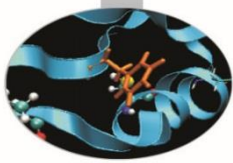
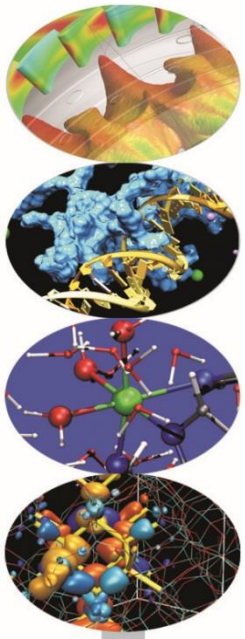
CINECA SCAI
SuperComputing Applications and Innovation



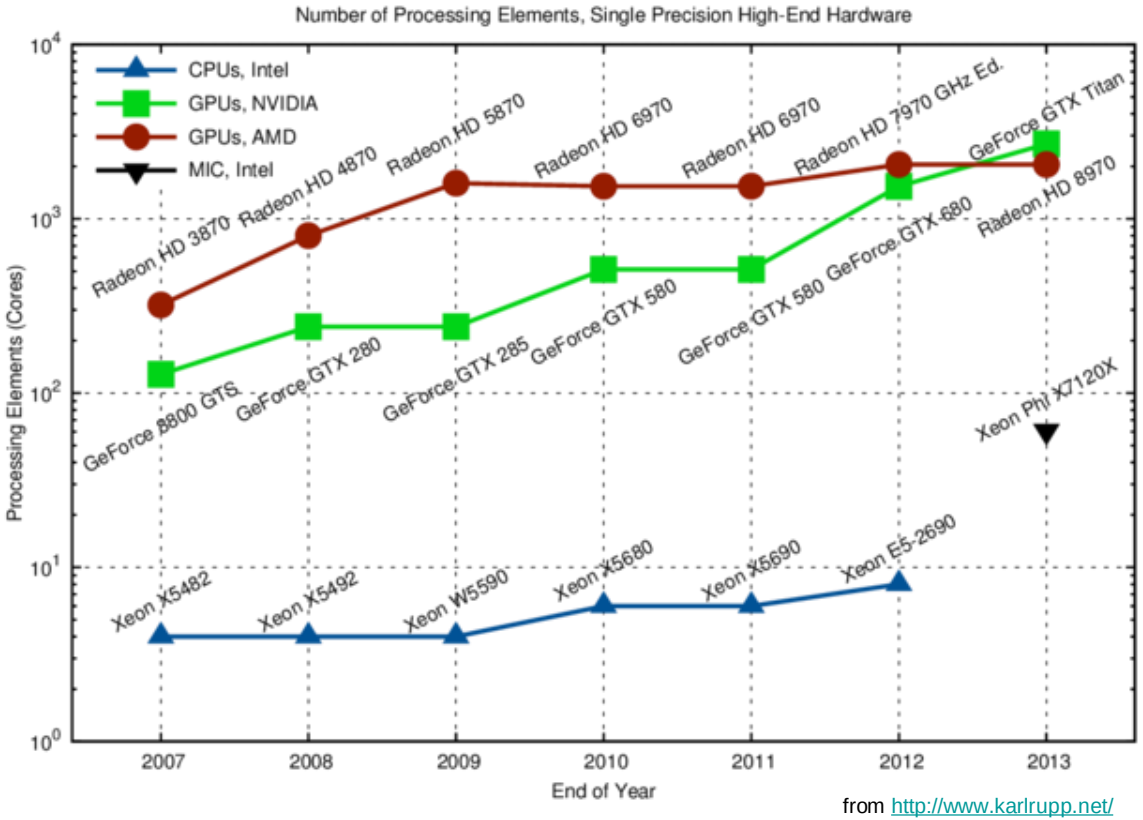
Introduction to OpenCL with examples

Piero Lanucara, *SCAI*

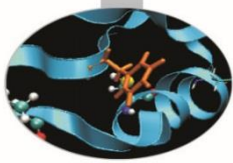
1 July 2015

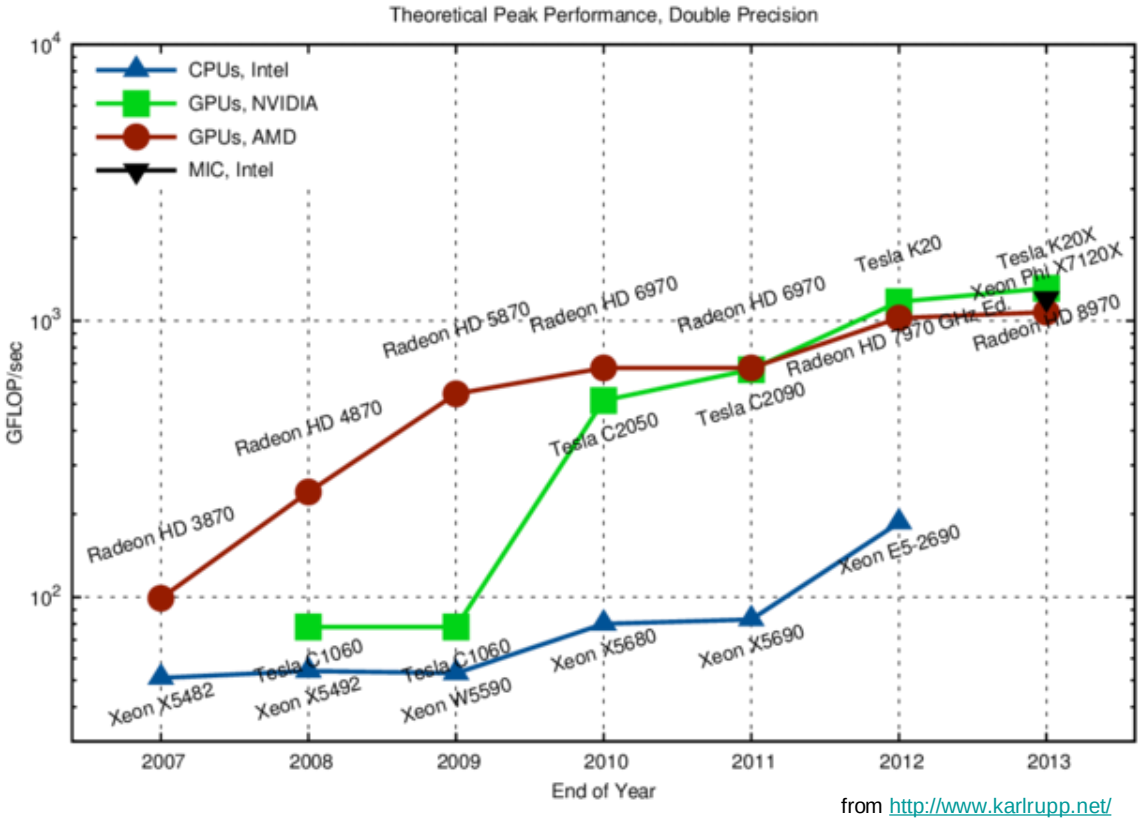


PRACE CINECA

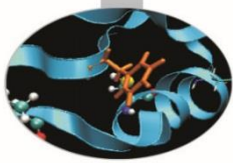


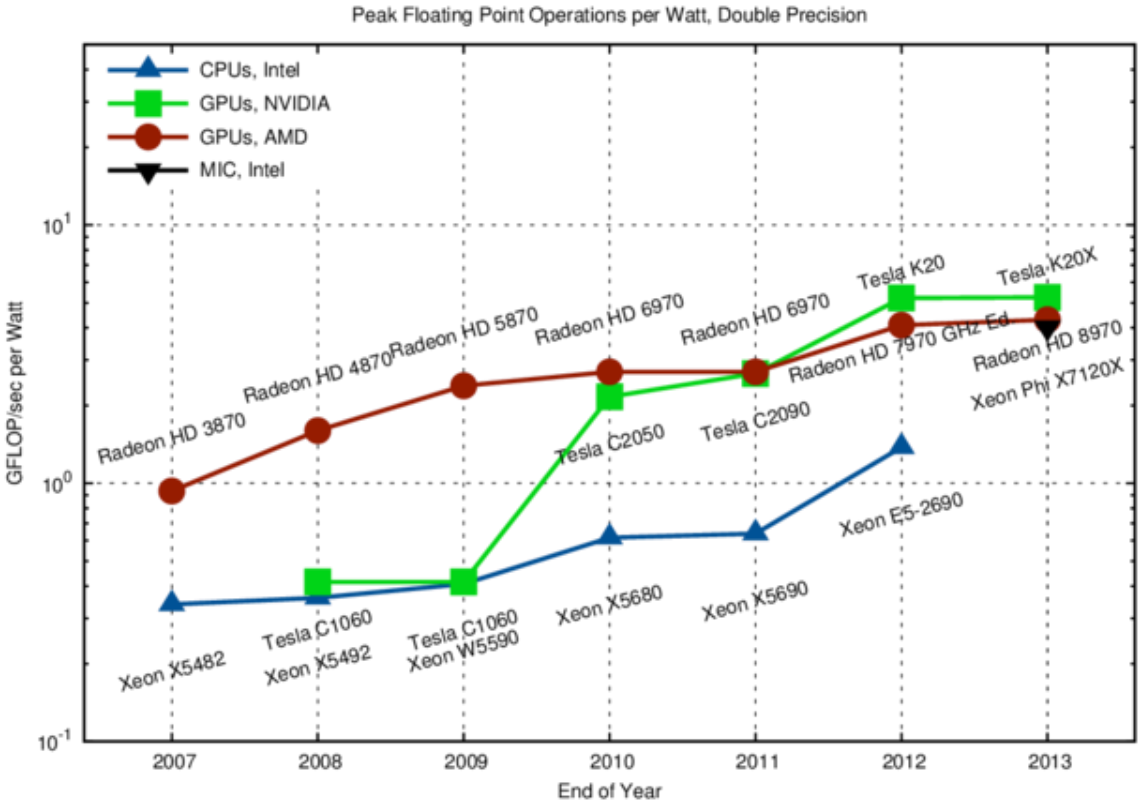
from http://www.karlrupp.net/



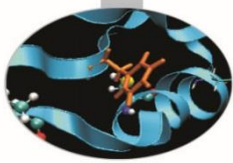


from http://www.karlrupp.net/





from http://www.karlrupp.net/



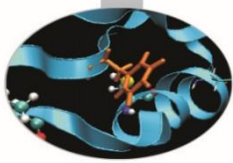
Heterogeneous High Performance Programming framework

- http://www.hpcwire.com/hpcwire/2012-02-28/opencl_gains_ground_on_cuda.html

HPC wire

"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.

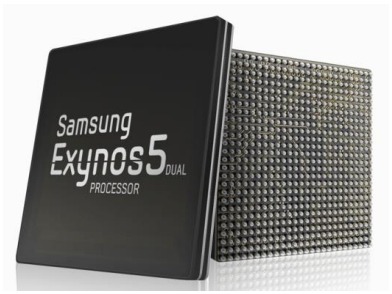
Both OpenCL and CUDA provide a general-purpose model for data parallelism as well as low-level access to hardware, but only OpenCL provides an open, industry-standard framework. As such, it has garnered support from nearly all processor manufacturers including AMD, Intel, and NVIDIA, as well as others 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?



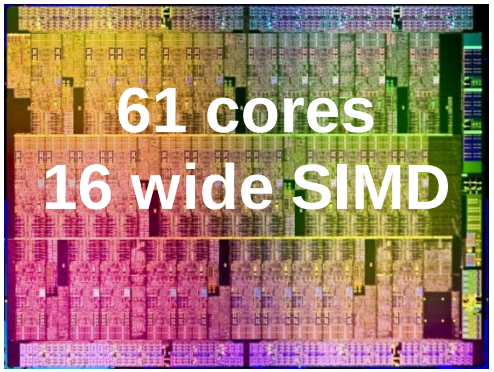
E.g. Samsung® Exynos 5:

- Dual core ARM A15 1.7GHz, Mali T604 GPU

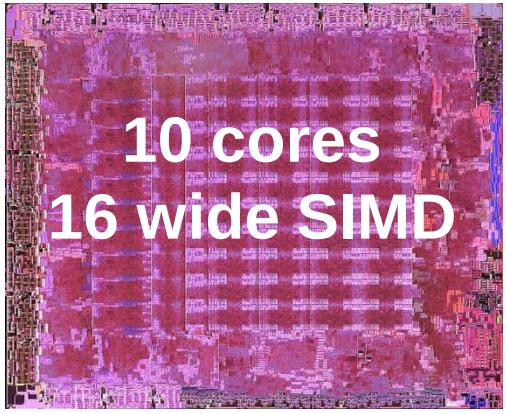
OpenCL lets Programmers write a single portable program that uses ALL resources in the heterogeneous platform

Microprocessor trends

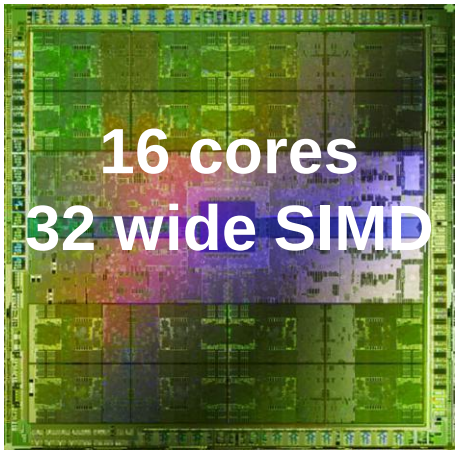
Individual processors have many (possibly heterogeneous) cores.



Intel® Xeon Phi™ coprocessor



ATI™ RV770

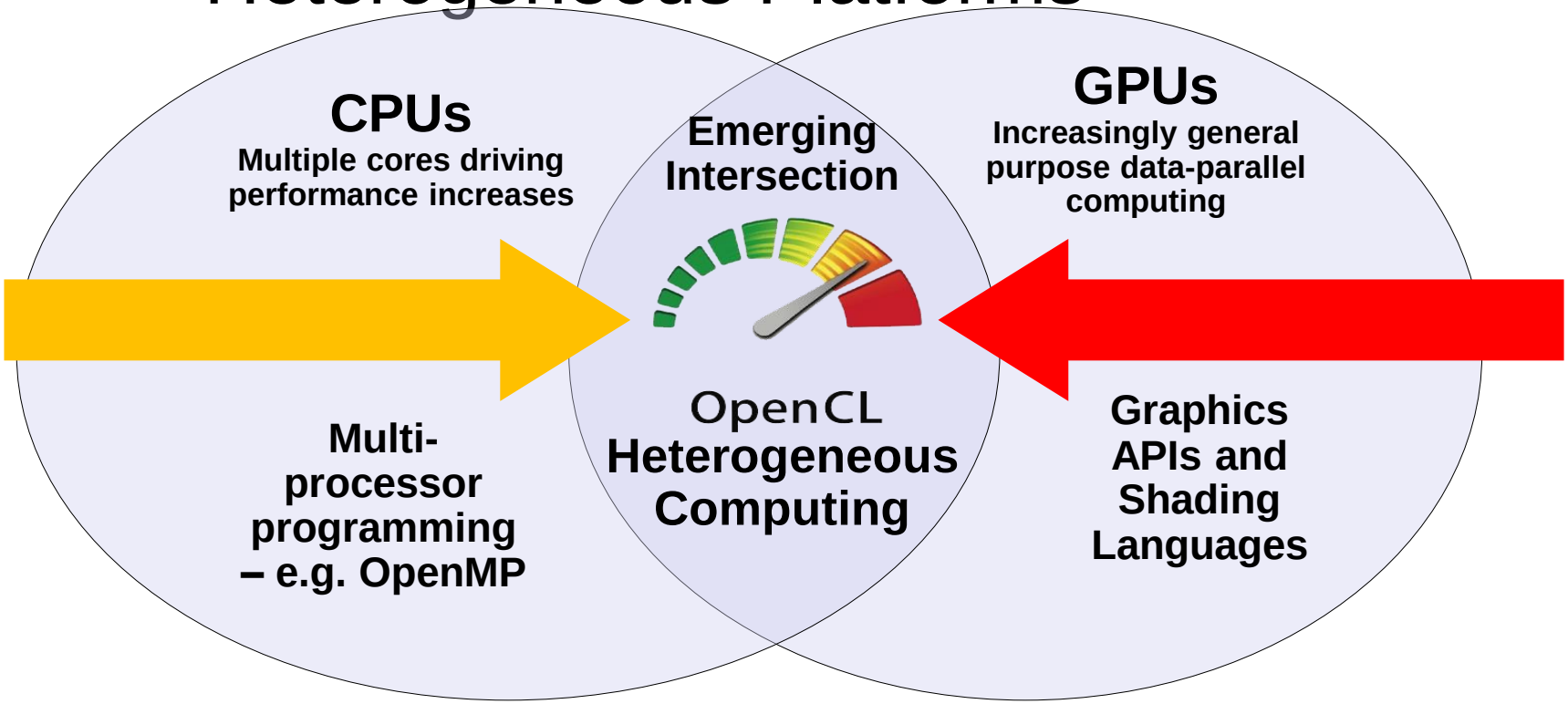


NVIDIA® Tesla® C2090

The Heterogeneous many-core challenge:

How are we to build a software ecosystem for the Heterogeneous many core platform?

