24th Summer School on PARALLEL COMPUTING

Introduction to Accelerators: CUDA+OpenCL

Piero Lanucara – p.lanucara@cineca.it
SuperComputing Applications and Innovation Department

CINECA

Single core

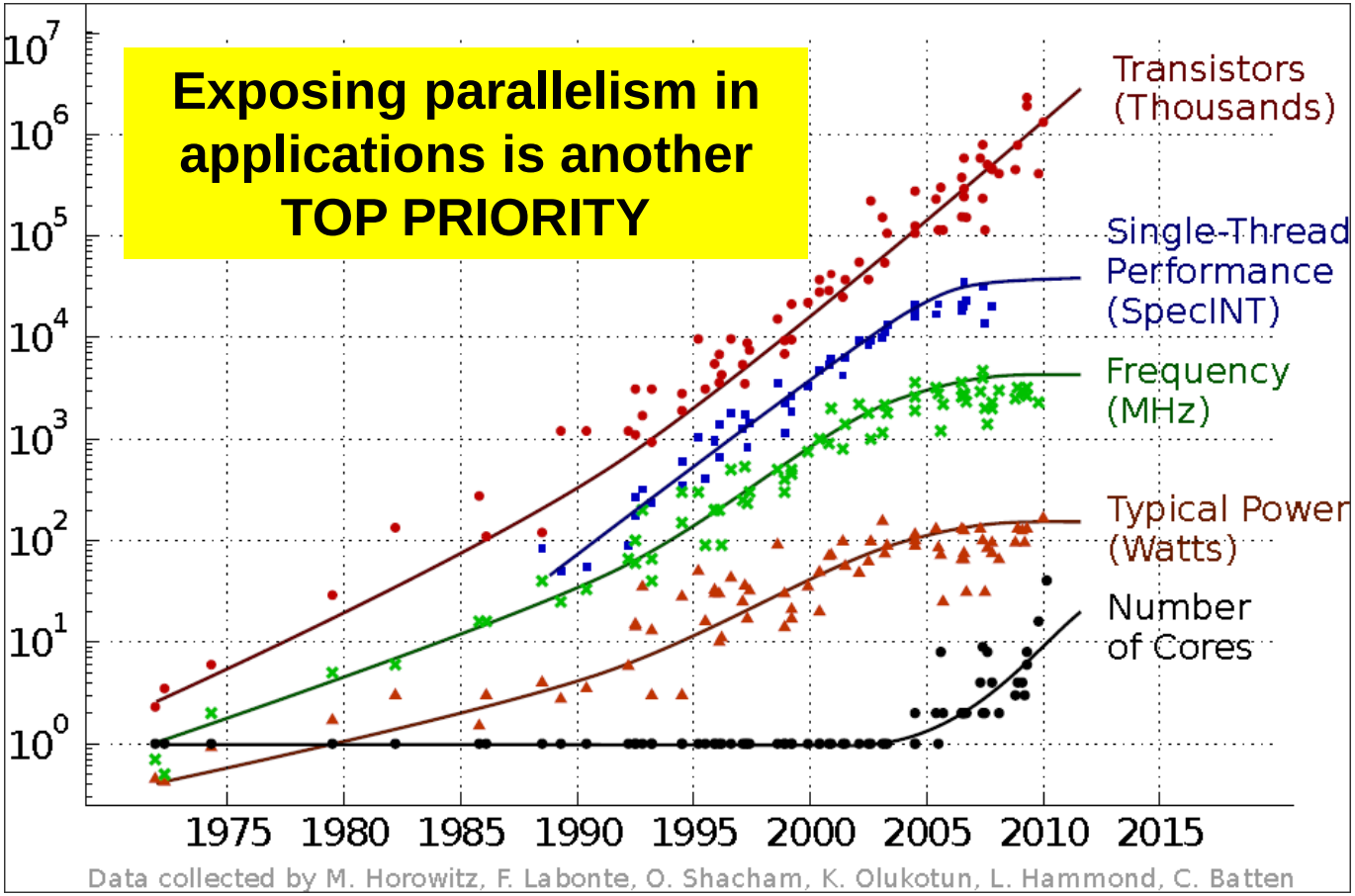
Multi core

Many core

GPU

Intel MIC

35 YEARS OF MICROPROCESSOR TREND DATA

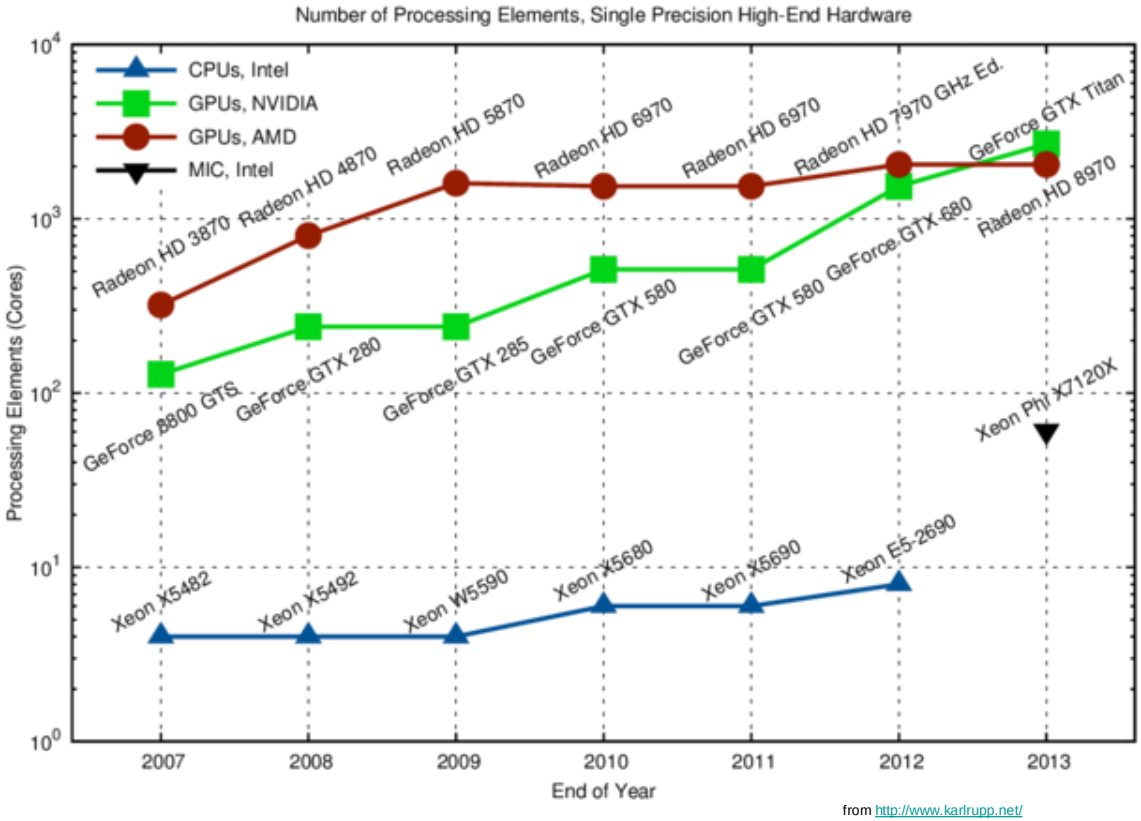


Source: C. Moore's talk at Salishan, April 2011

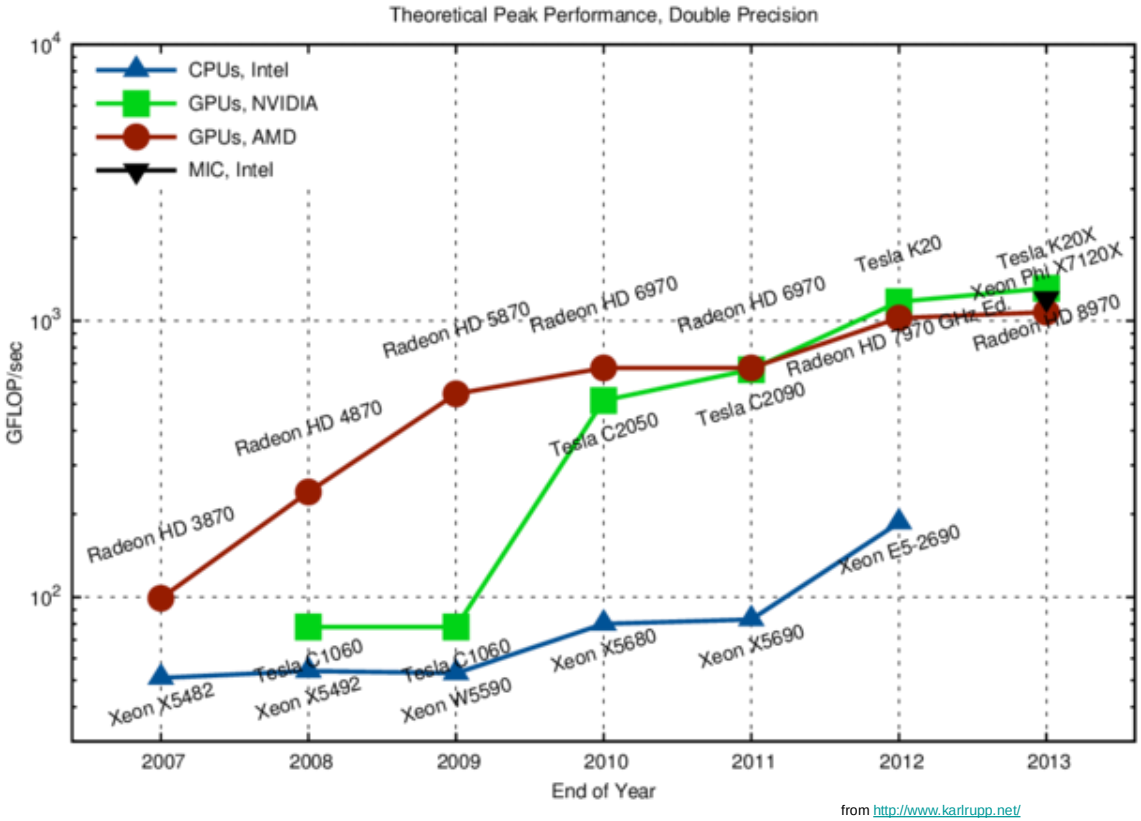
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3120000	33862.7	54902.4	17808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	17173.2	20132.7	7890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786432	8586.6	10066.3	3945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115984	6271.0	7788.9	2325
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462462	5168.1	8520.1	4510
8	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458752	5008.9	5872.0	2301
9	DOE/NNSA/LLNL United States	**Vulcan** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393216	4293.3	5033.2	1972
10	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147456	2897.0	3185.1	3423

THE GREEN 500™

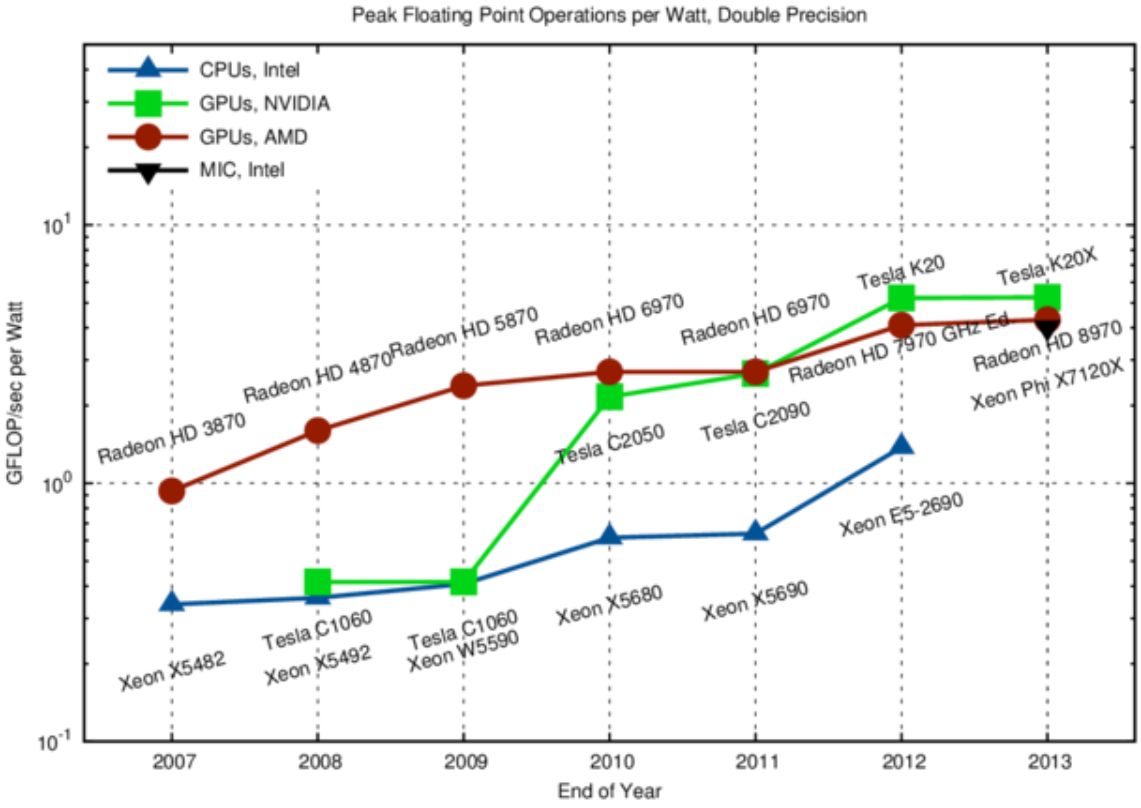
Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	4,503.17	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x	27.78
2	3,631.86	Cambridge University	Wilkes - Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20	52.62
3	3,517.84	Center for Computational Sciences, University of Tsukuba	HA-PACS TCA - Cray 3623G4-SM Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, NVIDIA K20x	78.77
4	3,185.91	Swiss National Supercomputing Centre (CSCS)	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Level 3 measurement data available	1,753.66
5	3,130.95	ROMEO HPC Center - Champagne-Ardenne	romeo - Bull R421-E3 Cluster, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR, NVIDIA K20x	81.41
6	3,068.71	GSIC Center, Tokyo Institute of Technology	TSUBAME 2.5 - Cluster Platform SL390s G7, Xeon X5670 6C 2.930GHz, Infiniband QDR, NVIDIA K20x	922.54
7	2,702.16	University of Arizona	iDataPlex DX360M4, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR14, NVIDIA K20x	53.62
8	2,629.10	Max-Planck-Gesellschaft MPI/IPP	iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x	269.94
9	2,629.10	Financial Institution	iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x	55.62
10	2,358.69	CSIRO	CSIRO GPU Cluster - Nitro G16 3GPU, Xeon E5-2650 8C 2.000GHz, Infiniband FDR, Nvidia K20m	71.01



