of NDRange)
  - executes them with SIMD instructions
- Use the recommended work-group size as multiple of 16 (SIMD width for float, int, …data type).
OpenCL on Intel MIC

- Memory bandwidth should be saturated. Use
  - Prefetching
  - Blocking (tiling)
  - Data structures layout optimized
- All aforementioned techniques have already been applied to matrix multiply kernel, resulting in extreme performances on Intel MIC device.

More than 400 GFLOPs for SGEMM
Himeno Benchmark

- Dr. Ryutaro Himeno, Director of the Advanced Center for Computing and Communication, has developed this benchmark to evaluate performance of incompressible fluid analysis code.
- This benchmark takes measurements to proceed major loops in solving the Poisson's equation solution using the Jacobi iteration method.
- Being the code very simple and easy to compile and to execute, users can measure actual speed (in MFLOPS) immediately on different machines and languages/tools (C, Fortran, OpenMP, MPI, …).
- The most computational part of Himeno Benchmark is restricted to Jacobi routine.
- A simple OpenCL version of Himeno Benchmark was realized in order to test performances on NVIDIA and Intel Platforms.
Jacobi routine: sequential code

```c
Float jacobi(int nn, int MIMX, int MJMX, int MKMX, intimax, int jmax, int kmax, float a1, float a2, float a3, float a4, float b1, float b2, float b3, float c1, float c2, float c3, float p, float wrk1, float bnd, float wrk2)
{
    #define idxz(i,j,k) ((k)+MKMX*((j)+MJMX*(i)))
    int i,j,k,n;
    float gosa, s0, ss;
    for(n=0 ; n< nn ; ++n){
        gosa = 0.0;
        for(i=1 ; i<imax-1 ; ++i)
            for(j=1 ; j<jmax-1 ; ++j)
                for(k=1 ; k<kmax-1 ; ++k){
                    s0 = a1[idxz(i,j,k)] * p[idxz(i+1,j,k)]
                        + a2[idxz(i,j,k)] * p[idxz(i,j+1,k)]
                        + a3[idxz(i,j,k)] * p[idxz(i,j,k+1)]
                        + b1[idxz(i,j,k)] * ( p[idxz(i+1,j+1,k)] - p[idxz(i+1,j,k-1)]
                                      - p[idxz(i+1,j-1,k)] + p[idxz(i+1,j,k) ] )
                        + b2[idxz(i,j,k)] * ( p[idxz(i+1,j,k+1)] - p[idxz(i+1,j,k-1)]
                                      - p[idxz(i+1,j-1,k)] + p[idxz(i+1,j,k) ] )
                        + b3[idxz(i,j,k)] * ( p[idxz(i+1,j+1,k)] - p[idxz(i+1,j,k+1)]
                                      - p[idxz(i+1,j,k-1)] + p[idxz(i+1,j,k) ] )
                        + c1[idxz(i,j,k)] * p[idxz(i-1,j,k) ]
                        + c2[idxz(i,j,k)] * p[idxz(i-1,j,k) ]
                        + c3[idxz(i,j,k)] * p[idxz(i-1,j,k) ]
                        + wrk1[idxz(i,j,k)];
                    ss = ( s0 * a4[idxz(i,j,k)] - p[idxz(i,j,k)] ) * bnd[idxz(i,j,k)];
                    gosa+= ss*ss;
                    wrk2[idxz(i,j,k)] = p[idxz(i,j,k)] + omega * ss;
                }
            for(i=1 ; i<imax-1 ; ++i)
                for(j=1 ; j<jmax-1 ; ++j)
                    for(k=1 ; k<kmax-1 ; ++k)
                        p[idxz(i,j,k)] = wrk2[idxz(i,j,k)];
        }
    return(0);
}
```