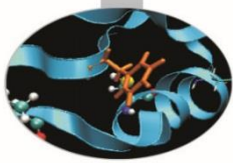
Industry Standards for Programming Heterogeneous Platforms



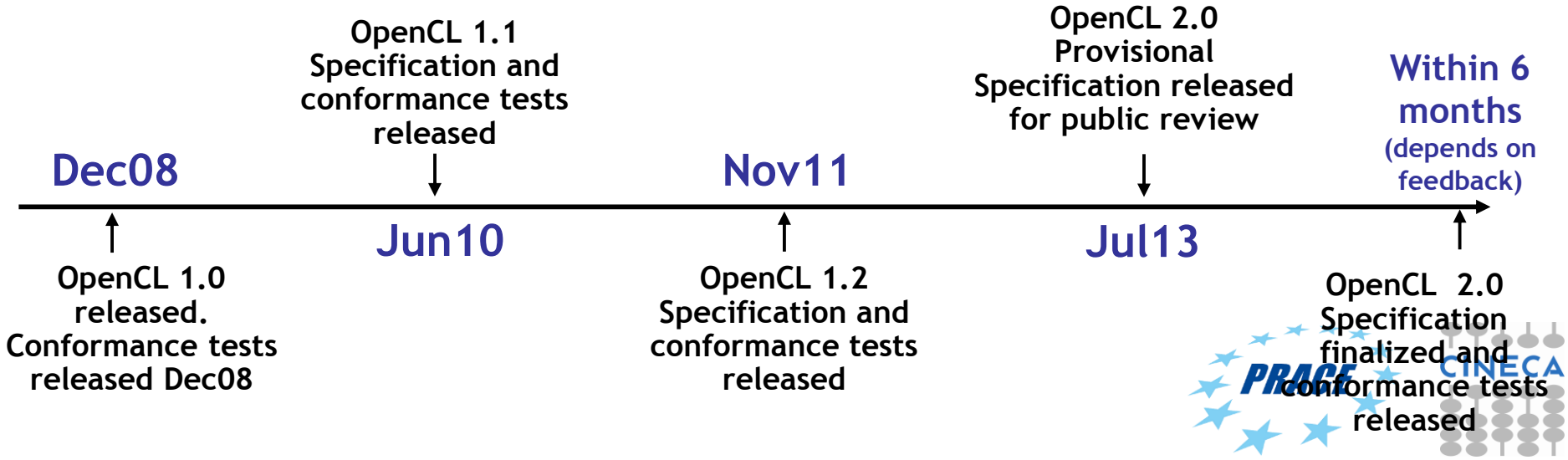
OpenCL – Open Computing Language

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

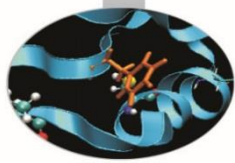
OpenCL Timeline



- 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



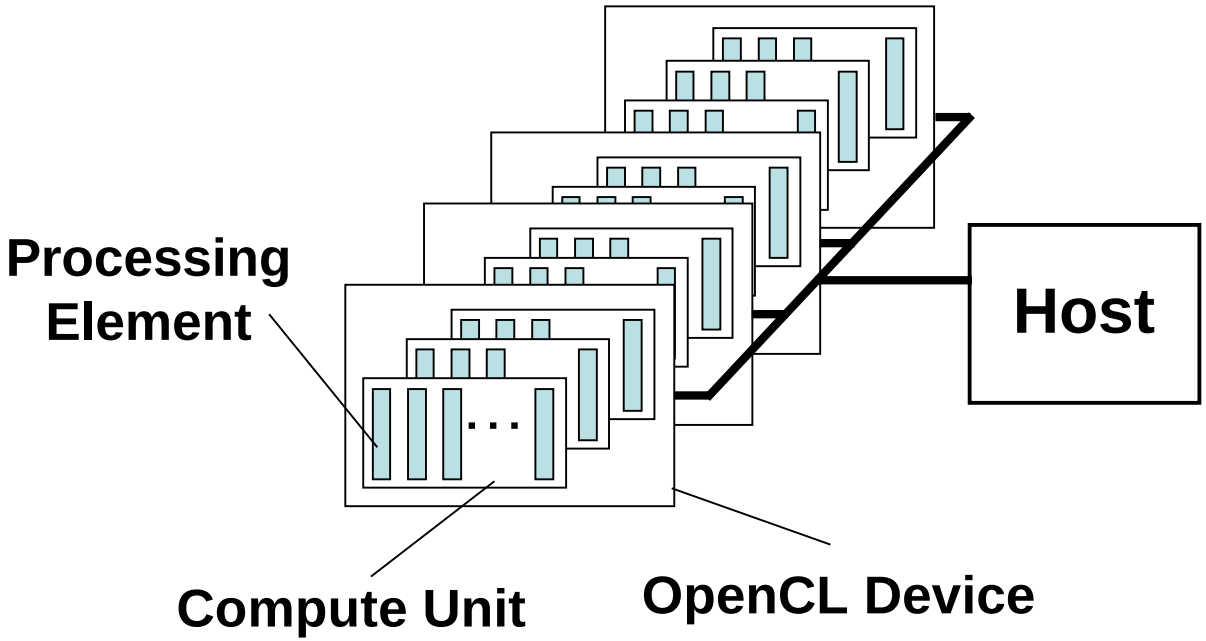
OpenCL Working Group within Khronos



- 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.

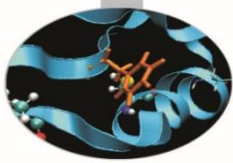
3DLABS SEMICONDUCTOR · ACTIVISION BLIZZARD · AMD · ARM · BROADCOM · EA · codeplay · ERICSSON · freescale semiconductor · HI CORP. · IBM · intel · Imagination TECHNOLOGIES · Los Alamos NATIONAL LABORATORY · MOTOROLA · movidia · NOKIA · NVIDIA · QNX SOFTWARE SYSTEMS · RAPIDMIND · SAMSUNG · Seaweed SYSTEMS · TAKUMI · TEXAS INSTRUMENTS · PRACE · CINECA · KHRONOS GROUP

OpenCL Platform Model



- 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:

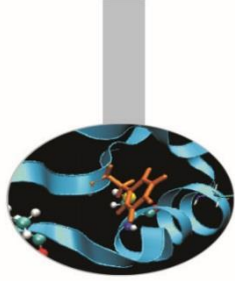
- Treated as one OpenCL device
  - One CU per core
  - 1 PE per CU, or if PEs mapped to SIMD lanes, *n* PEs per CU, where *n* matches the SIMD width
- Remember:
  - the CPU will also have to be its own host!

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

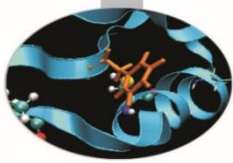
Traditional loops

```
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];
}
```

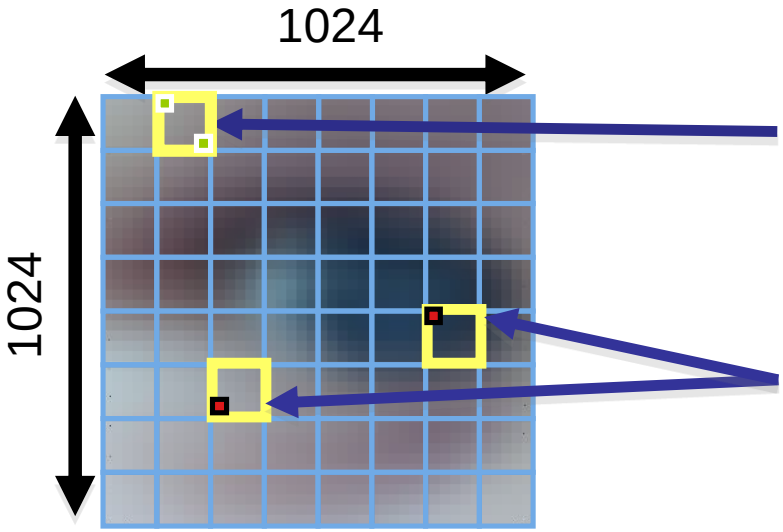
Data Parallel OpenCL

```
__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



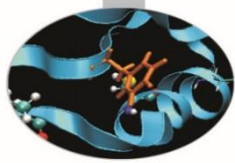
- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (**work-group**, executes together)



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

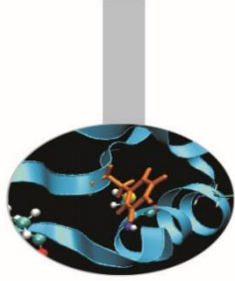
Cannot synchronize between work-groups within a kernel

- Choose the dimensions that are “best” for your algorithm



OpenCL N Dimensional Range (NDRange)

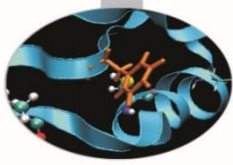
- The problem we want to compute should have some **dimensionality**;
  - For example, compute a kernel on all points in a cube
- When we execute the kernel we specify **up to 3 dimensions**
- We also **specify the total problem size** in each dimension – this is called the **global** size
- We associate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



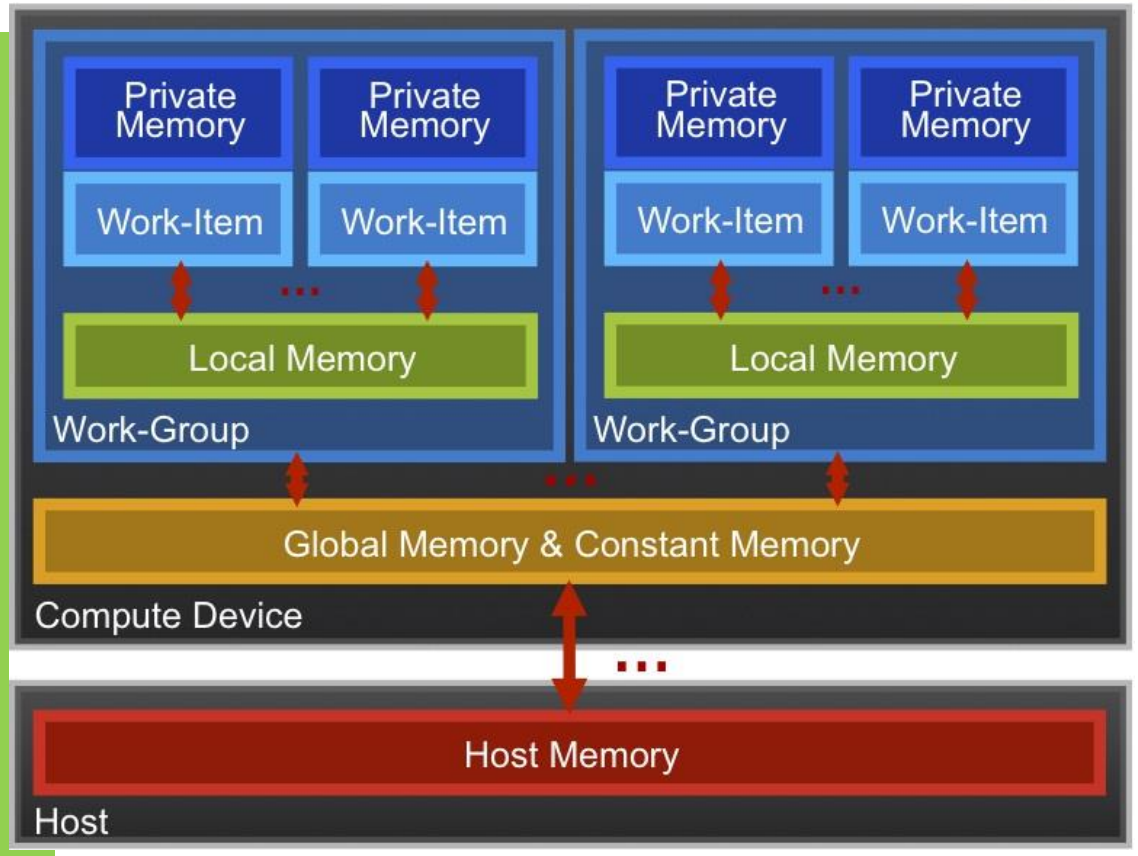
OpenCL N Dimensional Range (NDRange)

- 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



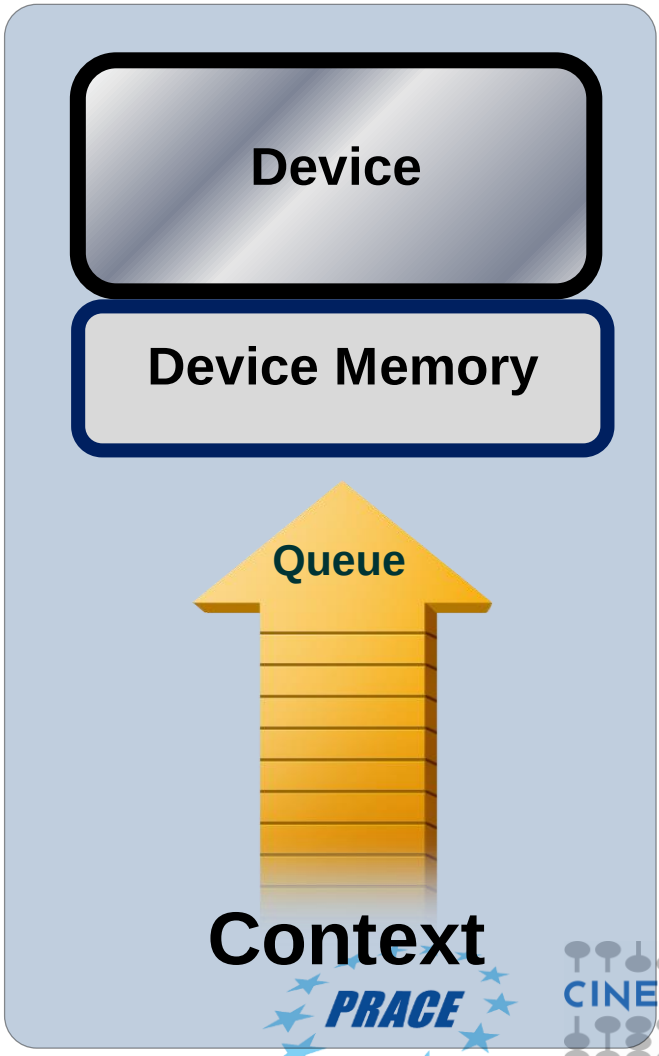
- ***Private Memory***
  - Per work-item
- ***Local Memory***
  - Shared within a work-group
- ***Global Memory /Constant Memory***
  - Visible to all work-groups
- ***Host memory***
  - On the CPU



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

Context and Command-Queues

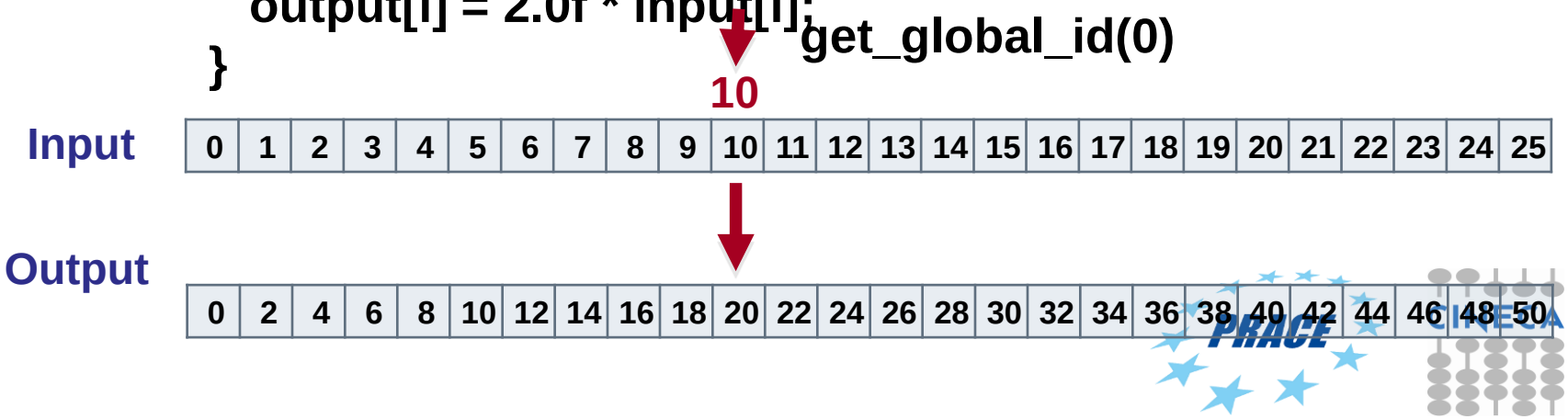
- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization, and memory transfer operations) are submitted through a ***command-queue***.
- Each ***command-queue*** points to a single device within a context.



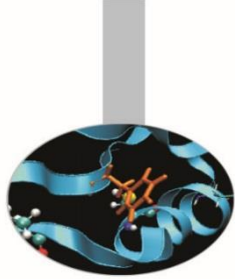
Execution model (kernels)

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

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

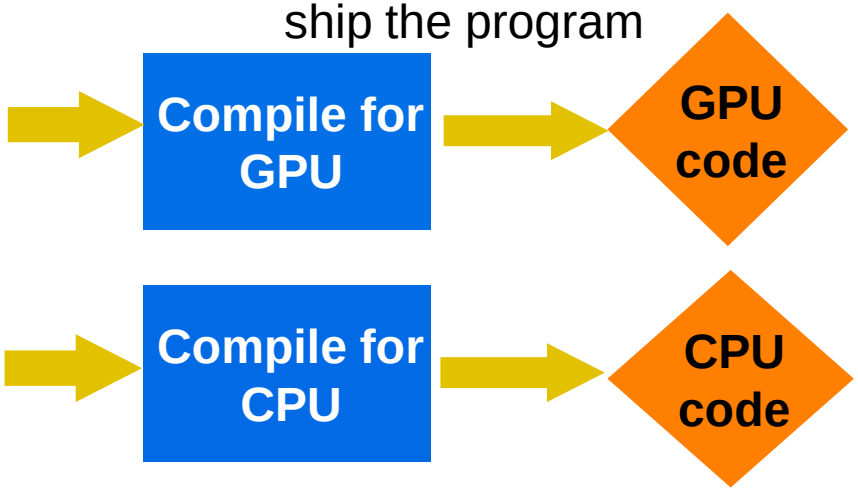


Building Program Objects

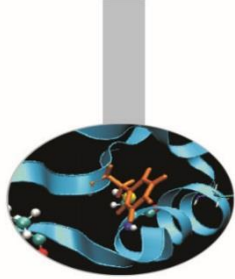


- The program object encapsulates:
  - A context
  - The program kernel source or binary
  - List of target devices and build options
- The C API build process to create a program object:
  - clCreateProgramWithSource()
  - clCreateProgramWithBinary()

OpenCL uses runtime compilation … because in general you don't know the details of the target device when you ship the program




```
__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);
}
```

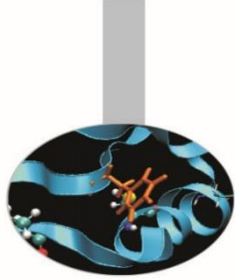


Example: vector addition

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

C[i] = A[i] + B[i] for i=0 to N-1

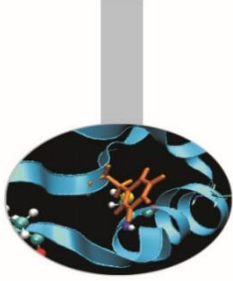
- For the OpenCL solution, there are two parts
  - Kernel code
  - Host code



Vector Addition - Kernel