from http://www.karlrupp.net/



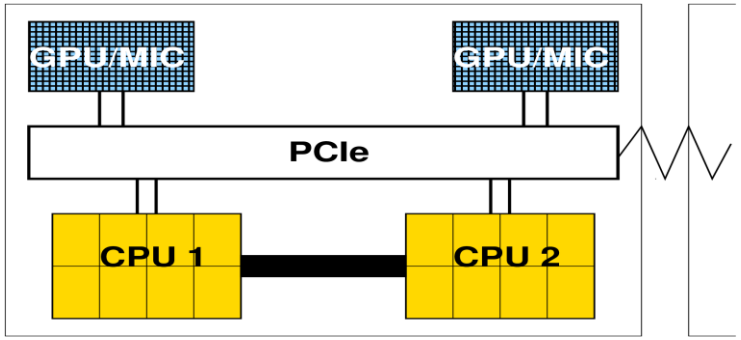
from http://www.karlrupp.net/



from http://www.karlrupp.net/

GPU and many core computing: a view from the top

Basic principle (today's GPUs, many-core coprocessors):
- accelerator “cards” for standard cluster nodes (PCIe)
- many (~50...500) “lightweight” cores (~ 1 GHz)
- high thread concurrency, fast (local) memories



System architecture:
- currently: x86 “Linux-clusters” with nodes comprising
- 2 CPUs (2x 8 cores)
- max. 2...3 accelerator cards (GPU, MIC) per node
- future: smaller CPU component (extreme: “host-less”,many-core chips)

Programming paradigms:
- use CPU for program control, communication and maximum single-thread performance
- “offload” data-parallel parts of computation to accelerator for maximum throughput performance
- requires heterogeneous programming & load-balancing, careful assessment of “speedups”

Motivation

Compute performance

- GPU/many-core computing is promising huge application-performance gains
- caveat: sustained performance on “real-world”, scientific applications
- observations:
- apparent GPU success stories: PetaFlops performance (Gordon-Bell Price nominations)
- from aggressive marketing for Intel MIC, NVIDIA GPUs…

… towards more realistic attitudes: factor 2x..3x speedups (GPU vs. multi-core CPU)

Energy efficiency

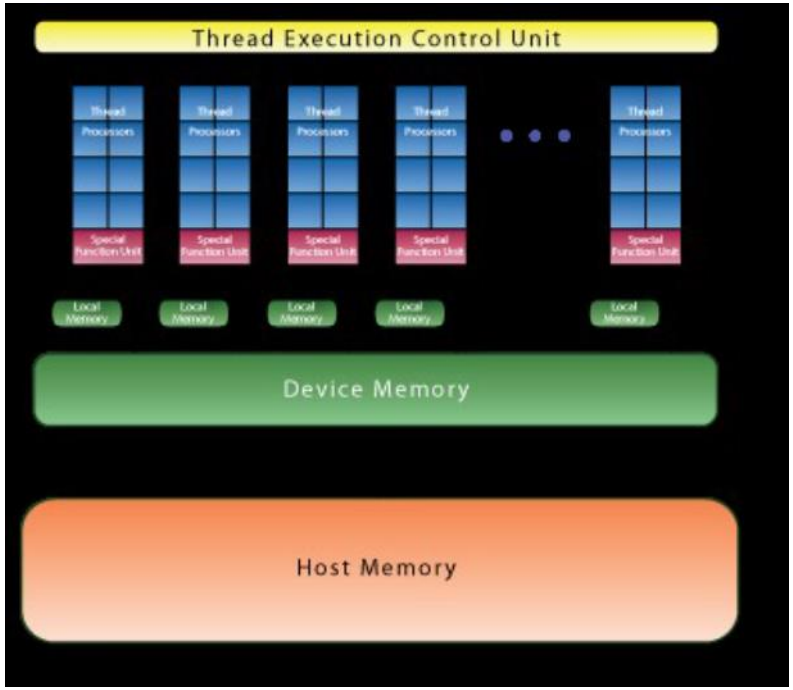
- GPU/many-core computing is promising substantial energy-efficiency gains (a must for exascale)
- caveat: sustained efficiency on “real-world” CPU-GPU clusters

Existing resources

- there is significant GPU/many-core-based compute-power around in the world
- by many, the technology is considered inevitable for the future
- **caveat: the price to pay for application development** ?

Hardware overview (NVIDIA Tesla series)

- since 2011: “Fermi”: first product with HW support for double-precision and ECC memory
- up to 512 cores, 6 GB RAM
- high internal memory bandwidth ~180 GB/s
- 0.5 TFlops (DP, floating point)
- data exchange with host via PCIe (~8 GB/s)
- enhancements: MPI optimization, intra-node comm. (“GPU direct”, “HyperQ”, ...)



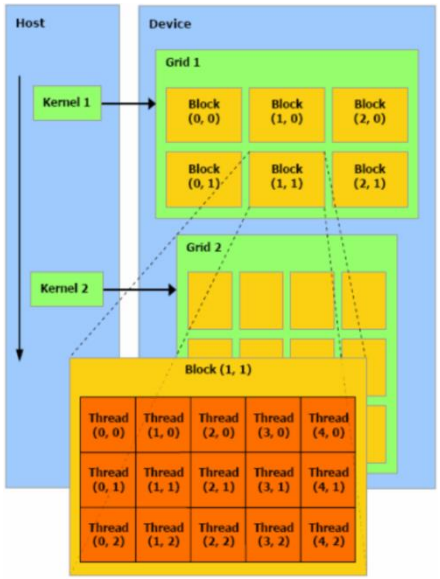
Q1/2013: “Kepler K20”:

- GK110 GPU: up to 2880 cores, 6...12 GB RAM
- internal memory bandwidth: ~200 GB/s
- nominal peak performance: ~ 1.3 TFlops (DP)

plans for a “hostless” chip (for Exascale)

Software & programming models

- paradigm: split program into host code (CPU) and device code (GPU)
- GPU hardware architecture requires highly homogeneous program flow (SIMT, no if-branches!)
- PCIe bottleneck for communication of data between CPU and GPU:
- O(n2)...O(n3) computations for communication of n data
- overlapping of communication and computation phases