Jacobi routine: OpenCL kernel

Float jacobi(int nn, int MIMX, int MJMX, int MKMX, intimax, int jmax, int kmax, float* a1, float* a2, float* a3, float* a4, float* b1, float* b2, float* b3, float* c1, float* c2, float* c3, float* p, float* wrk1, float* bnd, float* wrk2)
{
    #define idxz(i,j,k) ((k)+MKMX*((j)+MJMX*(i)))
    int i, j, k, n;
    float gosa, s0, ss;
    for(n=0; n<nn; ++n)
    {
        gosa = 0.0;
        for(i=1; i<imax-1; i++)
            for(j=1; j<jmax-1; j++)
                for(k=1; k<kmax-1; k++)
                {
                    s0 = a1[idxz(i,j,k)] * p[idxz(i+1,j,k)]
                        + a2[idxz(i,j,k)] * p[idxz(i,j+1,k)]
                        + b1[idxz(i,j,k)] * ( p[idxz(i+1,j+1,k)] - p[idxz(i+1,j-1,k)]
                        - p[idxz(i-1,j+1,k)] + p[idxz(i-1,j-1,k)] )
                        + b2[idxz(i,j,k)] * ( p[idxz(i+1,j+1,k+1)] - p[idxz(i-1,j-1,k+1)]
                        - p[idxz(i-1,j+1,k+1)] + p[idxz(i-1,j-1,k+1)] )
                        + b3[idxz(i,j,k)] * ( p[idxz(i+1,j+1,k+1)] - p[idxz(i+1,j-1,k+1)]
                        - p[idxz(i-1,j+1,k+1)] + p[idxz(i-1,j-1,k+1)] )
                        - c1[idxz(i,j,k)] * p[idxz(i-1,j,k)]
                        + c2[idxz(i,j,k)] * p[idxz(i,j-1,k)]
                        + c3[idxz(i,j,k)] * p[idxz(i,j,k-1)]
                        + wrk1[idxz(i,j,k)];
                    ss = ( s0 * a4[idxz(i,j,k)] - p[idxz(i,j,k)] ) * bnd[idxz(i,j,k)];
                    gosa += ss*ss;
                }
    }
    return(0);
}

End of Update kernel

Swap kernel

int l=get_global_id(0);
int J=get_global_id(1);
int K=get_global_id(2);

wrk2[idxz(i,i,k)] = p[idxz(i,i,k)] + omega * ss;

int l=get_global_id(0);
int J=get_global_id(1);
int K=get_global_id(2);

p[idxz(i,j,k)] = wrk2[idxz(i,j,k)];
Exercise 7

• **Goal:**
  – To inspect and verify that you can run the Himeno Benchmark (OpenCL version)

• **Procedure:**
  – Take the provided C *himenoBMTxps.c* host program from *SOURCES* dir.
  – Take the provided OpenCL *initmt.cl* kernel from *SOURCES* directory
  – Take the *swap.cl* and *update.cl* kernels from *SKEL* directory.
  – Insert the correct calls, compile and link (be careful, only *MIDDLE* or *LARGE* dataset can be selected – use `-DMIDDLE` or equivalent `-`).
  – Run on NVIDIA and Intel platform.

• **Expected output:**
  – A message verifying that the Himeno Benchmark completed successfully
Exercise 7 (again)

- **Additional Goal:**
  - To inspect and verify the performances of Himeno Benchmark OpenCL

- **Procedure:**
  - Play with LOCAL WORK-GROUPS parameter and input size.
  - Compile and link.
  - Run on NVIDIA and Intel platform.

- **Expected output:**
  - You should verify correctness and performances on NVIDIA and Intel MIC.
Himeno Benchmark on Intel MIC

- Hardware prefetching is enabled by default on Intel MIC (from GDDR to L2 cache).
- Compiler insert prefetch instructions into assembly based on kernel code analysis (automatic prefetching).
  - Variables in global and constant address spaces.
  - Well detected memory access pattern
- **Auto Prefetching level** is enabled by default. Can be changed using “-auto-prefetch-level=0/1/2/3” option argument of `clBuildProgram()` API call.
Exercise 7 (again)

• Additional Goal:
  – To inspect and verify the performances of Himeno Benchmark OpenCL using different manual and auto-prefetching levels.

• Procedure:
  – Play with different auto-prefetching levels.
  – Try to insert prefetch built-in into update.cl source code.
  – Compile and link (using `-DINTEL_MIC`).
  – Run on Intel platform.

• Expected output:
  – You should verify correctness and performances Intel MIC.
Conclusions

- Nine (9) things we missed during this presentation....
  - Vector data types, ...
  - Events, ...
  - C++ interface, Python interface, Fortran interface (via Ompss runtime)
  - Profiling
  - Debugging
  - Porting CUDA code to OpenCL
  - OpenCL 2.0 and beyond
Conclusions

And Six (6) things we won’t miss to say….

- OpenCL is an open standard, which is targeted for portability on heterogeneous devices (GPUs, AMD, Intel MIC, ...)
- OpenCL has widespread industrial support
- OpenCL is the only parallel programming standard that enables mixing task parallel and data parallel code in a single program while load balancing across ALL of the platform’s available resources
- To reach its full potential, however, OpenCL needs to deliver Performance portability.
- OpenCL gives users the chance to efficiently use native SIMD engines (like vector units) of CPUs, accelerators, ....
- OpenCL will help you use existing and future devices increasingly provide tremendous computing power at a reduced energy-requirement and price
References

Optimizing OpenCL applications on Intel Xeon Phi

OpenCL home page
http://www.khronos.org/opencl/

General Matrix Multiply Sample

Himeno Benchmark home page
http://accc.riken.jp/2444.htm

Simon McIntosh-Smith home page
http://www.cs.bris.ac.uk/~simonm/