```
__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];
}
```

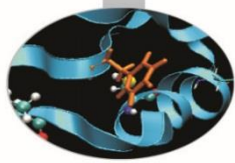
Vector Addition – Host



- 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 he 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:

```
err = clGetPlatformIDs(1, &firstPlatformId,
                                    &numPlatforms);
```

- Use the first CPU device the platform provides:

```
err = clGetDeviceIDs(firstPlatformId,
            CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);
```

- Create a simple context with a single device:

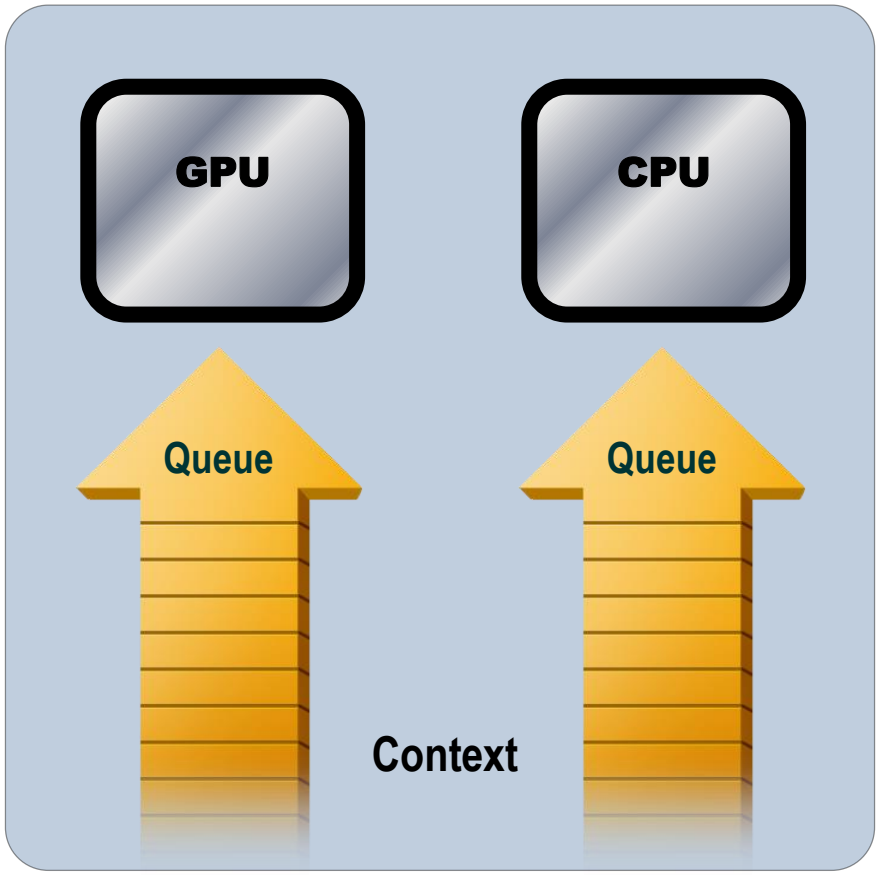
```
context = clCreateContext(firstPlatformId, 1,
                    &device_id, NULL, NULL, &err);
```

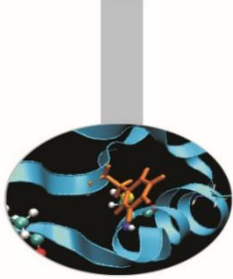
- Create a simple command-queue to feed our device:

```
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

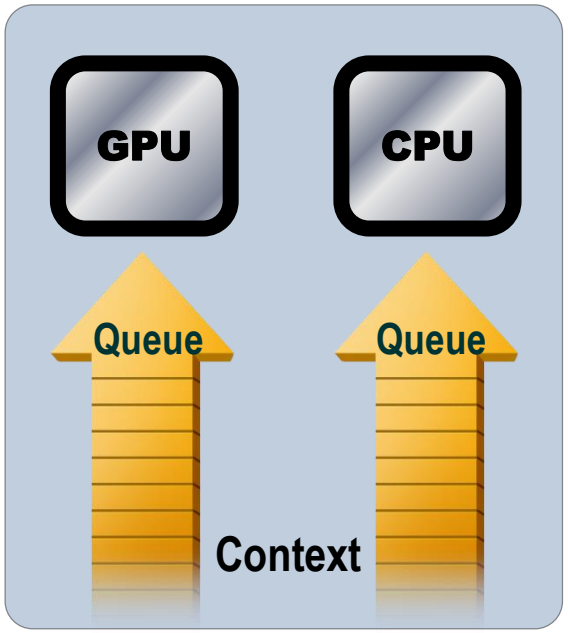


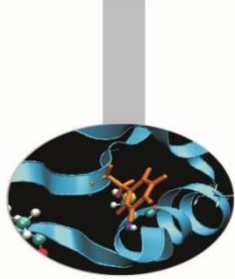


Command-Queue execution details

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

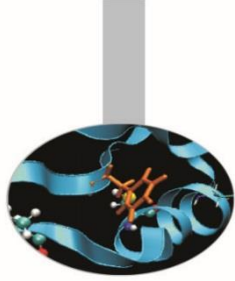
- Define source code for the kernel-program as a string literal (great for toy programs) or read from a file (for real applications).

- 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

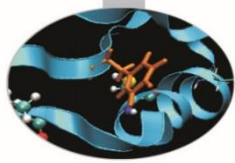
- Fetch and print error messages:

```
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:

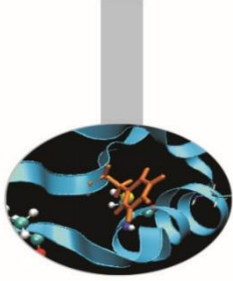
```
float h_a[LENGTH], h_b[LENGTH], h_c[LENGTH];
for (i = 0; i < length; i++) {
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
}
```

Memory Objects:

- A handle to a reference-counted region of global memory.

- Define OpenCL memory objects:

```
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);
```



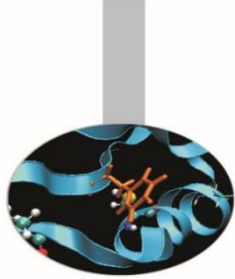
Creating and manipulating buffers

- 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);
```



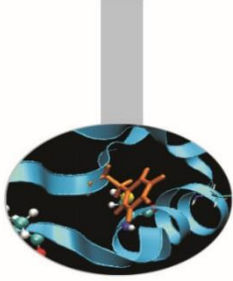
Creating and manipulating buffers

- 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

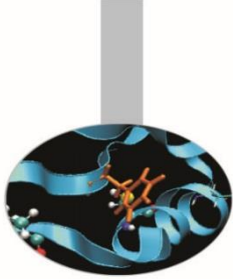
- Create kernel object from the kernel function “vadd”:

```
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



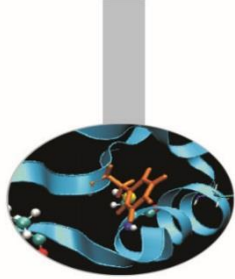
- Write Buffers from host into global memory (as non-blocking operations):

```
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);
```

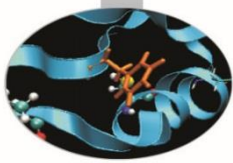
5. Enqueue commands



- 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.

```
err = clEnqueueReadBuffer(commands, d_c, CL_TRUE,
        sizeof(float)*count, h_c, 0, NULL, NULL);
```

Vector Addition – Host Program



```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,
                  CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetC                                                  cb);

cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);

// allocate the buffer memory objects
memobj
    CL_I                                                ILL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcb, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                  sizeof(cl_float)*n, NULL, NULL);

// create the
program = c
                  &program_source, NULL, NULL);
```

Define platform and queues

Define memory objects

Create the program

```
// build the prog
err = clBuildPro                                        ULL,NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0]
err |= clS                                              ],
                  sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                  sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute ker
err = clEnque                                           el, 1, NULL,
                  global_work_size, NULL,0,NULL,NULL);