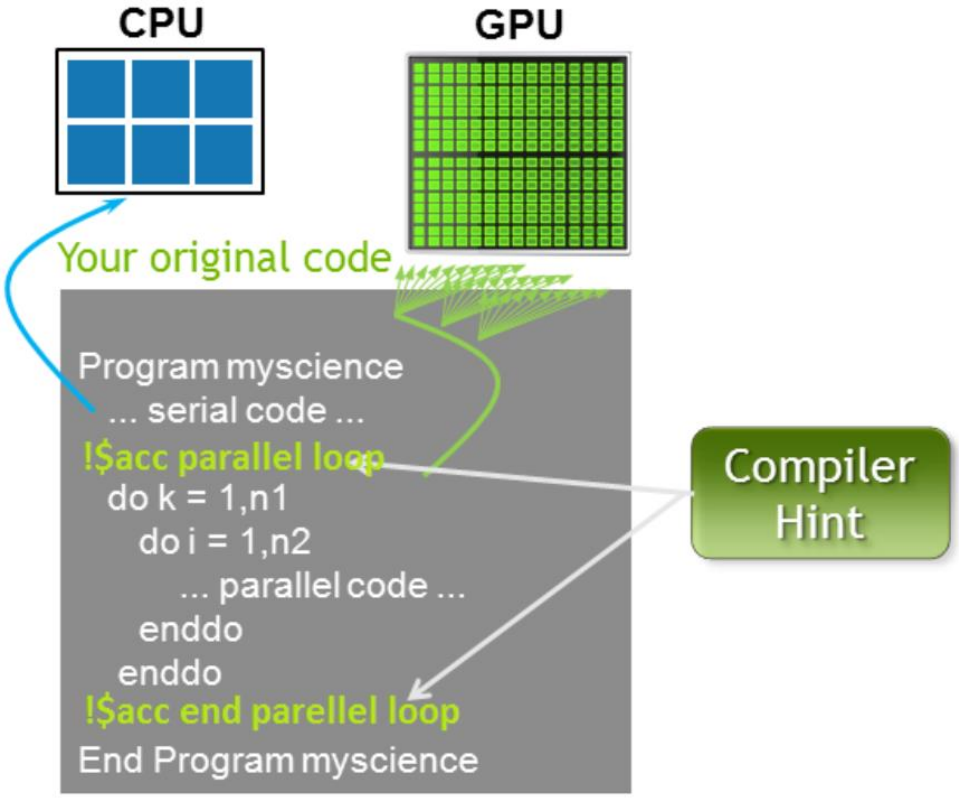


Programming languages

- CUDA (NVIDIA**), OpenCL** (open standard)
- host program (C, executes on CPU) and device kernels (C, launch on GPU)
- numerical libraries: CUBLAS, CUFFT, higher LA: CULA, MAGMA
- tools: debuggers, profiling, system monitoring,…
- CUDA-FORTRAN (PGI)
- directive-based approaches (PGI, CRAY, CAPS, OpenACC, OpenMP-4)
- high-level, comparable to OpenMP
- proprietary (CRAY, PGI, HMPP, ...) → OpenACC → OpenMP

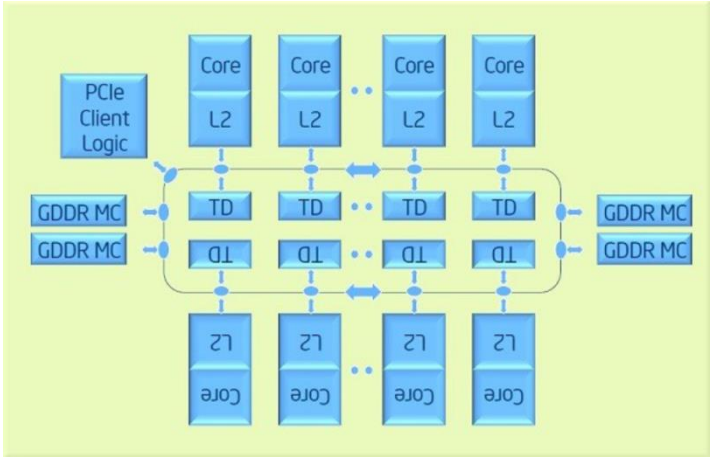
OpenACC

- joint effort of vendors to shortcut/guide OpenMP 4.0 standardization effort
- functional (not performance) portability
- minimally invasive to existing code
- facilitates incremental porting
- compilers: PGI, CRAY, CAPS
- no free lunch!



Hardware overview

- since 2011: “Knights Ferry”: software development platform
- Q4/2012: “Knights Corner”: first product of the new Intel Xeon Phi processor line (MIC arch)
- approx 60 x86 cores (~ 1GHz), 8 GB RAM
- internal memory bandwidth: 175 GB/s
- nominal peak performance: 1 TFlops (DP)
- more than a device: runs Linux OS, IP addressable
- data exchange with host via PCIe (~8 GB/s)
- towards a true many-core chip (“Knights Landing”, 2014)



Software & programming models

- paradigms:

1) offload model (like GPU: split program into host code (CPU) and device code (MIC))

2) cluster models (MPI ranks distributed across CPUs and/or MICs)

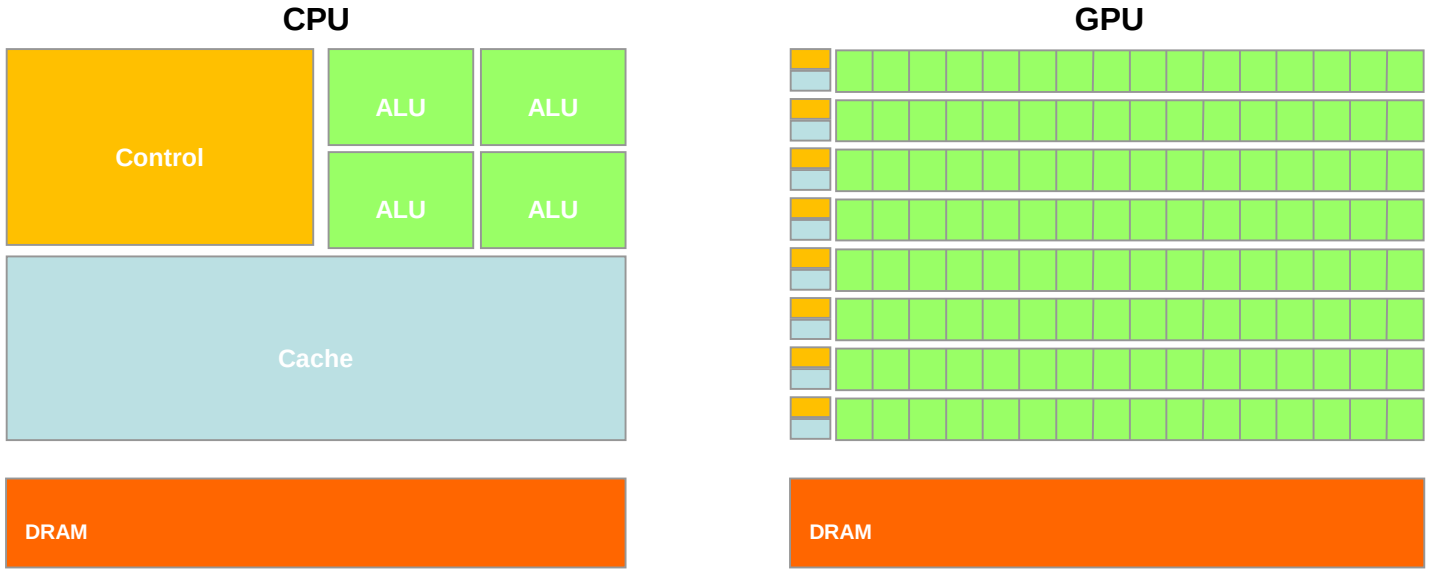
- tools & libraries: the familiar Intel tool chain: compilers, MPI/OpenMP, MKL, ...
- syntax: “data offload” directives + OpenMP (and/or MPI)
- **OpenCL**

- **G**raphics **P**rocessor **U**nit
  - a device equipped with an highly parallel microprocessor (*many-core*) and a private memory with very high bandwidth
- born in response to the growing demand for high definition 3D rendering graphic applications

CPU vs GPU Architectures

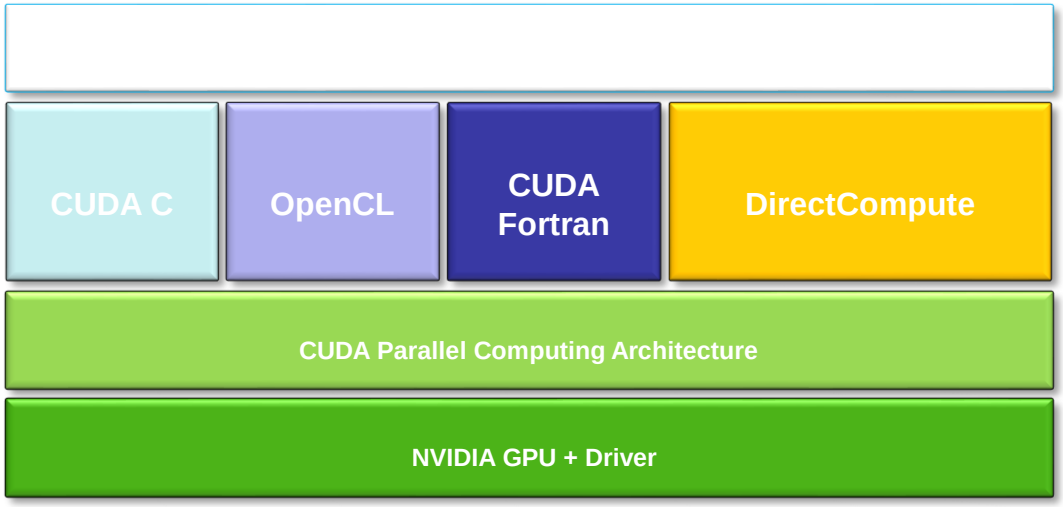
GPU hardware is specialized for problems which can be classified as *intense data-parallel computations*

the same set of operation is executed many times in parallel on different data
designed such that more transistors are devoted to data processing rather than data caching and flow control



"The GPU devotes more transistors to Data Processing"
(NVIDIA CUDA Programming Guide)

- Compute Unified Device Architecture (CUDA)
- a general purpose parallel computing platform and programming model that easy GPU programming, which provides:
  - a new hierarchical multi-threaded programming paradigm
  - a new architecture instruction set called PTX (Parallel Thread eXecution)
  - a small set of syntax extensions to higher level programming languages (C, Fortran) to express thread parallelism within a familiar programming environment
  - A complete collection of development tools to compile, debug and profile CUDA programs.



CUDA Programming Model

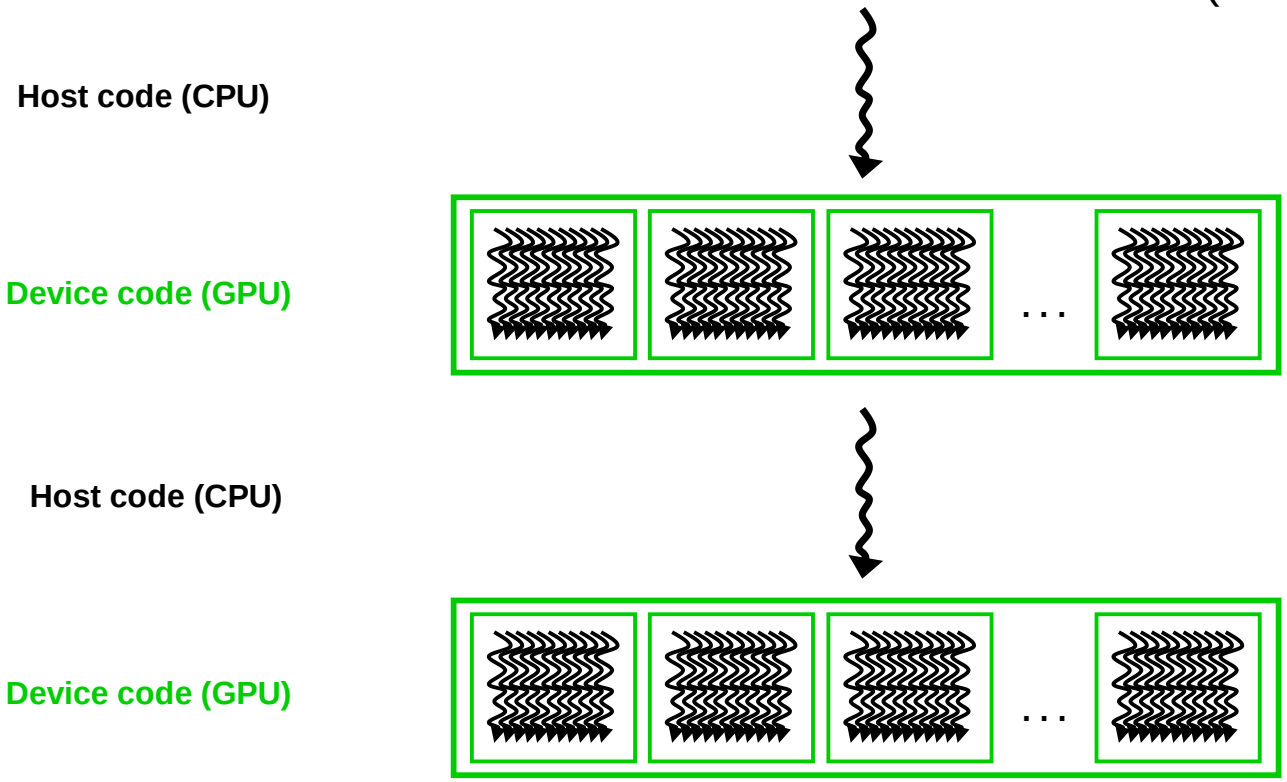
- GPU is seen as an auxiliary coprocessor with its own memory space
- *data-parallel*, *computational-intensive* portions of a program can be executed on the GPU

  each *data-parallel* computational portion can be isolated into a function, called CUDA kernel, that is executed on the GPU
  CUDA kernels are executed by many different threads in parallel
  each thread can compute different data elements independently
  the GPU parallelism is very close to the SPMD (Single Program Multiple Data) paradigm. Single Instruction Multiple Threads (SIMT) according to the Nvidia definition.
- GPU threads are extremely *light weight*