// read output array
err = clEn
                  n*sizeof(cl_float), dst,
                  0, NULL, NULL);
```

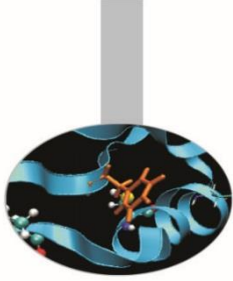
Build the program

Create and setup kernel

Execute the kernel

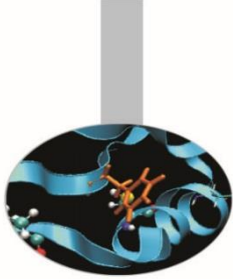
Read results on the host

It's complicated, but most of this is "boilerplate" and not as bad as it looks.



OpenCL C for Compute Kernels

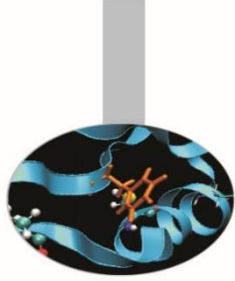
- Derived from **ISO C99**
  - A few ***restrictions***: no recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported (#include etc.)
- Built-in data types
  - Scalar and vector data types, pointers
  - Data-type conversion functions:
    - 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

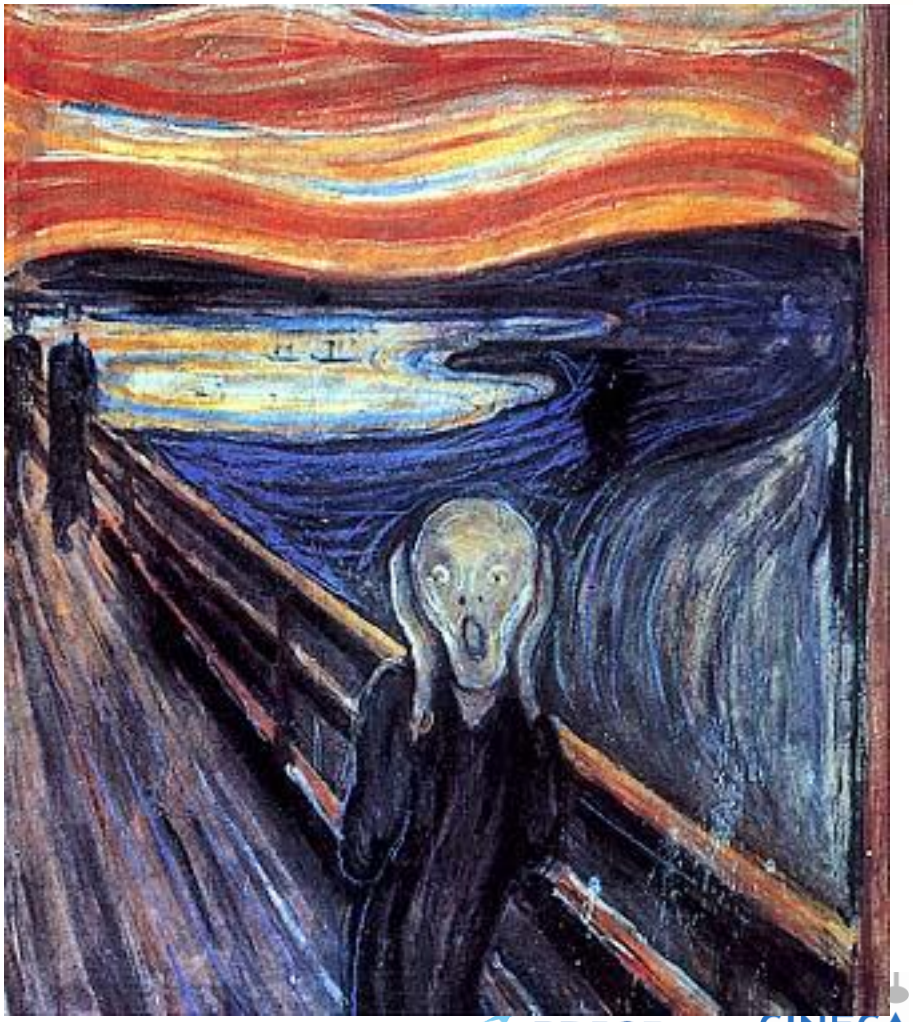
OpenCL C Language Highlights

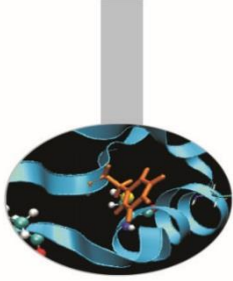


- Function qualifiers
  - **__kernel** qualifier declares a function as a kernel
    - I.e. makes it visible to host code so it can be enqueued
  - Kernels can call other kernel-side functions
- Address space qualifiers
  - **__global, __local, __constant, __private**
  - Pointer kernel arguments must 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.



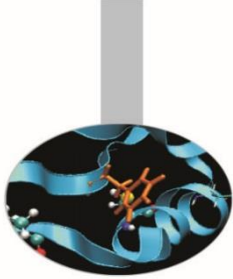


The C++ Interface

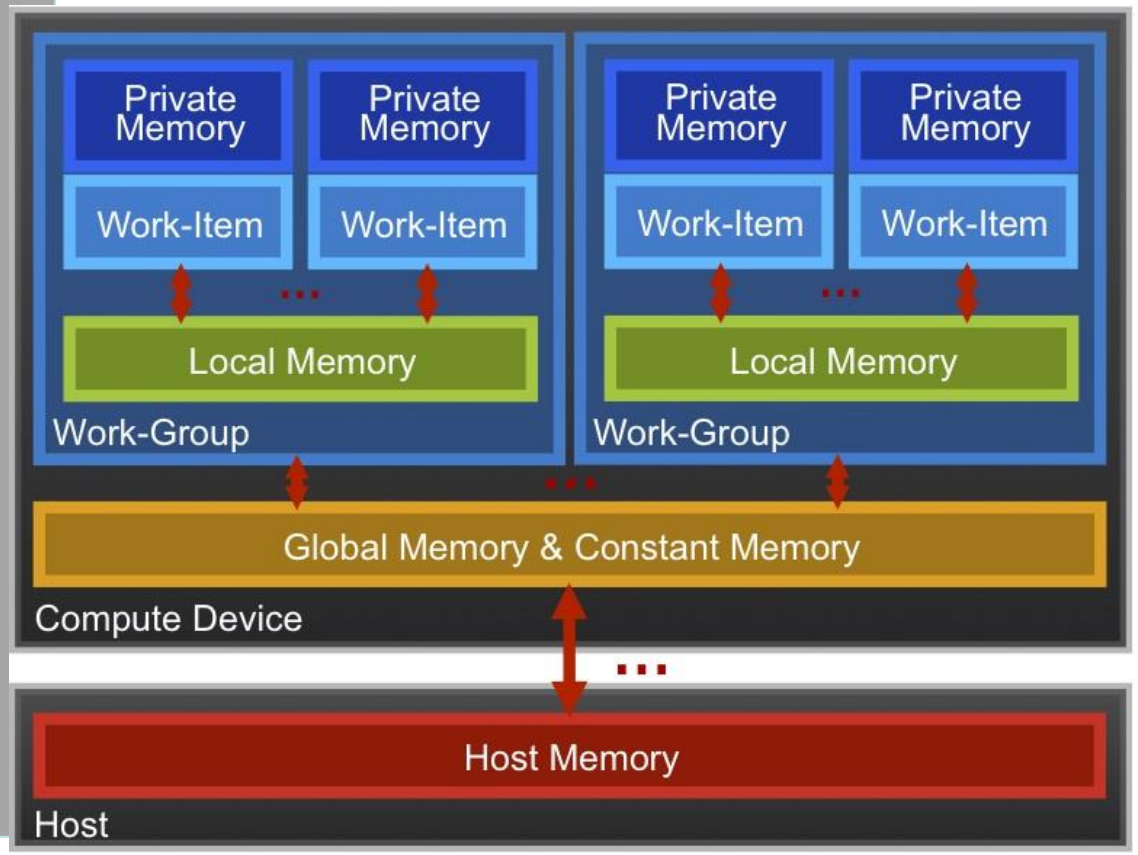
- Khronos has defined a common C++ header file containing a high level interface to OpenCL, cl.hpp
- This interface is dramatically easier to work with[1]
- Key features:
  - Uses common defaults for the platform and command-queue, saving the programmer from extra coding for the most common use cases
  - 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



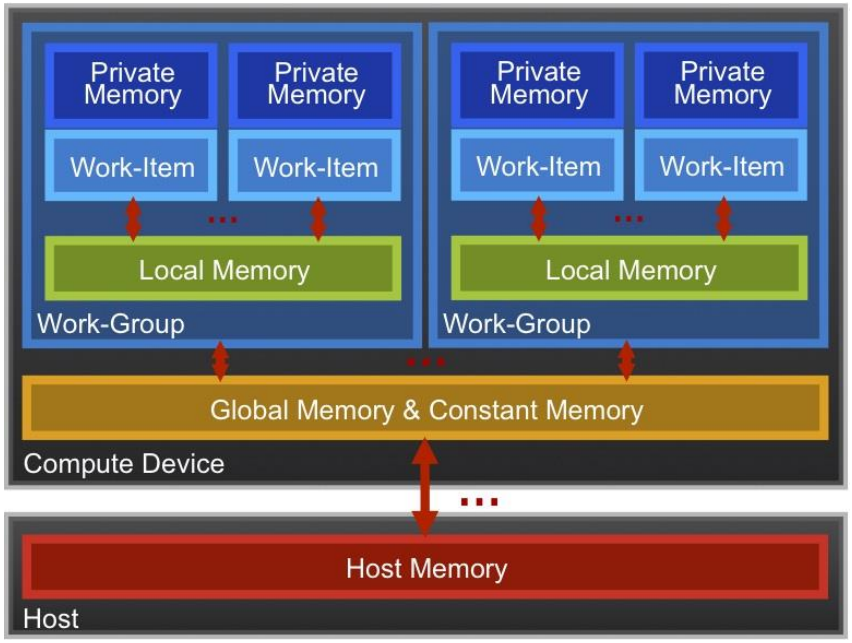
- Private Memory
  - Per work-item
- Local Memory
  - Shared within a work-group
- Global/Constant Memory
  - Visible to all work-groups
- Host memory
  - On the CPU



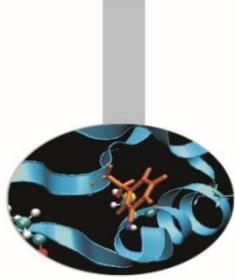
Memory management is **explicit**:
You are responsible for moving data from
host → global → local *and* back

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

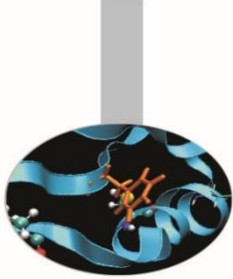


Private Memory

- 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

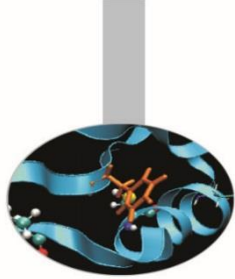
* Occupancy on a GPU



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

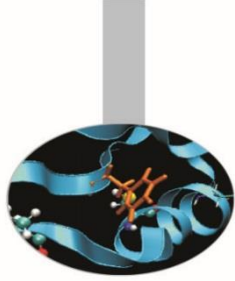
* Typical figures for a 2013 GPU

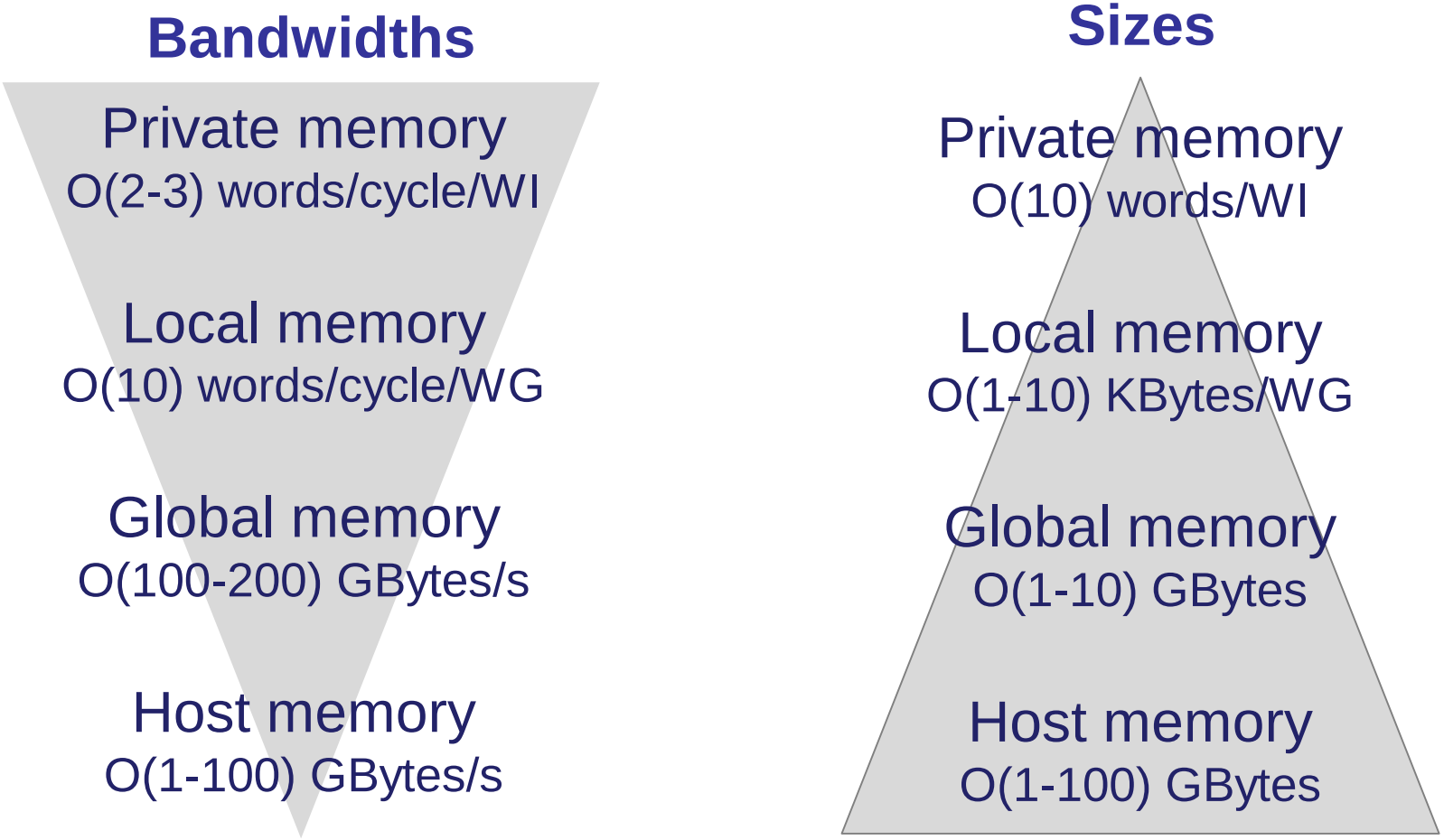


Local Memory

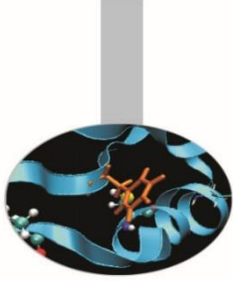
- Local Memory doesn't always help…
  - CPUs don't have special hardware for it
  - This can mean excessive use of Local Memory might slow down kernels on CPUs
  - GPUs now have effective on-chip caches which can provide much of the benefit of Local Memory but without programmer intervention
  - So,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

The Memory Hierarchy





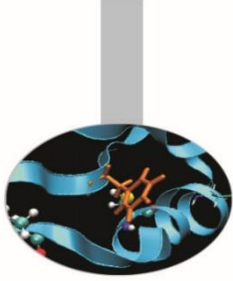
Speeds and feeds approx. for a high-end discrete GPU, circa 2011



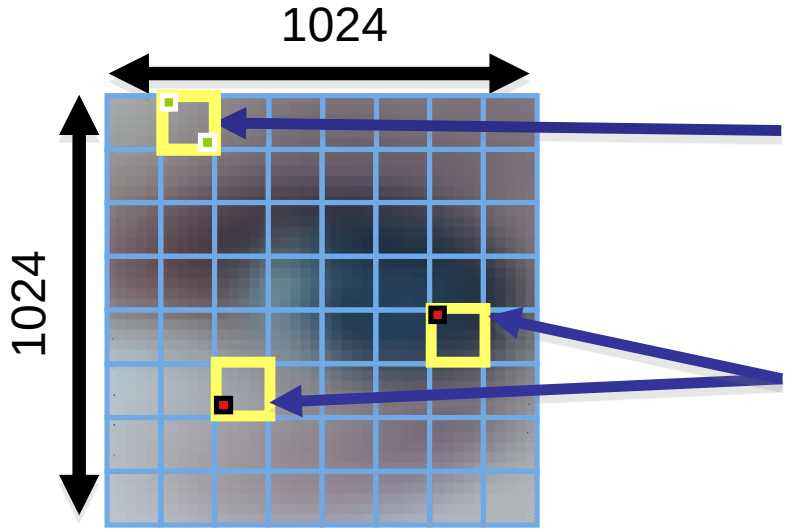
Memory Consistency

- 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)

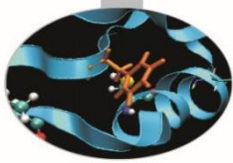


Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within a kernel

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.

Work-Item Synchronization



Ensure correct order of memory operations to local memory (with flushes or queuing a memory fence)l or global

- Within a work-group

  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:

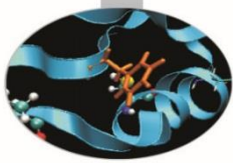
    - ALL work-items in the work-group, OR
    - NO work-item in the work-group

- 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



- Targets a broader range of CPU-like and GPU-like devices than CUDA
  - Targets ... endors
  - Manufacturers may not be
- OpenCL ... with much
- A single ... t achieve peak performance ... types

Performance????

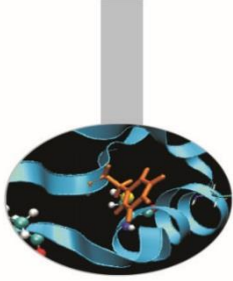
Portable performance in OpenCL

- 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

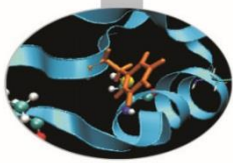
1070 GFLOPs peak



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:

  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	…

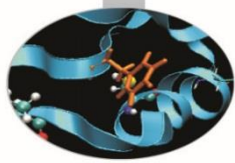
Adjacent work-items like to access adjacent memory

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

x	y	z	a	…	x	y	z	a	…	x	y	z	a	…	x	y	z	a	…

Individual work-items like to access adjacent memory

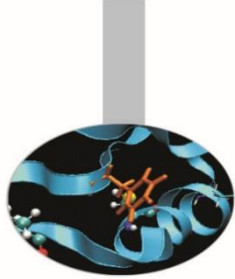
Advice for performance portability



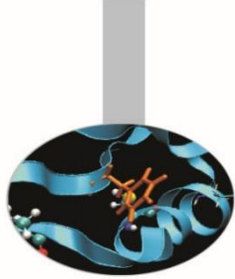
- Optimal Work-Group sizes will differ between devices
  - E.g. CPUs tend to prefer 1 Work-Item per Work-Group, while GPUs prefer lots of Work-Items per Work-Group (usually a multiple of the number of PEs per Compute Unit, i.e. 32, 64 etc.)
- From OpenCL v1.1 you can discover the preferred Work-Group 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

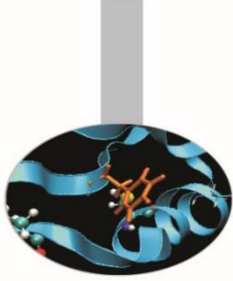


- Tiling size (work-group sizes, dimensionality etc.)
  - For block-based algorithms (e.g. matrix multiplication)
  - Different devices might run faster on different block sizes
- Data layout
  - Array of Structures or Structure of Arrays (AoS vs. SoA)
  - Column or Row major
- Caching and prefetching
  - 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)



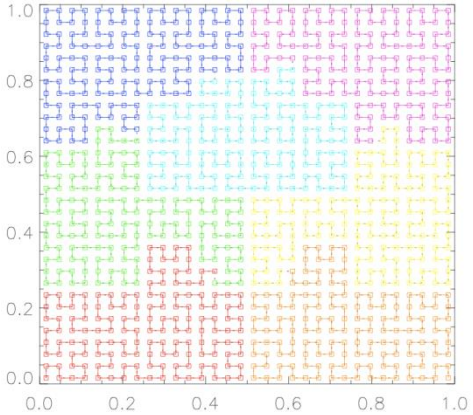
Auto tuning

- 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.



How much fast? The Hydro benchmark

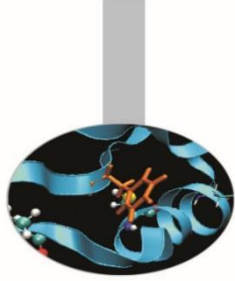
Hydro is a simplified version of RAMSES (CEA, France astrophysics code to study large scale structure and galaxy formation)



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

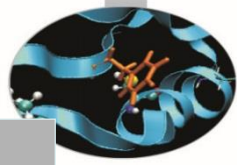
The Hydro benchmark



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!)

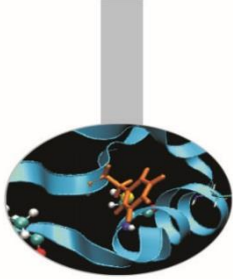
More than 16 Intel Xeon SandyBridge cores are needed to compare OpenCL 1 K20 device

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

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

OpenAcc run it fails using Pgi compiler

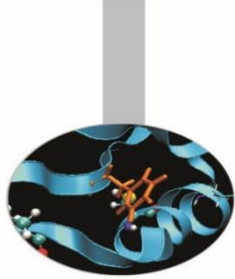
Hydro OpenCL scaling



performances are good. Scalability is limited by domain size

OpenCL+MPI run, varying the number of NVIDIA Tesla K20 device,4091x4091 domain,100 iterations

Number of K20 devices	Elapsed time (sec.) without initialization	Speed-Up
1	42.0	1.0
2	23.5	1.7
4	12.2	3.4
8	8.56	4.9
16	5.70	7.3



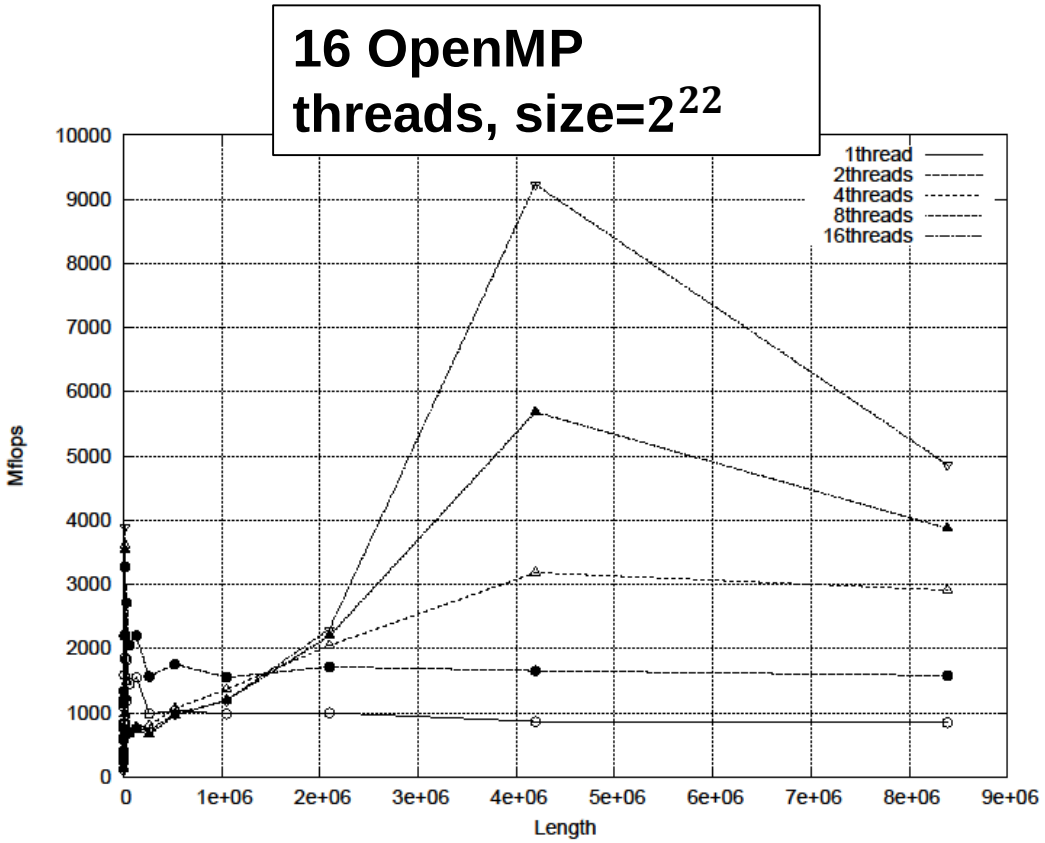
How much fast? The EuroBen Benchmark

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

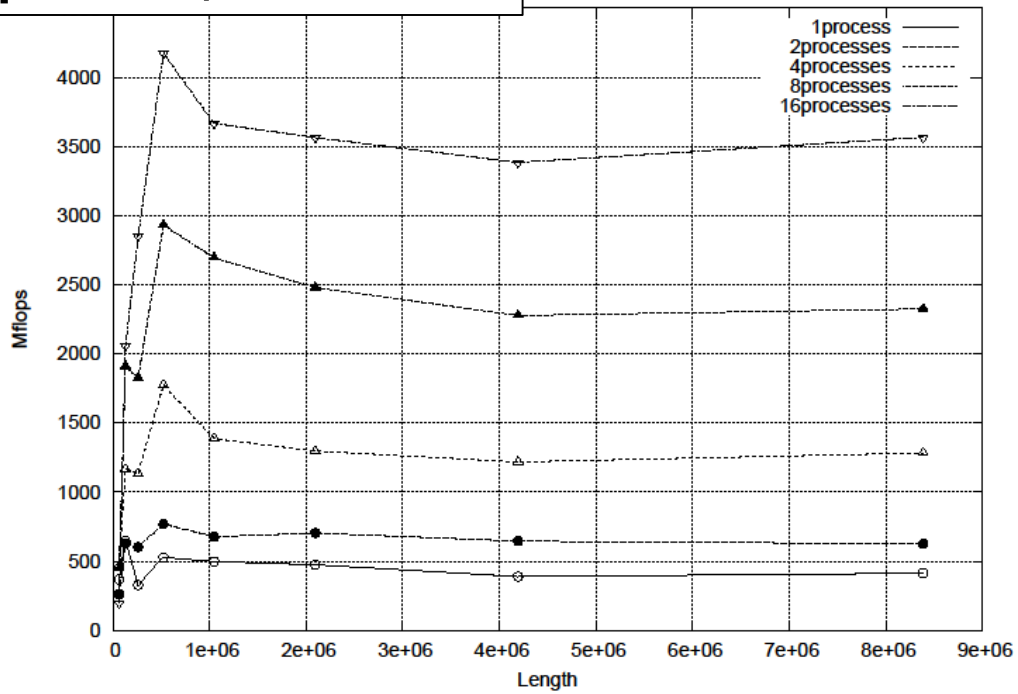
Host: Intel Xeon SandyBridge cores

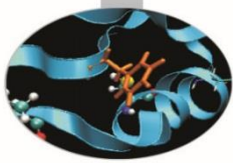


MOD2F benchmark

16 MPI process, size=2^{19}

Host: Intel Xeon SandyBridge cores

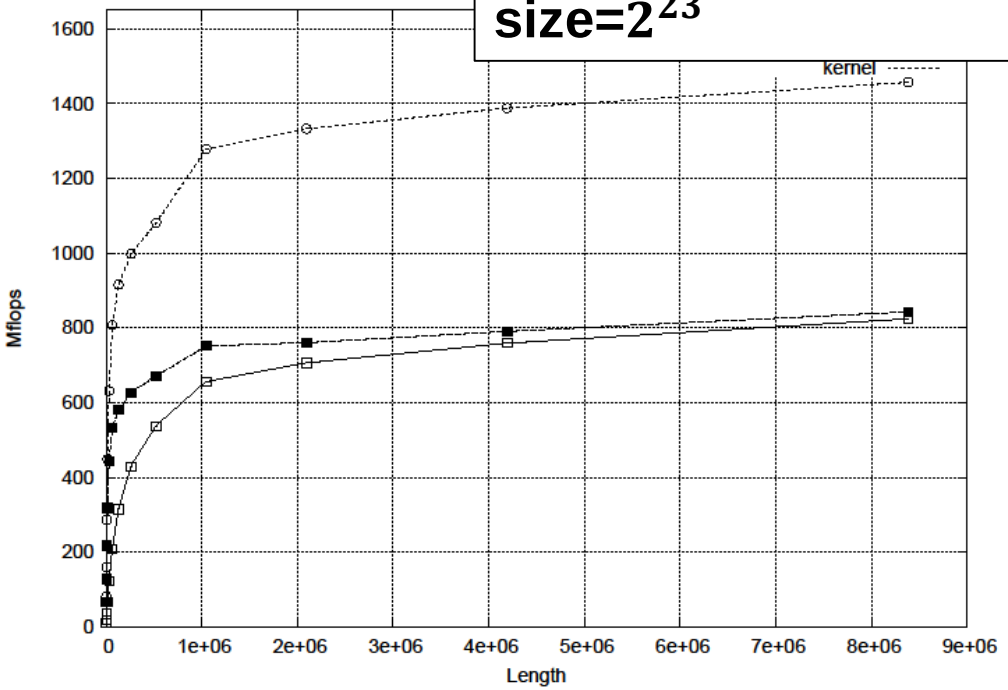




MOD2F benchmark

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

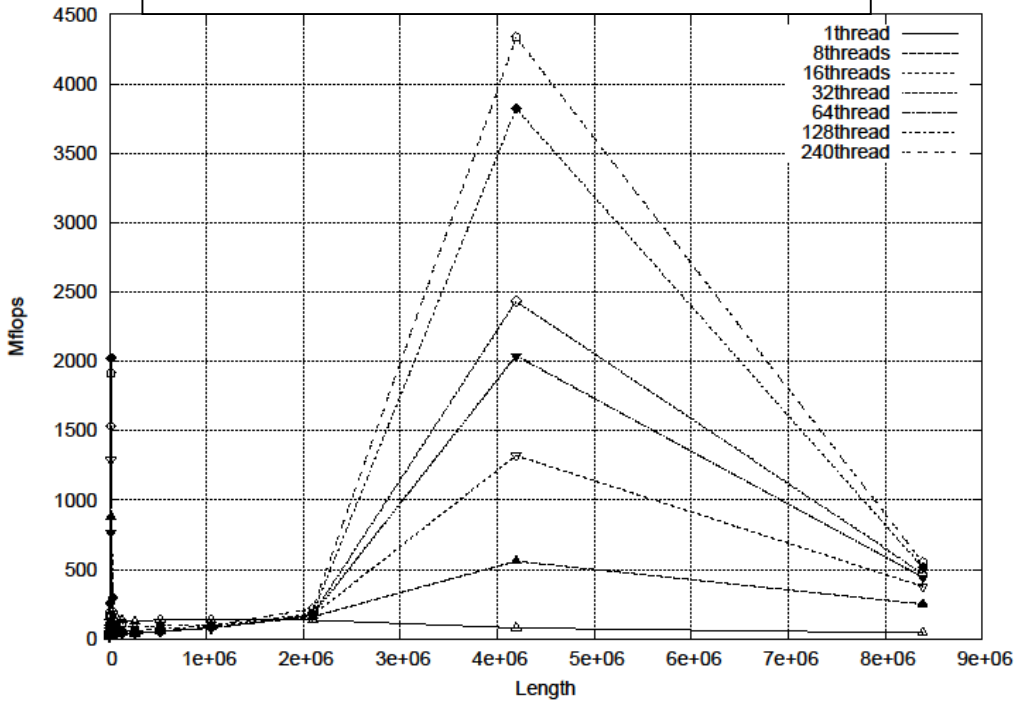
Host: Intel Xeon SandyBridge cores



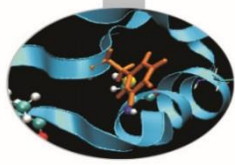
MOD2F benchmark

Native: Intel MIC, up to 240 hw threads

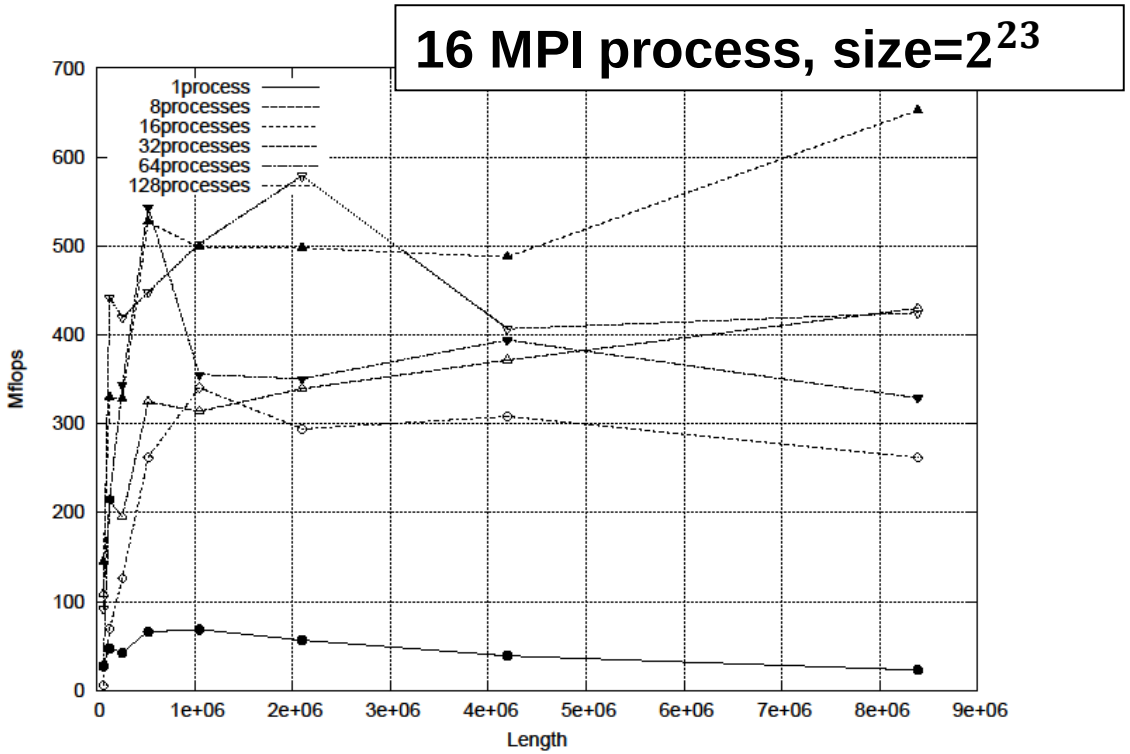
240 OpenMP threads, size=2^{22}



MOD2F benchmark



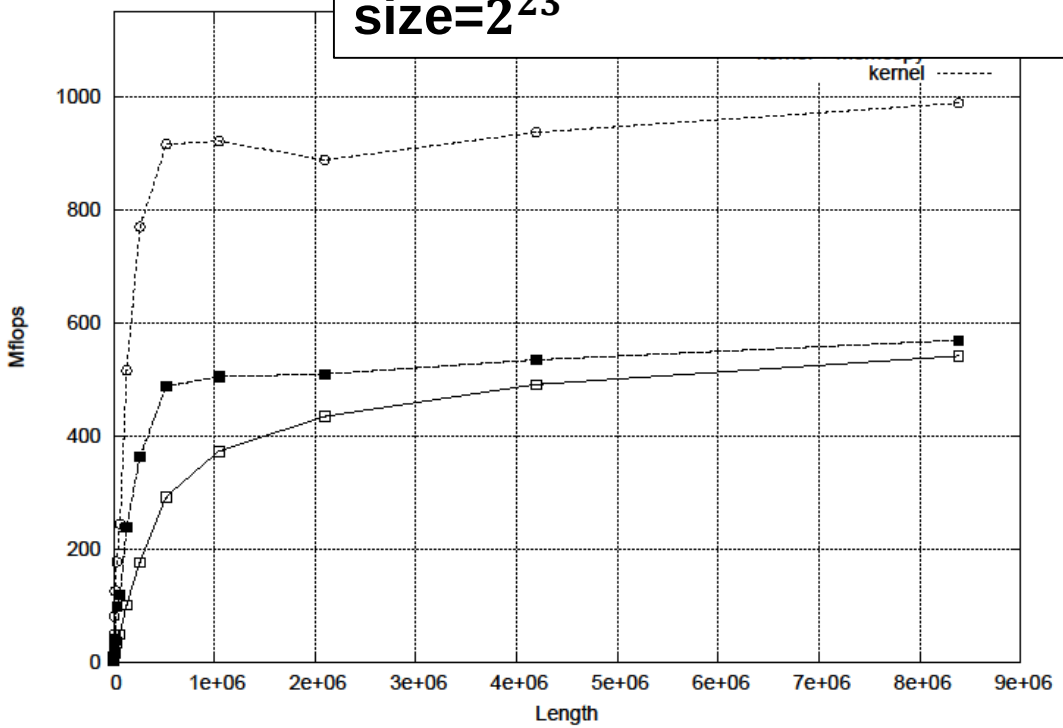
Native: Intel MIC, up to 240 hw threads



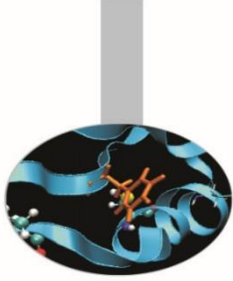
MOD2F benchmark

Native: Intel MIC, up to 240 hw threads

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

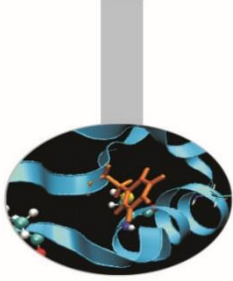


CINECA SCAI
SuperComputing Applications and Innovation



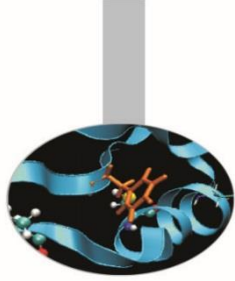
Porting CUDA to OpenCL

PRACE CINECA



- If you have CUDA code, you've already done the hard work!
  - I.e. working out how to split up the problem to run effectively on a many-core device
- Switching between CUDA and OpenCL is mainly changing the host code syntax
  - Apart from indexing and naming conventions in the kernel code (simple to change!)

Allocating and copying memory



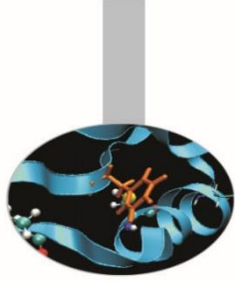
	CUDA C	OpenCL C

Allocate	`float* d_x;`
	`cudaMalloc(&d_x, sizeof(float)*size);`
		`cl_mem d_x =`
		`  clCreateBuffer(context,`
		`  CL_MEM_READ_WRITE,`
		`  sizeof(float)*size,`
		`  NULL, NULL);`

Host to Device	`cudaMemcpy(d_x, h_x,`
	`  sizeof(float)*size,`
	`  cudaMemcpyHostToDevice);`
		`clEnqueueWriteBuffer(queue, d_x,`
		`  CL_TRUE, 0,`
		`  sizeof(float)*size,`
		`  h_x, 0, NULL, NULL);`

Device to Host	`cudaMemcpy(h_x, d_x,`
	`  sizeof(float)*size,`
	`  cudaMemcpyDeviceToHost);`
		`clEnqueueReadBuffer(queue, d_x,`
		`  CL_TRUE, 0,`
		`  sizeof(float)*size,`
		`  h_x, 0, NULL, NULL);`

Allocating and copying memory



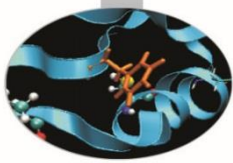
	CUDA C	OpenCL C++

Allocate	`float* d_x;`
	`cudaMalloc(&d_x,`
	`  sizeof(float)*size);`
		`cl::Buffer`
		`  d_x(begin(h_x), end(h_x), true);`

Host to Device	`cudaMemcpy(d_x, h_x,`
	`  sizeof(float)*size,`
	`  cudaMemcpyHostToDevice);`
		`cl::copy(begin(h_x), end(h_x),`
		`  d_x);`

Device to Host	`cudaMemcpy(h_x, d_x,`
	`  sizeof(float)*size,`
	`  cudaMemcpyDeviceToHost);`
		`cl::copy(d_x,`
		`  begin(h_x), end(h_x));`

Declaring dynamic local/shared memory



CUDA C

1. Define an array in the kernel source as extern

```
__shared__ int array[];
```

2. When executing the kernel, specify the third parameter as size in bytes of shared memory

```
func<<<num_blocks,
 num_threads_per_block,
 shared_mem_size>>>(args);
```

OpenCL C++

1. Have the kernel accept a local array as an argument

```
__kernel void func(
    __local int *array)
 {}
```

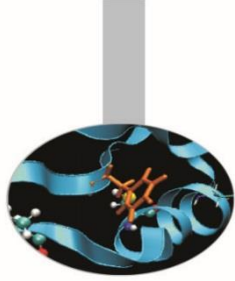
2. Define a local memory kernel kernel argument of the right size

```
cl::LocalSpaceArg localmem =
   cl::Local(shared_mem_size);
```

3. Pass the argument to the kernel invocation

```
func(EnqueueArgs(…),localmem);
```

Declaring dynamic local/shared memory



CUDA C

1. Define an array in the kernel source as extern

```
__shared__ int array[];
```

2. When executing the kernel, specify the third parameter as size in bytes of shared memory

```
func<<<num_blocks,
 num_threads_per_block,
 shared_mem_size>>>(args);
```

OpenCL C

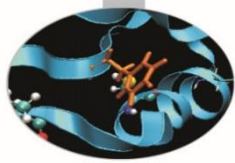
1. Have the kernel accept a local array as an argument

```
__kernel void func(
    __local int *array) {}
```

2. Specify the size by setting the kernel argument

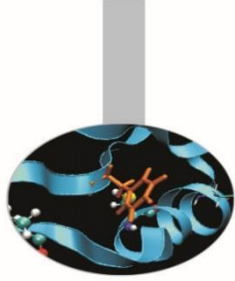
```
clSetKernelArg(kernel, 0,
 sizeof(int)*num_elements,
 NULL);
```

Dividing up the work



- To enqueue the kernel
  - CUDA – specify the number of thread blocks and threads per block
  - OpenCL – specify the problem size and (optionally) number of work-items per work-group

Enqueue a kernel (C)



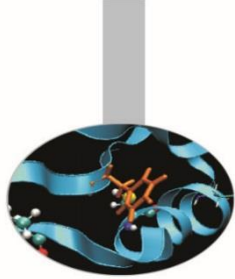
CUDA C

```
dim3 threads_per_block(30,20);

dim3 num_blocks(10,10);

kernel<<<num_blocks,
        threads_per_block>>>();
```

OpenCL C

```
const size_t global[2] =
        {300, 200};

const size_t local[2] =
        {30, 20};

clEnqueueNDRangeKernel(
    queue, &kernel,
    2, 0, &global, &local,
    0, NULL, NULL);
```

Enqueue a kernel (C++)



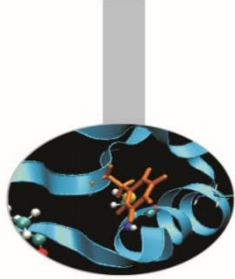
CUDA C

```
dim3 threads_per_block(30,20);

dim3 num_blocks(10,10);

kernel<<<num_blocks,
  threads_per_block>>>(…);
```

OpenCL C++

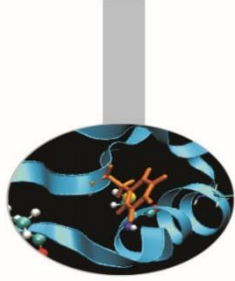
```
const cl::NDRange
      global(300, 200);

const cl::NDRange
      local(30, 20);