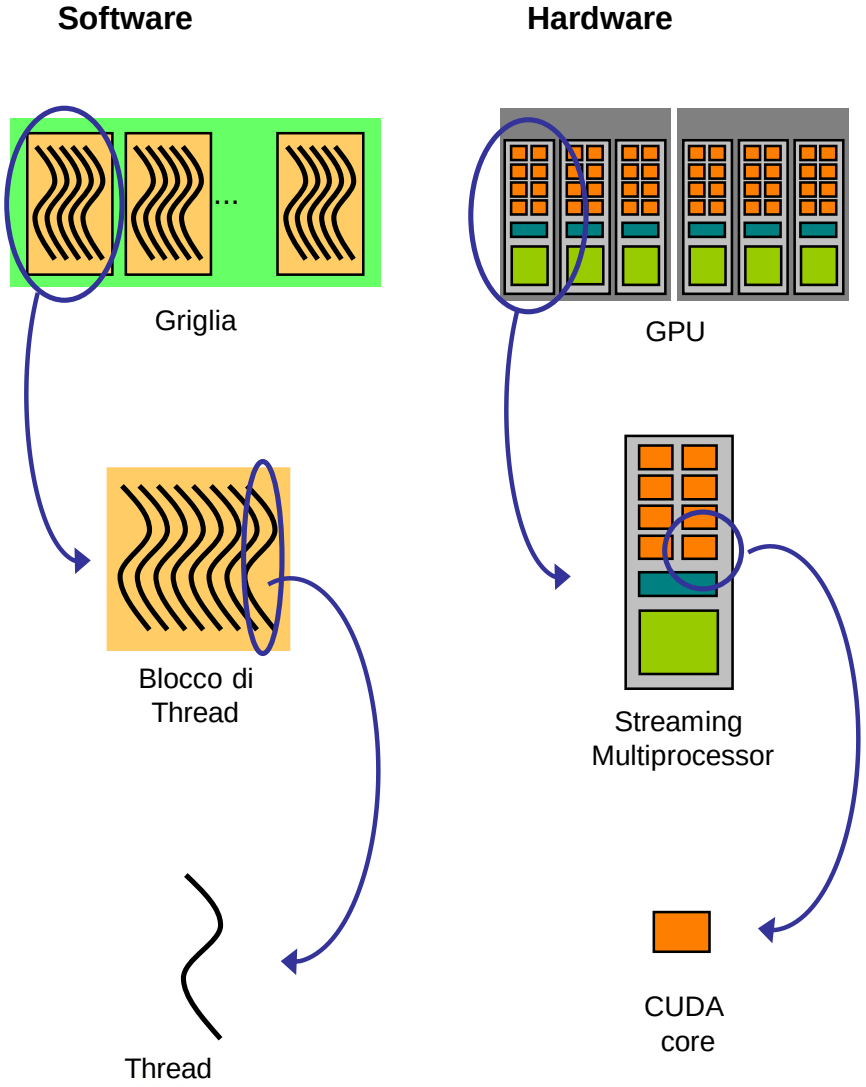
  no penalty in case of a *context-switch* (each thread has its own registers)
  the more are the threads *in flight*, the more the GPU hardware is able to hide memory or computational latencies, i.e better overall performances at executing the kernel function

CUDA Execution Model

- serial portions of a program, or those with low level of parallelism, keep running on the CPU (host)
- Data-parallel , computational intensive portions of the program are isolated into CUDA kernel function. The CUDA kernel are executed onto the GPU (device)



More on the CUDA Execution Model



when a CUDA kernel is invoked:
each thread block is assigned to a SM in a round-robin mode

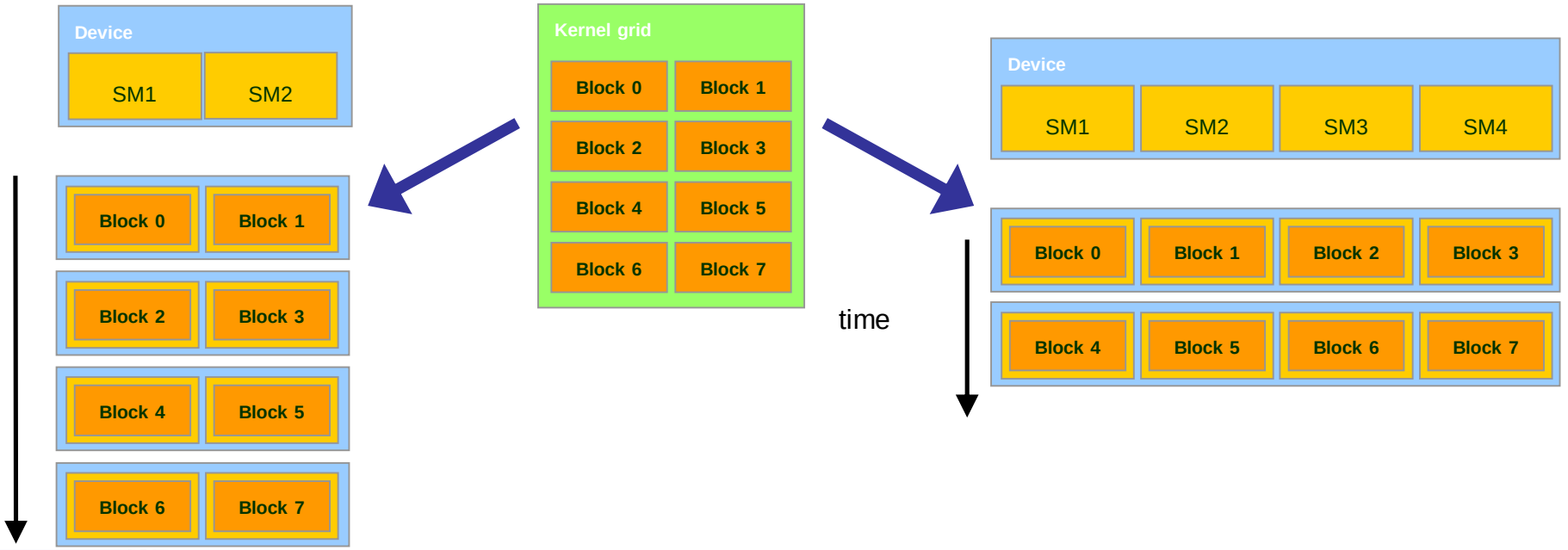
- a maximum number of blocks can be assigned to each SM, depending on hardware generation and on how many resorses each block needs to be executed (registers, shared memory, etc)
- the runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete the execution of blocks previously assigned to them.
- once a block is assigned to a SM, it remains on that SM until the work for all threads in the block is completed
- each block execution is independent from the other (no synchronization is possible among them)

thread of each block are partitioned into warps of 32 *thread* each, so to map each thread with a unique consecutive thread index in the block, starting from index 0.
the scheduler select for execution a warp from one of the residing blocks in each SM.
A warp execute one common instruction at a time