kernel(
  EnqueueArgs(global, local),
  …);
```

Indexing work



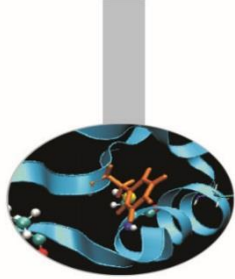
gridDim	get_num_groups()
blockIdx	get_group_id()
blockDim	get_local_size()
gridDim * blockDim	get_global_size()
threadIdx	get_local_id()
blockIdx * blockdim + threadIdx	get_global_id()

Differences in kernels



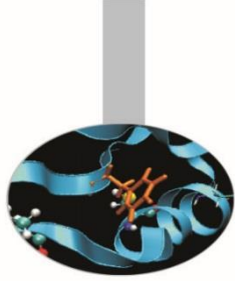
- Where do you find the kernel?
  - OpenCL - either a string (const char *), or read from a file
  - CUDA – a function in the host code
- Denoting a kernel
  - OpenCL - __kernel
  - CUDA - __global__
- When are my kernels compiled?
  - OpenCL – at runtime
  - CUDA – with compilation of host code

Host code



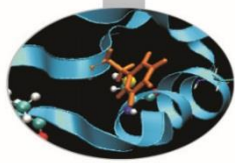
- **By default, CUDA initializes the GPU automatically**
  - If you needed anything more complicated (multi-device etc.) you must do so manually
- **OpenCL always requires explicit device initialization**
  - **It runs not just on NVIDIA® GPUs and so you must tell it which device(s) to use**

Thread Synchronization



__syncthreads()	barrier()
__threadfenceblock()	mem_fence(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE)
No equivalent	read_mem_fence()
No equivalent	write_mem_fence()
__threadfence()	Finish one kernel and start another

Translation from CUDA to OpenCL



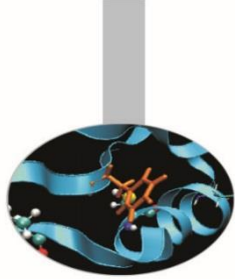
CUDA	OpenCL
GPU	Device (CPU, GPU etc)
Multiprocessor	Compute Unit, or CU
Scalar or CUDA core	Processing Element, or PE
Global or Device Memory	Global Memory
Shared Memory (per block)	Local Memory (per workgroup)
Local Memory (registers)	Private Memory
Thread Block	Work-group
Thread	Work-item
Warp	No equivalent term (yet)
Grid	NDRange

OpenCL live@Eurora

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



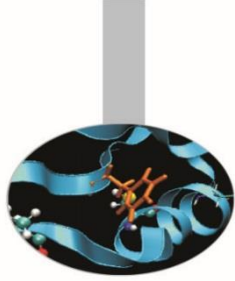
- 13 Multiprocessors
- 2496 CUDA Cores
- 5 GB of global memory
- GPU clock rate 760MHz

NVIDIA Tesla K20

- 236 compute units
- 8 GB of global memory
- CPU clock rate 1052 MHz

Intel MIC Xeon Phi

Setting up OpenCL on Eurora



- Login on front-end.

Then:

>*module load profile/advanced*
> module load intel_opencl/none--intel--cs-xe-2013--binary

It defines:

INTEL_OPENCL_INCLUDE

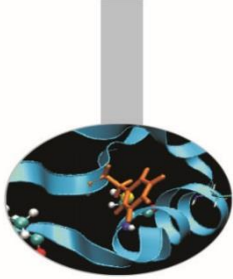
and

INTEL_OPENCL_LIB

environmental variables that can be used:

>cc -I$INTEL_OPENCL_INCLUDE -L$INTEL_OPENCL_LIB -lOpenCL vadd.c -o vadd

Running on Intel



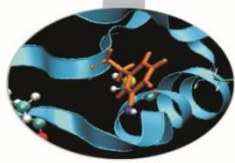
```
PROFILE=FULL_PROFILE
VERSION=OpenCL 1.2 LINUX
NAME=Intel(R) OpenCL
VENDOR=Intel(R) Corporation
EXTENSIONS=cl_khr_fp64 cl_khr_global_int32_base_atomics
cl_khr_global_int32_extended_atomics
cl_khr_local_int32_base_atomics
cl_khr_local_int32_extended_atomics
cl_khr_byte_addressable_store
--0--
DEVICE NAME=      Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz
DEVICE VENDOR=Intel(R) Corporation
DEVICE VERSION=OpenCL 1.2 (Build 67279)
DEVICE_MAX_COMPUTE_UNITS=16
DEVICE_MAX_WORK_GROUP_SIZE=1024
DEVICE_MAX_WORK_ITEM_DIMENSIONS=3
DEVICE_MAX_WORK_ITEM_SIZES=1024 1024 1024
DEVICE_GLOBAL_MEM_SIZE=16685436928
--1--
DEVICE NAME=Intel(R) Many Integrated Core Acceleration Card
DEVICE VENDOR=Intel(R) Corporation
DEVICE VERSION=OpenCL 1.2 (Build 67279)
DEVICE_MAX_COMPUTE_UNITS=236
DEVICE_MAX_WORK_GROUP_SIZE=1024
DEVICE_MAX_WORK_ITEM_DIMENSIONS=3
DEVICE_MAX_WORK_ITEM_SIZES=1024 1024 1024
DEVICE_GLOBAL_MEM_SIZE=6053646336
--2--
DEVICE NAME=Intel(R) Many Integrated Core Acceleration Card
DEVICE VENDOR=Intel(R) Corporation
DEVICE VERSION=OpenCL 1.2 (Build 67279)
DEVICE_MAX_COMPUTE_UNITS=236
DEVICE_MAX_WORK_GROUP_SIZE=1024
DEVICE_MAX_WORK_ITEM_DIMENSIONS=3
DEVICE_MAX_WORK_ITEM_SIZES=1024 1024 1024
DEVICE_GLOBAL_MEM_SIZE=6053646336
Computed sum = 549754961920.0.
Check passed.
```

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

Intel MIC device was selected

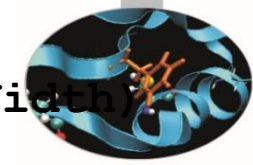
Results are OK no matter what performances

Exercise

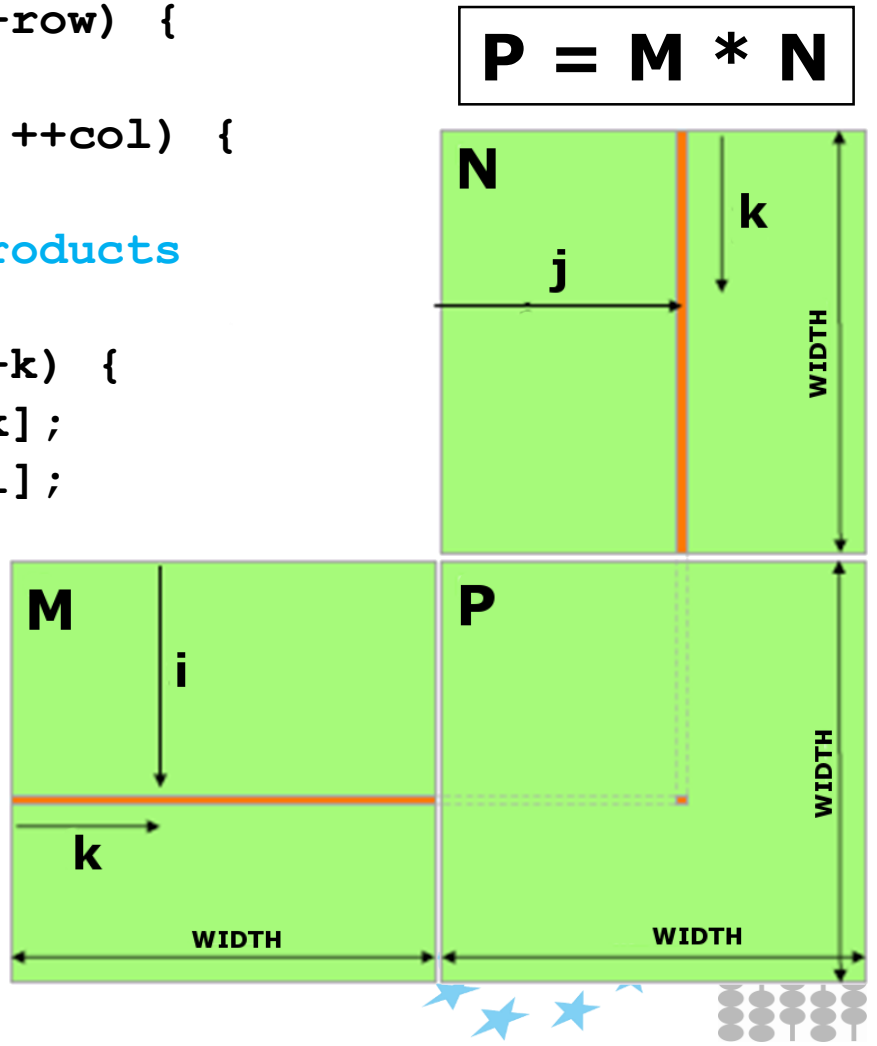


- Goal:
  - To inspect and verify that you can run an OpenCL kernel on Eurora machines
- Procedure:
  - Take the provided C ***vadd.c*** and ***vadd.cl*** source programs from VADD directory
  - Compile and link ***vadd.c***
  - Run on NVIDIA or Intel platform.
- Expected output:
  - A message verifying that the vector addition completed successfully
  - Some useful info about OpenCL environment (Intel and NVIDIA)

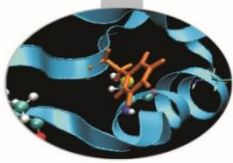
Matrix-Matrix product: HOST

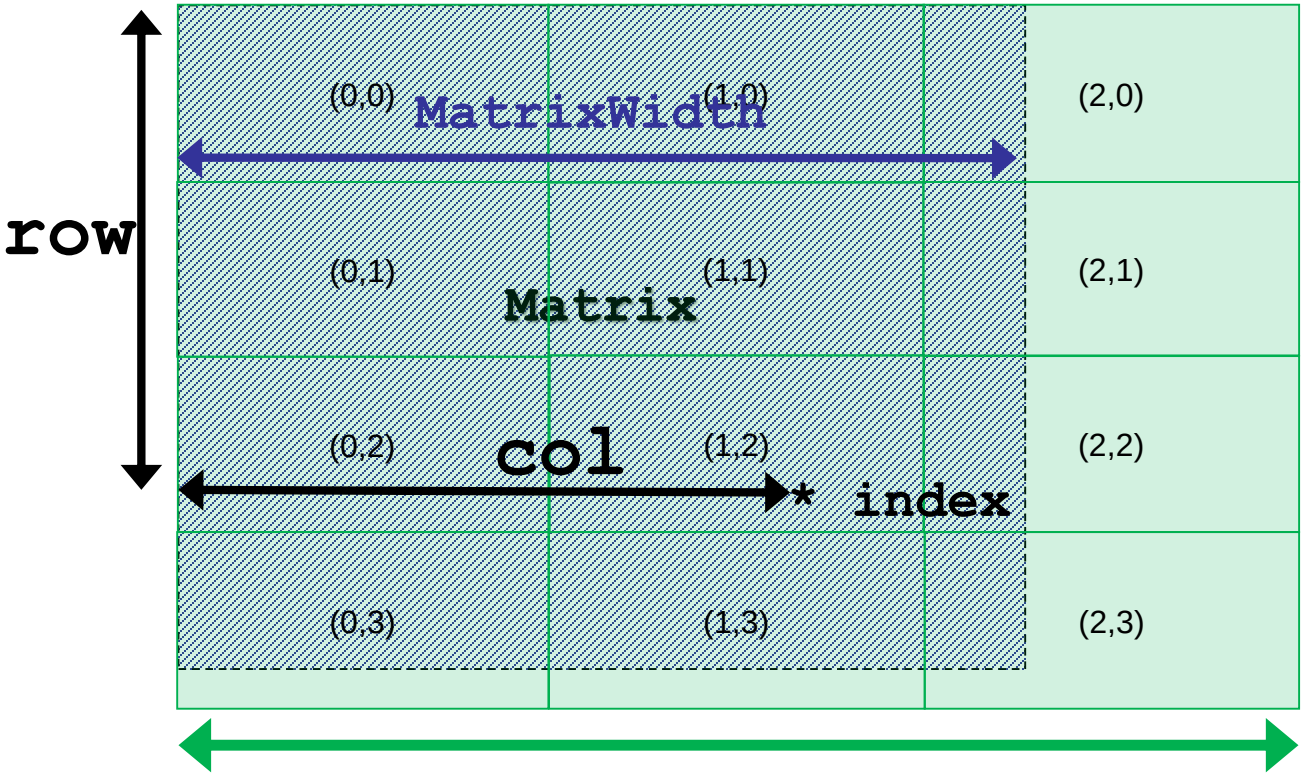


```
void MatrixMulOnHost (float* M, float* N, float* P, int Width)
{
  // loop on rows
  for (int row = 0; row < Width; ++row) {
    // loop on columns
    for (int col = 0; col < Width; ++col) {

      // accumulate element-wise products
      float pval = 0;
      for (int k = 0; k < Width; ++k) {
        float a = M[row * Width + k];
        float b = N[k * Width + col];
        pval += a * b;
      }

      // store final results
      P[row * Width + col] = pval;
    }
  }
}
```



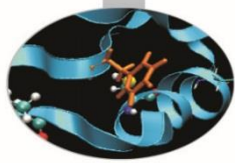
Matrix-Matrix product: launch grid





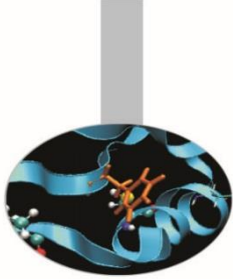

```
col = blockIdx.x * blockDim.x + threadIdx.x;
row = blockIdx.y * blockDim.y + threadIdx.y;

index = row * MatrixWidth + col;
```

Matrix-Matrix product: CUDA Kernel

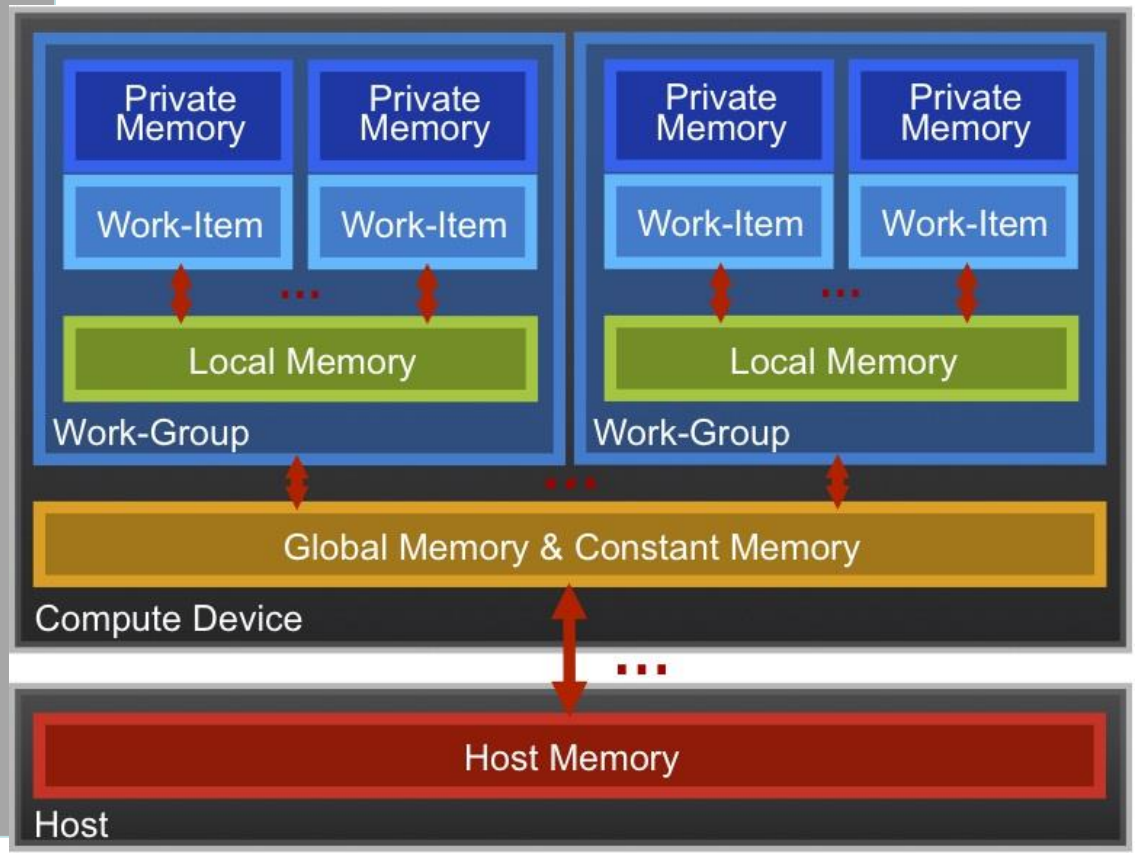


```
__global__ void MMKernel (float* dM, float *dN, float *dP,
                          int width) {
  // row,col from built-in thread indeces(2D block of threads)
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

  // check if current CUDA thread is inside matrix borders
  if (row < width && col < width) {

    // accumulate element-wise products
    // NB: pval stores the dP element computed by the thread
    float pval = 0;
    for (int k=0; k < width; k++)
       pval += dM[row * width + k] * dN[k * width + col];

    // store final results (each thread writes one element)
    dP[row * width + col] = Pvalue;
  }
}
```

OpenCL Memory model

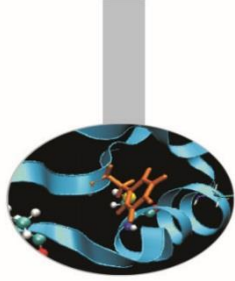


- Private Memory
  - Per work-item
- Local Memory
  - Shared within a work-group
- Global/Constant Memory
  - Visible to all work-groups
- Host memory
  - On the CPU

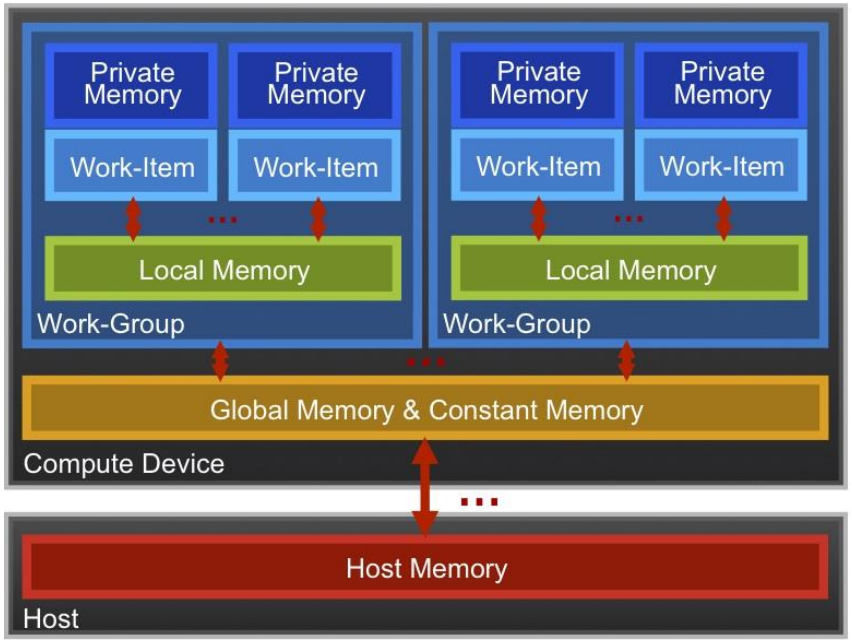


Memory management is **explicit**:
You are responsible for moving data from
host → global → local *and* back

OpenCL Memory model

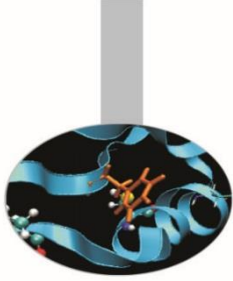


- Private Memory
  - Fastest & smallest: O(10) words/WI
- Local Memory
  - Shared by all WI's in a work-group
  - But not shared between work-groups!
  - O(1-10) Kbytes per work-group
- Global/Constant Memory
  - O(1-10) Gbytes of Global memory
  - O(10-100) Kbytes of Constant memory
- Host memory
  - On the CPU - GBytes

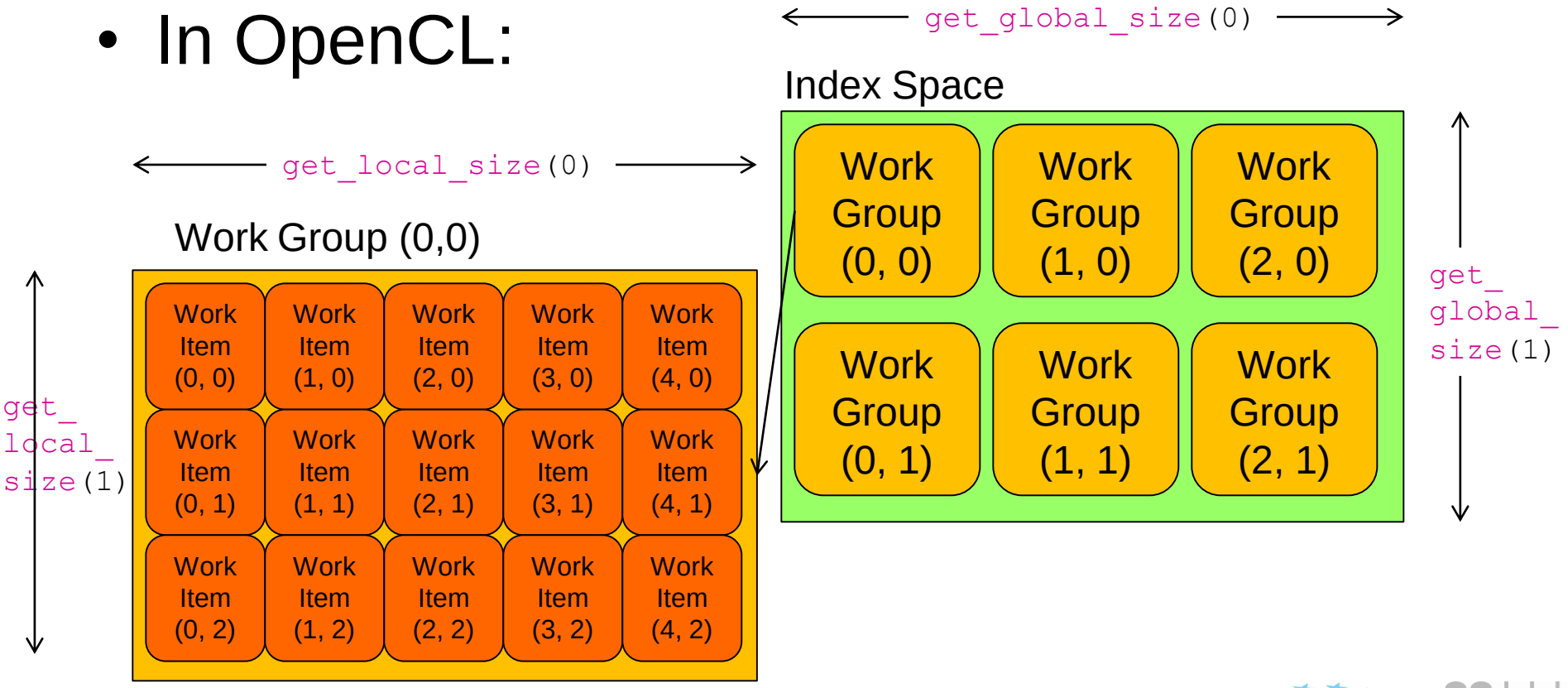


Memory management is **explicit**:
O(1-10) Gbytes/s bandwidth to discrete GPUs for
Host <-> Global transfers

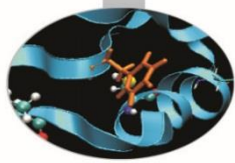
OpenCL mapping



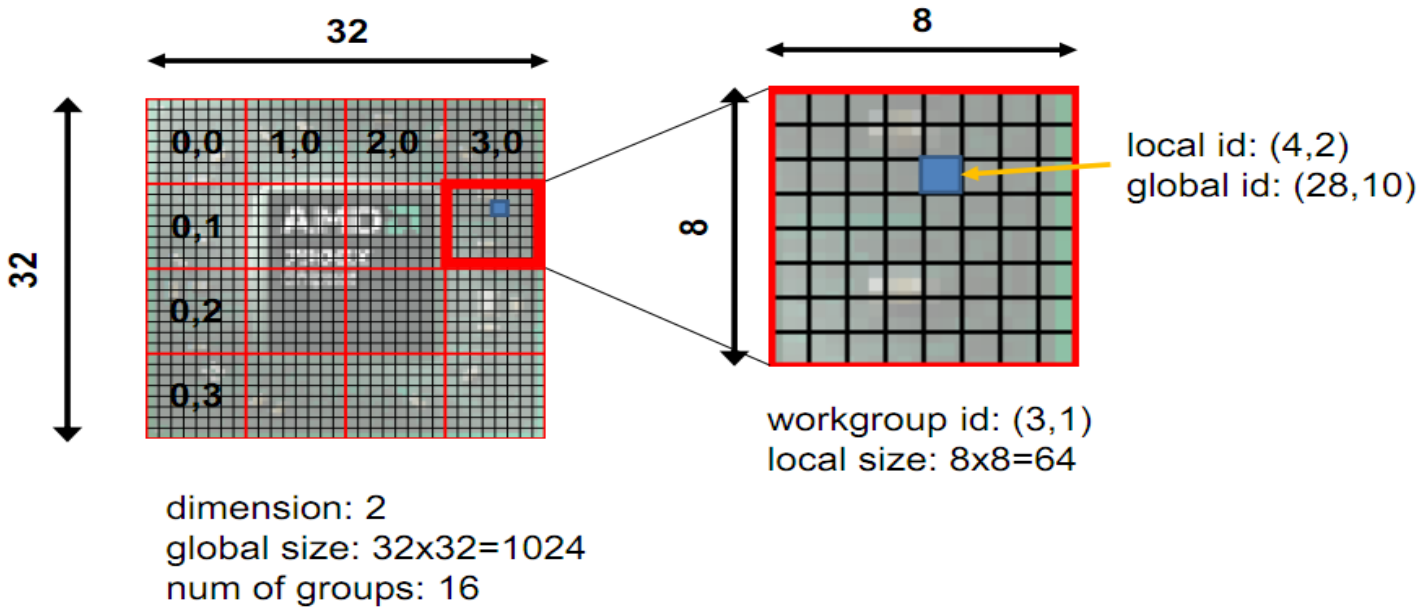
- In OpenCL:



OpenCL mapping (again)

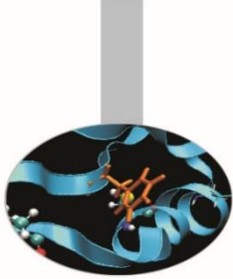


Kernels: Work-item and Work-group Example



You should use OpenCL mapping functions for element values recovery(this may be a common source of bugs when write a kernel)

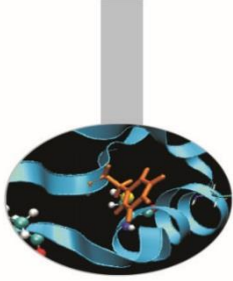
Matrix multiplication: pseudo OpenCL kernel



```
__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;
   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];
                                    }
                                 }
                              }
}
```

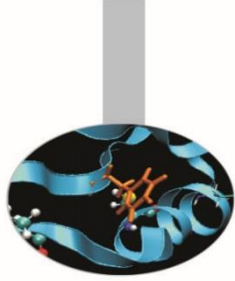
Remove outer loops and set work-item co-ordinates

Matrix multiplication: OpenCL kernel



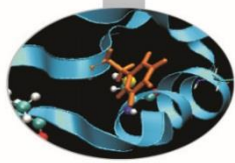
```
__kernel void mat_mul(
 const int Mdim, const int Ndim, const int Pdim,
 __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    j = get_global_id(0);
    i = get_global_id(1);
    // C(i, j) = sum(over k) A(i,k) * B(k,j)
    for (k = 0; k < Pdim; k++) {
      C[i*Ndim+j]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

Exercise 1: Matrix Multiplication



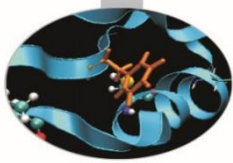
- Goal:
  - To write your first complete OpenCL kernel “from scratch”
  - To multiply a pair of matrices
- Procedure:
  - Start with the previous matrix multiplication OpenCL kernel
  - Rearrange and use local scalars for intermediate C element values (a common optimization in matrix-Multiplication functions)
- Expected output:
  - A message to standard output verifying that the chain of vector additions produced the correct result
  - Report the runtime and the MFLOPS

Matrix multiplication: OpenCL kernel improved



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

Matrix Size	Platform	Kernel time (sec.)	GFLOP/s
2048	NVIDIA K20s	0.24	71
2048	Intel MIC	0.47	37



Matrix-Matrix product: selecting optimum thread block size

Which is the best thread block /work-group size to select (i.e. `TILE_WIDTH`)?
On **Fermi** architectures: each SM can handle up to ***1536*** total threads

`TILE_WIDTH` = 8

8x8 = 64 threads >>> 1536/64 = 24 blocks needed to fully load a SM
… yet there is a limit of maximum 8 resident blocks per SM for cc 2.x
so we end up with just 64x8 = 512 threads per SM on a maximum of 1536 (only **33%** occupancy)

`TILE_WIDTH` = 16

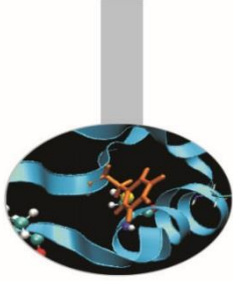
16x16 = 256 threads >>> 1536/256 = 6 blocks to fully load a SM
6x256 = 1536 threads per SM … reaching **full occupancy** per SM!

`TILE_WIDTH` = 32

32x32 = 1024 threads >>> 1536/1024 = 1.5 = 1 block fully loads SM
1024 threads per SM (only **66%** occupancy)

`TILE_WIDTH` = 16

Matrix-Matrix product: selecting optimum thread block size



Which is the best thread block size/work-group size to select (i.e. **`TILE_WIDTH`**)?
On **Kepler** architectures: each SM can handle up to ***2048*** total threads

`TILE_WIDTH` = **8**

8x8 = 64 threads >>> 2048/64 = 32 blocks needed to fully load a SM
… yet there is a limit of maximum 16 resident blocks per SM for cc 3.x
so we end up with just 64x16 = 1024 threads per SM on a maximum of 2048 (only **50%** occupancy)

`TILE_WIDTH` = **16**

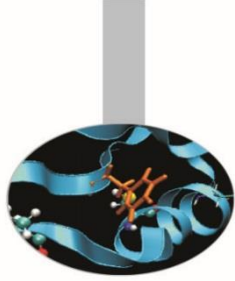
16x16 = 256 threads >>> 2048/256 = 8 blocks to fully load a SM
8x256 = 2048 threads per SM … reaching **full occupancy** per SM!

`TILE_WIDTH` = **32**

32x32 = 1024 threads >>> 2048/1024 = 2 blocks fully load a SM
2x1024 = 2048 threads per SM … reaching **full occupancy** per SM!

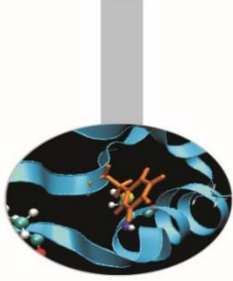
`TILE_WIDTH` = **16** or **32**

Exercise 2: Matrix Multiplication



- Goal:
  - To test different thread block size
  - To multiply a pair of matrices
- Procedure:
  - Start with the previous matrix multiplication OpenCL kernel
  - Test different thread block size in C source code. Compare results and find the optimum value (on both OpenCL platforms)
- Expected output:
  - A message to standard output verifying that the chain of vector additions produced the correct result
  - Report the runtime and the MFLOPS

Matrix-Matrix product: selecting optimum thread block size



Which is the best thread block size/work-group size to select (i.e. `TILE_WIDTH`)?
On **Kepler** architectures: each SM can handle up to ***2048*** total threads

`TILE_WIDTH` = 8

8x8 = 64 threads >>> 2048/64 = 32 blocks needed to fully load a SM
… yet there is a limit of maximum 16 resident blocks per SM for cc 3.x
so we end up with just 64x16 = 1024 threads per SM on a maximum of 2048 (only **50%** occupancy)

`TILE_WIDTH` = 16

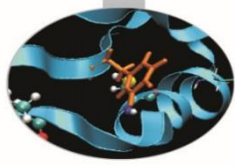
16x16 = 256 threads >>> 2048/256 = 8 blocks to fully load a SM
8x256 = 2048 threads per SM … reaching **full occupancy** per SM!

`TILE_WIDTH` = 32

32x32 = 1024 threads >>> 2048/1024 = 2 blocks fully load a SM
2x1024 = 2048 threads per SM … reaching **full occupancy** per SM!

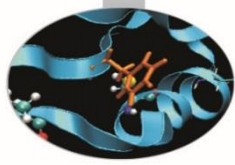
TILE_WIDTH	Kernel time (sec.)	GFLOP/s (NVIDIA K20)
8	0.33	52
16	0.20	82
32	0.16	104

Exercise 3: Matrix Multiplication



- Goal:
  - To check inside matrix borders
  - To multiply a pair of matrices
- Procedure:
  - Start with the previous matrix multiplication OpenCL kernel
  - Test the check inside matrix borders kernel and the original one. Compare results and performances (on both OpenCL platforms)
- Expected output:
  - A message to standard output verifying that the chain of vector additions produced the correct result
  - Report the runtime and the MFLOPS

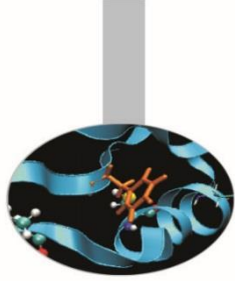
Matrix-Matrix product: check inside matrix borders



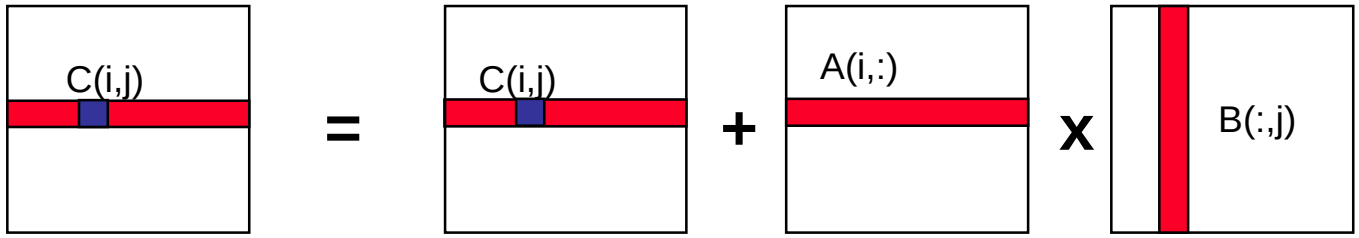
```
__global__ void MMKernel (float* dM, float *dN, float *dP,
                          int width) {
  // row,col from built-in thread indeces(2D block of
threads)
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

  // check if current CUDA thread is inside matrix borders
  if (row < width && col < width) {
```

kernel chek (Yes/No)	Matrices Size	Kernel Error	GFLOP/s (Intel MIC)
Yes	2047	/	20
Yes	2048	/	35
No	2047	Failed (different results from reference)	21
No	2048	/	37

Optimizing matrix multiplication

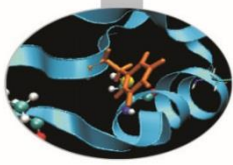


- There may be significant overhead to manage work-items and work-groups.
- So let's have each work-item compute a full row of C



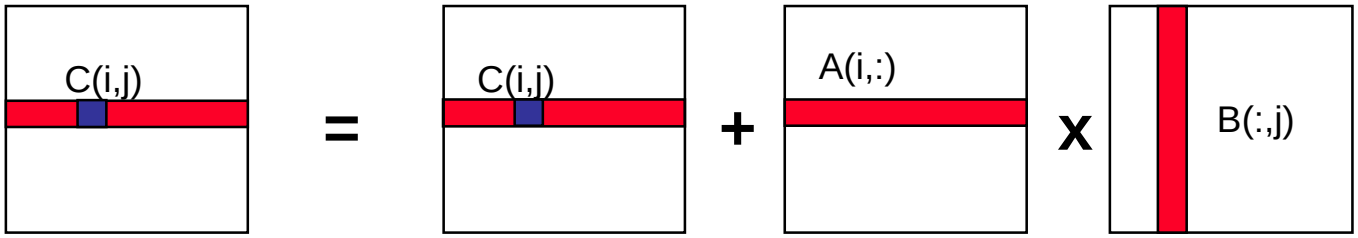
Dot product of a row of A and a column of B for each element of C

Exercise 1-1: Matrix Multiplication



- Goal:
  - Let each work-item to compute a full row of C
  - To multiply a pair of matrices
- Procedure:
  - Start with the previous matrix multiplication OpenCL kernel
  - Modify in order to have each work-item computing a full row of C
  - Test the new kernel.Compare results and performances (on both OpenCL platforms)
- Expected output:
  - A message to standard output verifying that the chain of vector additions produced the correct result
  - Report the runtime and the MFLOPS

Optimizing matrix multiplication

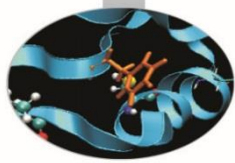


Dot product of a row of A and a column of B for each element of C

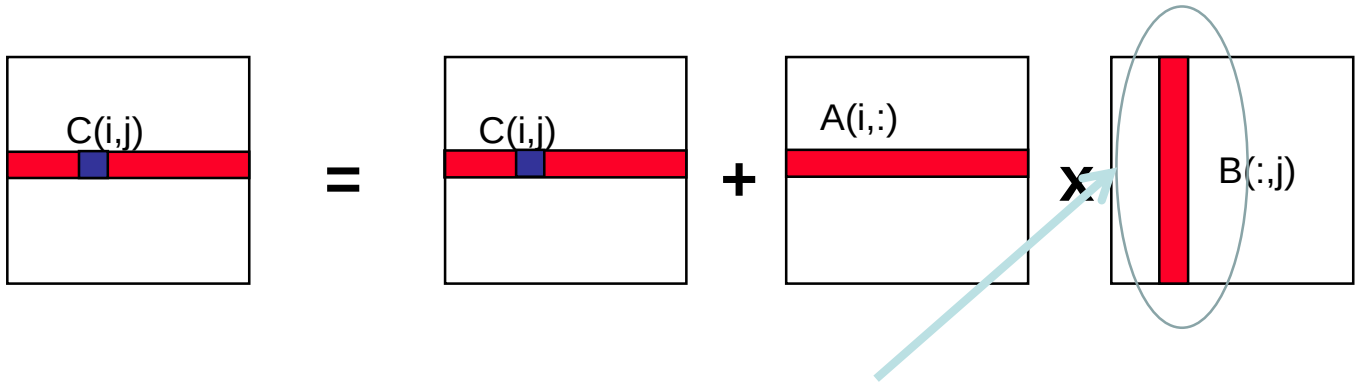
Matrix Size	Platform	Kernel time (sec.)	GFLOP/s
2048	NVIDIA K20s	1.17	15
2048	Intel MIC	0.88	20

This change doesn't really help.