- each CUDA core take care of one thread in the warp
- fully efficiency when all threads agree on their execution path

Trasparent Scalability

CUDA runtime system can execute blocks in any order relative to each other.
This flexibility enables to execute the same application code on hardware with different numbers of SM



CUDA syntax extensions to the C language

- CUDA defines a small set of extensions to the high level language as the C in order to define the kernels and to configure the kernel execution.
- A CUDA kernel function is defined using the __global__ declaration
- when a CUDA kernel is called, it will be executed N times in parallel by N different CUDA threads on the device
- the number of CUDA threads that execute that kernel is specified using a new syntax, called kernel execution configuration
  - cudaKernelFunction **<<<...>>>** (arg_1, arg_2, ..., arg_n)
- each thread has a unique thread ID
  - the thread ID is accessible within the CUDA kernel through the built-in **threadIdx** variable
- the built-in variables **threadIdx** are a 3-component vector
  - use .x, .y, .z to access its components

A simple CUDA program

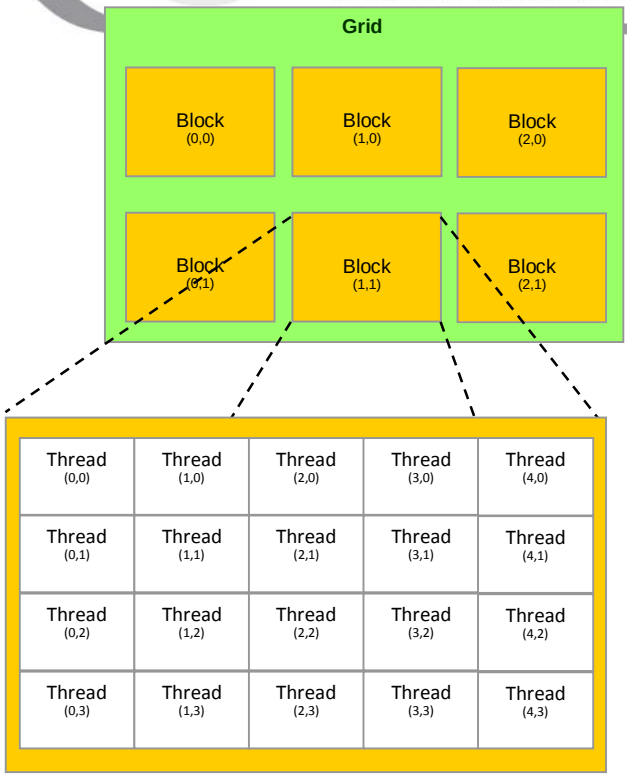

```
int main(int argc, char *argv[]) {
  int i;
  const int N = 1000;
  double u[N], v[N], z[N];

  initVector (u, N, 1.0);
  initVector (v, N, 2.0);
  initVector (z, N, 0.0);

  printVector (u, N);
  printVector (v, N);

  // z = u + v
  for (i=0; i<N; i++)
      z[i] = u[i] + v[i];

  printVector (z, N);

  return 0;
}
```

```
__global__
void gpuVectAdd( const double *u,
  const double *v, double *z)
{  // use GPU thread id as index
   i = threadIdx.x;
   z[i] = u[i] + v[i];
}
```

```
int main(int argc, char *argv[]) {
  ...

  // z = u + v
  {
    // run on GPU using
    // 1 block of N threads in 1D
    gpuVectAdd <<<1,N>>> (u, v, z);
  }

  ...
}
```

CUDA Threads

- threads are organized into blocks of threads
  - blocks can be 1D, 2D, 3D sized in threads
  - blocks can be organized into a 1D, 2D, 3D grid of blocks
  - each block of threads will be executed independently
  - no assumption is made on the blocks execution order
- each block has a unique block ID
  - the block ID is accessible within the CUDA kernel through the built-in **blockIdx** variable
- The built-in variable **blockIdx** is a 3-component vector
  - use .x, .y, .z to access its components



`blockIdx`:
block coordinates inside the grid

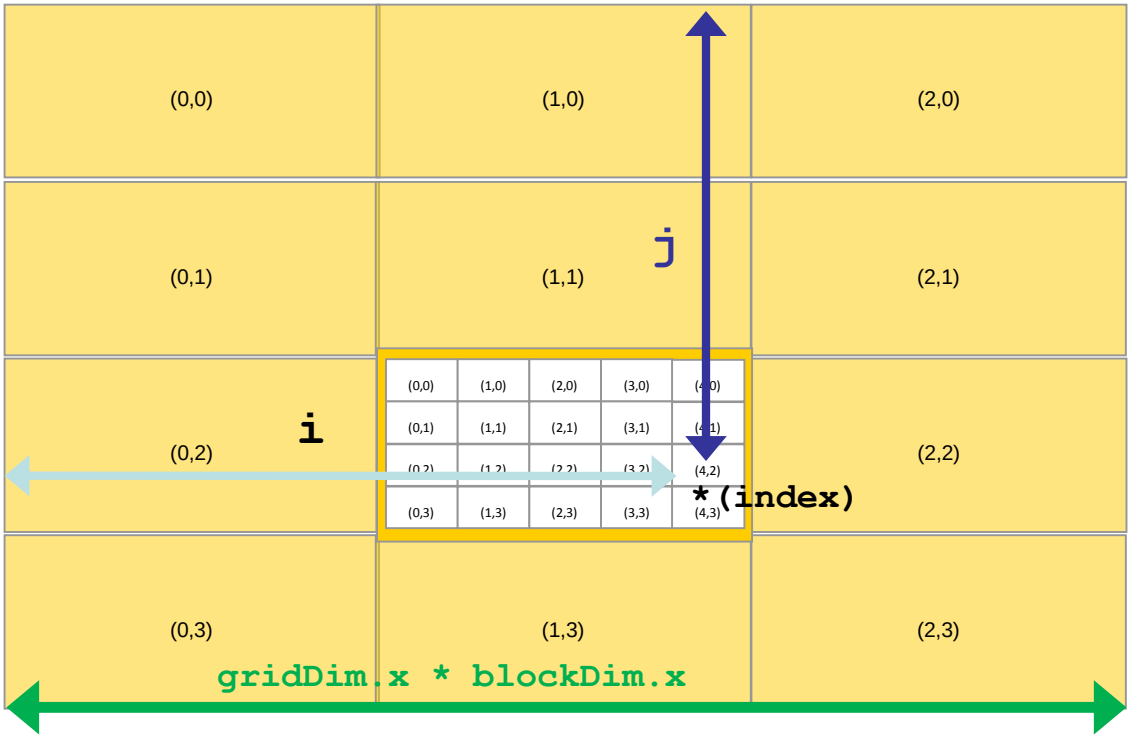
`blockDim`:
block dimensions in thread units

`gridDim`:
grid dimensions in block units

Simple 1D CUDA vector add


```
__global__
void gpuVectAdd( int N, const double *u, const double *v, double *z)
{
    // use GPU thread id as index
    index = blockIdx.x * blockDim.x + threadIdx.x;

    // check out of border access
    if ( index < N ) {
       z[index] = u[index] + v[index];
    }
}

int main(int argc, char *argv[]) {
   ...

   // use 1D block threads
   dim3 blockSize = 512;

   // use 1D grid blocks
   dim3 gridSize = (N + blockSize-1) / blockSize.x;

   gpuVectAdd <<< gridSize,blockSize >>> (N, u, v, z);
   ...
}
```