Optimizing matrix multiplication

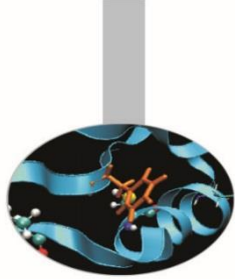


- Notice that, in one row of C, each element reuses the same column of B.
- Let's copy that column of B into private memory of the work-item that's (exclusively) using it to avoid the overhead of loading it from global memory for each C(i,j) computation.



Private memory of each work-item

Private Memory

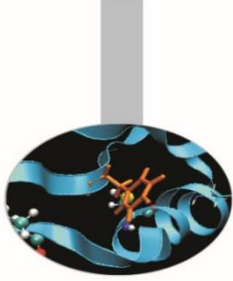


- Managing the memory hierarchy is one of ***the*** most important things to get right to achieve good performance

- 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

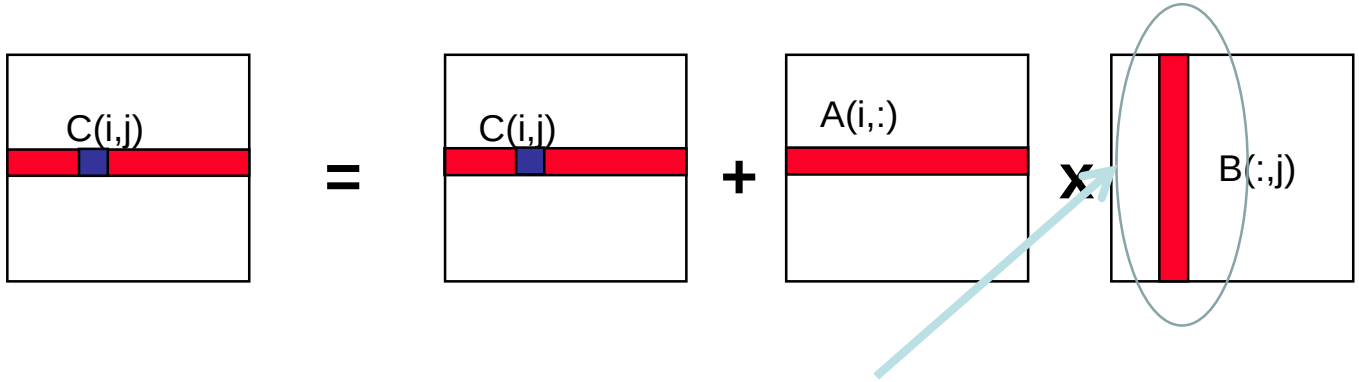
* Occupancy on a GPU

Exercise 1-2: using private memory



- Goal:
  - Use private memory to minimize memory movement costs and optimize performance of your matrix multiplication program
- Procedure:
  - Start with previous matrix multiplication kernel
  - Modify the kernel so that each work-item copies its own column of B into private memory
  - Test the new kernel.Compare results and performances (on both OpenCL platforms)
- Expected output:
  - A message to standard output verifying that the matrix multiplication program is generating the correct results
  - Report the runtime and the MFLOPS

Optimizing matrix multiplication

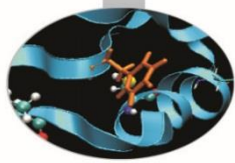


Private memory of each

Matrix Size	Platform	Kernel time (sec.)	GFLOP/s
2048	NVIDIA K20s	1.17	15
2048	Intel MIC	0.21	80

This has started to help.

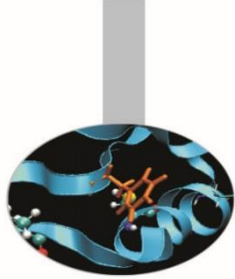
Local Memory*



- 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

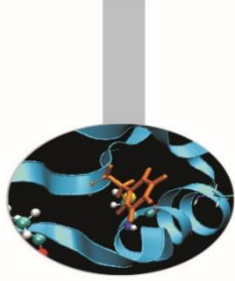
* Typical figures for a 2013 GPU

Local Memory



- Local Memory doesn't always help…
  - CPUs, MICs 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!

Using *Local/Shared Memory* for Thread Cooperation

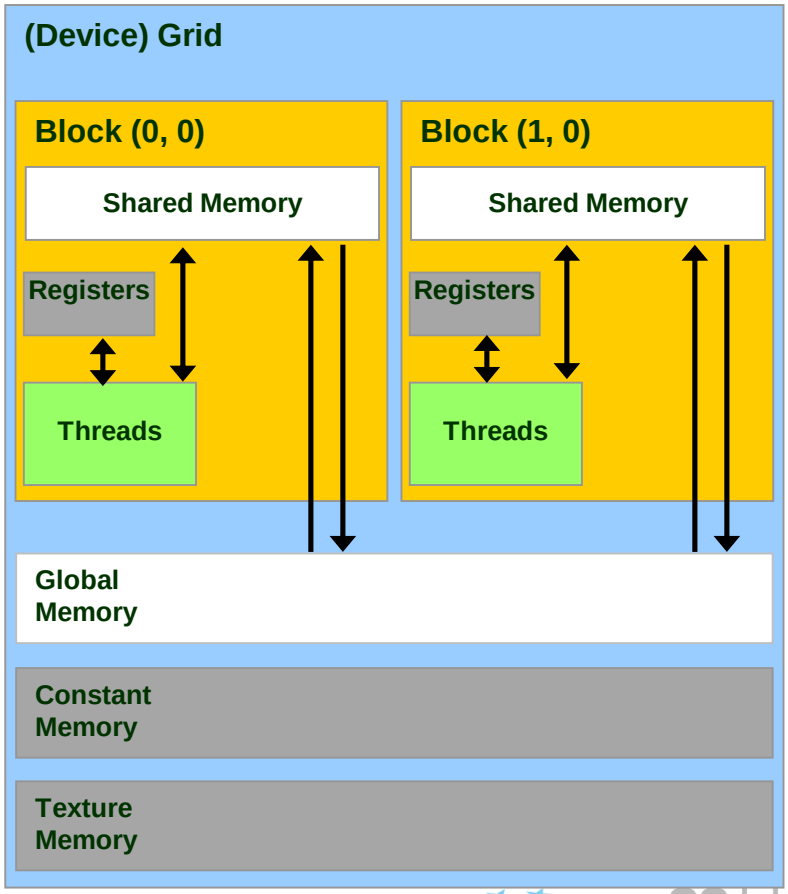


Threads belonging to the same block can cooperate togheter using the shared memory to share data

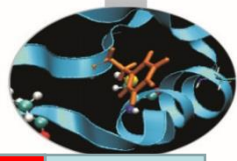
if a thread needs some data which has been already retrived by another thread in the same block, this data can be shared using the shared memory

Typical Shared Memory usage:

1. declare a buffer residing on shared memory (this buffer is per block)
2. load data into shared memory buffer
3. synchronize threads so to make sure all needed data is present in the buffer
4. performe operation on data
5. synchronize threads so all operations have been performed
6. write back results to global memory



Matrix-matrix using Shared Memory



Cij=0.

Cycle on block

kb=0, N/NB

As(it,jt) = A(ib*NB + it, kb*NB + jt)

Bs(it,jt) = B(kb*NB + it, jb*NB + jt)

Thread Synchronization

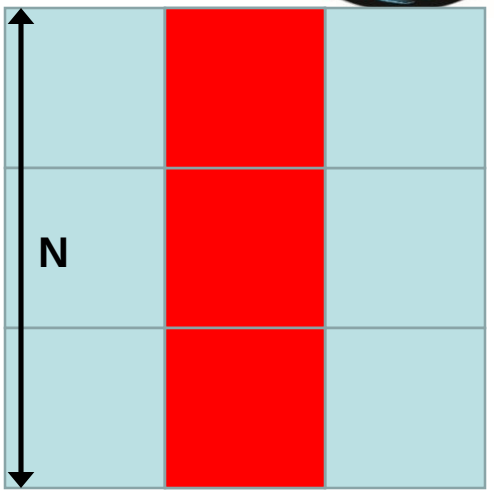
Cycle on block k=1,NB

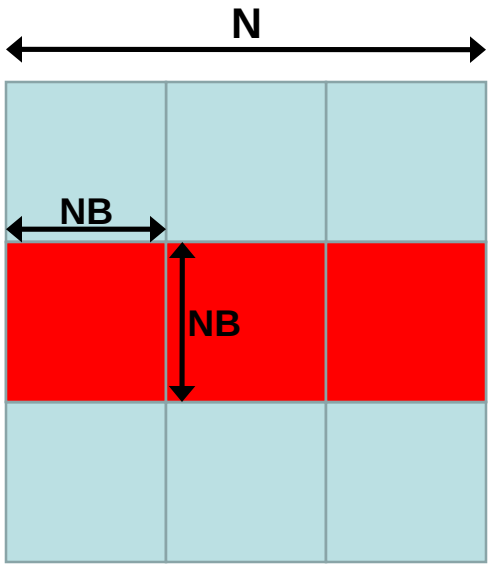
Cij=Cij+As(it,k)·Bs(k,jt)

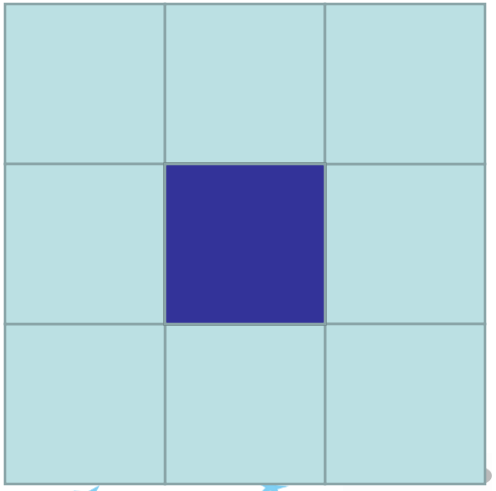
Thread Synchronization

C(i,j)=Cij

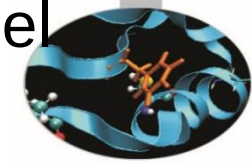
```
it = threadIdx.y
jt = threadIdx.x

ib = blockIdx.y
jb = blockIdx.x
```





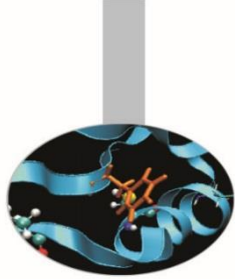


Matrix-matrix using Shared Memory: CUDA Kernel



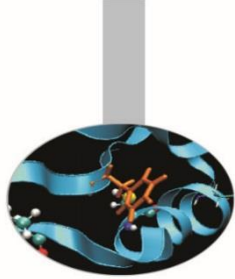
```
// Matrix multiplication kernel called by MatMul_gpu()
__global__ void MatMul_kernel (float *A, float *B, float *C, int N)
{

  // Shared memory used to store Asub and Bsub respectively
  __shared__  float Asub[NB][NB];
  __shared__  float Bsub[NB][NB];

  // Block row and column
  int ib = blockIdx.y;
  int jb = blockIdx.x;

  // Thread row and column within Csub
  int it = threadIdx.y;
  int jt = threadIdx.x;

  int a_offset , b_offset, c_offset;

  // Each thread computes one element of Csub
  // by accumulating results into Cvalue
  float Cvalue = 0;

  // Loop over all the sub-matrices of A and B that are
  // required to compute Csub
  // Multiply each pair of sub-matrices together
  // and accumulate the results
  for (int kb = 0; kb < (A.width / NB); ++kb) {

    // Get the starting address of Asub  and Bsub
    a_offset = get_offset (ib, kb, N);
    b_offset = get_offset (kb, jb, N);

    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    Asub[it][jt] = A[a_offset + it*N + jt];
    Bsub[it][jt] = B[b_offset + it*N + jt];

    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    __syncthreads();

    // Multiply Asub and Bsub together
    for (int k = 0; k < NB; ++k) {
      Cvalue += Asub[it][k] * Bsub[k][jt];
    }
    // Synchronize to make sure that the preceding
    // computation is done
    __syncthreads();
  }

  // Get the starting address (c_offset) of Csub
  c_offset = get_offset (ib, jb, N);
  // Each thread block computes one sub-matrix Csub of C
  C[c_offset + it*N + jt] = Cvalue;

}
```

Exercise 4: Matrix Multiplication



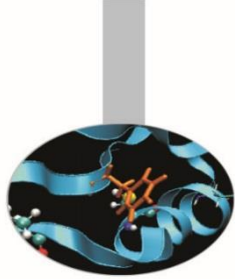
- Goal:
  - To use shared/memory local
  - To multiply a pair of matrices
- Procedure:
  - Start with the previous matrix multiplication CUDA kernel
  - Modify source in order to generate an OpenCL kernel
  - Compare results and performances (on both OpenCL platforms)
- Expected output:
  - A message to standard output verifying that the chain of vector additions produced the correct result
  - Report the runtime and the MFLOPS

Matrix-matrix using Shared Memory: OpenCL Kernel:results



Matrix Size	Platform	Kernel time (sec.)	GFLOP/s
2048	NVIDIA K20s	0.10	166
2048	Intel MIC	0.15	115

OpenCL on Intel MIC

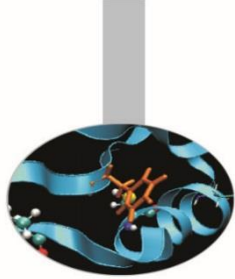


- Intel MIC combines many core onto a single chip. Each core runs exactly **4 hardware threads**. In particular:

1. **All cores/threads are a single OpenCL device**
2. **Separate hardware threads are OpenCL CU.**

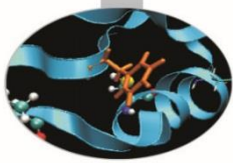
- In the end, you'll have parallelism at the work-group level (vectorization) and parallelism between work-groups (threading).

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 adiacent 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).

Matrix-matrix on Intel MIC (skeleton)



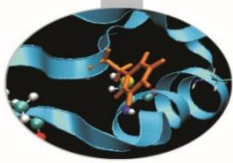
```
for i from 0 to NUM_OF_TILES_M-1
      for j from 0 to NUM_OF_TILES_N-1
            C_BLOCK = ZERO_MATRIX(TILE_SIZE_M, TILE_SIZE_N)
            for k from 0 to size-1
                  for ib = from 0 to TILE_SIZE_M-1
                        for jb = from 0 to TILE_SIZE_N-1
                     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

TILE_SIZE_K = size of block for internal computation of C_BLOCK

TILE_GROUP_M x TILE_GROUP_N = number of WI within each WG

TILE_SIZE_M x TILE_SIZE_N = number of elements of C computed by one WI

Matrix-matrix on Intel MIC (results)



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

Matrices Size	Kernel time (sec.)	GFLOP/s (Intel MIC)
3968	0.3	415

Conclusions

The future of Accelerator Programming

Most of the latest supercomputers are based on accelerators platform. This huge adoption is the result of:

- High (peak) performances
- Good energy efficiency
- Low price

Accelerators should be used everywhere and all the time. So, why aren't there?

Conclusions

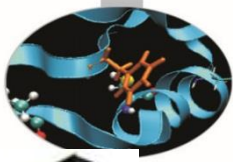
The future of Accelerator Programming

There are two main difficulties with accelerators:

- They can only execute certain type of programs efficiently (high parallelism, data reuse, regular control flow and data access)
- Architectural disparity with respect to CPU (cumbersome programming, portability is an issue)

Accelerators should be used everywhere and all the time. So, why aren't there?

Conclusions



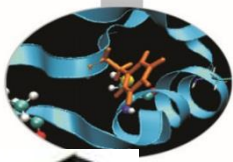
The future of Accelerator Programming

GPUs are now more general-purpose computing devices thanks to CUDA adoption. On the other hand, the fact that CUDA is a proprietary tool and its complexity triggered the creation of other programming approaches:

- **OpenCL**
- OpenAcc
- …
- …

Accelerators should be used everywhere and all the time. So, why aren't there?

Conclusions

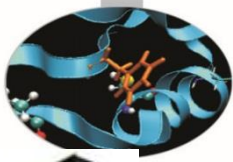


The future of Accelerator Programming

- **OpenCL is the non-proprietary counterpart of CUDA (also supports AMD GPUs, CPUs, MIC, FPGAs….really portable!) but just like CUDA , is very low level and require a lot of programming skills to be used**.
- OpenACC is a very high-level approach. Similar to OpenMP (they should be merged in a near(?) future) but still at its infancy and currently supported by a few compilers
- Other approaches like C++AMP only tied to exhotic HPC environment (Windows) and impractical for standard HPC applications

Accelerators should be used everywhere and all the time. So, why aren't there?

Conclusions



The future of Accelerator Programming

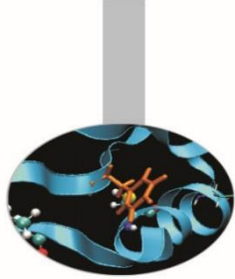
- So, how to (efficiently) program actual and future devices?
- A possible answer could be surprisingly simple and similar to how today's multicore (CPUs) are used (including SIMD extensions, accelerators,…)
- Basically, there are three levels:

- libraries
- automated tools
- do-it-yourself

- Programmers will employ library approach whenever possible. In absence of efficient libraries, tools could be used.
- For the remaining cases, the do-it-yourself approach will have to be used (**OpenCL or a derivative of it should be preferred to proprietary CUDA)**

Accelerators will be used everywhere and all the time. So, start to use them!

Credits



Among the others:

- Simon McIntosh Smith for OpenCL
- CUDA Team in CINECA (Luca Ferraro, Sergio Orlandini, Stefano Tagliaventi)
- MontBlanc project (EU) Team