Composing 2D CUDA Thread Indexing



`threadIdx`:
thread coordinates inside a block

`blockIdx`:
block coordinates inside the grid

`blockDim`:
block dimensions in thread units

`gridDim`:
grid dimensions in block units

```
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;

index = j * gridDim.x * blockDim.x + i;
```

2D array element-wise add (matrix add)

As an example, the following code adds two matrices *A* and *B* of size *NxN* and stores the result into the matrix *C*

```
__global__ void matrixAdd(int N, const float *A,  const float *B, float *C) {
  int i = blockIdx * blockDim.x + threadIdx.x;
  int j = blockIdx * blockDim.y + threadIdx.y;

  // matrix elements are organized in row major order in memory
  int index = i * N + j;

  C[index] = A[index] + B[index];
}

int main(int argc, char *argv[]) {
   ...

  // add NxN matrices on GPU using 1 block of NxN threads
  matrixAdd <<< 1, N >>> (N, A, B, C);
  ...
}
```

Memory allocation on GPU device

- CUDA API provides functions to manage data allocation on the device global memory:

- cudaMalloc(void** bufferPtr, size_t n)

  It allocates a buffer into the device global memory
  The first parameter is the address of a generic pointer variable that must point to the allocated buffer
  it must be cast to (void**)!
  The second parameter is the size in bytes of the buffer to be allocated

- cudaFree(void* bufferPtr)

  It frees the storage space of the object

Memory Initialization on GPU device

- cudaMemset(void* devPtr, int value, size_t count)
  - It fills the first count bytes of the memory area pointed to by devPtr with the constant byte of the int value converted to unsigned char.

    it's like the standard library C memset() function
    devPtr - Pointer to device memory
    value - Value to set for each byte of specified memory
    count - Size in bytes to set
- To initialize an array of double (float, int, …) to a specific value you need to execute a CUDA kernel.

Memory copy between CPU and GPU

- cudaMemcpy(void *dst, void *src, size_t size, direction)

  dst: destination buffer pointer
  src: source buffer pointer
  size: number of bytes to copy
  direction: macro name which defines the direction of data copy
  - from CPU to GPU: cudaMemcpyHostToDevice (H2D)
  - from GPU to CPU: cudaMemcpyDeviceToHost (D2H)
  - on the same GPU: cudaMemcpyDeviceToDevice

  the copy begins only after all previous kernel have finished
  the copy is blocking: it prevents CPU control to proceed further in the program until last byte has been transfered
  returns only after copy is complete

Three steps for a CUDA porting

1. **identify data-parallel, computational intensive portions**

   isolate them into functions (CUDA kernels candidates)
   identify involved data to be moved between CPU and GPU

2. **translate identified CUDA kernel candidates into real CUDA kernels**

   choose the appropriate thread index map to access data
   change code so that each thead acts on its own data

3. **modify code in order to manage memory and kernel calls**

   allocate memory on the device
   transfer needed data from host to device memory
   insert calls to CUDA kernel with execution configuration syntax
   transfer resulting data from device to host memory

Vector Sum

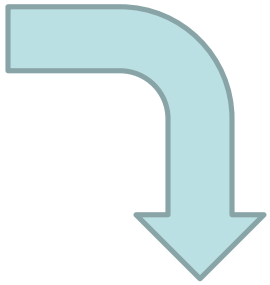
1. identify data-parallel computational intensive portions


```
int main(int argc, char *argv[]) {
  int i;
  const int N = 1000;
  double u[N], v[N], z[N];

  initVector (u, N, 1.0);
  initVector (v, N, 2.0);
  initVector (z, N, 0.0);

  printVector (u, N);
  printVector (v, N);

  // z = u + v
  for (i=0; i<N; i++)
      z[i] = u[i] + v[i];


  printVector (z, N);

  return 0;
}
```

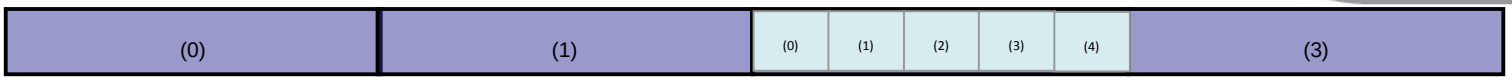
```
program vectoradd
integer :: i
integer, parameter :: N=1000
real(kind(0.0d0)), dimension(N):: u, v, z

call initVector (u, N, 1.0)
call initVector (v, N, 2.0)
call initVector (z, N, 0.0)

call  printVector (u, N)
call  printVector (v, N)

! z = u + v
do i = 1,N
    z(i) = u(i) + v(i)
end do

call printVector (z, N)

end program
```

- **each *thread* execute the same kernel, but act on different data**:
  - turn the loop into a CUDA kernel function
  - map each CUDA *thread* onto a unique index to access data
  - let each *thread* retrieve, compute and store its own data using the unique address
  - prevent out of border access to data if data is not a multiple of thread block size

```
const int N = 1000;
double u[N], v[N], z[N];

// z = u + v
for (i=0; i<N; i++)
    z[i] = u[i] + v[i];
```



```
__global__ void gpuVectAdd (int N, const double *u, const double *v, double *z)
{
  // index is a unique identifier for each GPU thread
  int index =  blockIdx * blockDim.x + threadIdx.x ;
  if (index < N)
     z[index] = u[index] + v[index];
}
```




```
__global__ void gpuVectAdd (int N, const double *u, const double *v, double *z)
{
    // index is a unique identifier of each GPU thread
    int index =  blockIdx.x * blockDim.x + threadIdx.x  ;
    if (index < N)
      z[index] = u[index] + v[index];
}
```

The **__global__** qualifier declares a CUDA kernel

CUDA kernels are special C functions:

- can be called from host only
- must be called using the *execution configuration* syntax
- *the return type must be void*
- they are asynchronous: control is returned immediately to the host code
- an explicit synchronization is needed in order to be sure that a CUDA kernel has completed the execution

Insert calls to CUDA kernels using the execution configuration syntax:

kernelCUDA<<<numBlocks,numThreads>>>(...)

specifying the thread/block hierarchy you want to apply:

- **numBlocks:** specify grid size in terms of thread blocks along each dimension
- **numThreads:** specify the block size in terms of threads along each dimension

```
dim3 numThreads(32);
dim3 numBlocks( ( N + numThreads - 1 ) / numThreads.x );
gpuVectAdd<<<numBlocks, numThreads>>>( N, u_dev, v_dev, z_dev );
```

```
type(dim3) :: numBlocks, numThreads
numThreads = dim3( 32, 1, 1 )
numBlocks = dim3( (N + numThreads.x - 1) / numThreads.x, 1, 1 )
call gpuVectAdd<<<numBlocks,numThreads>>>( N, u_dev, v_dev, z_dev )
```

Heterogeneous High Performance Programming framework

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

HPC wire

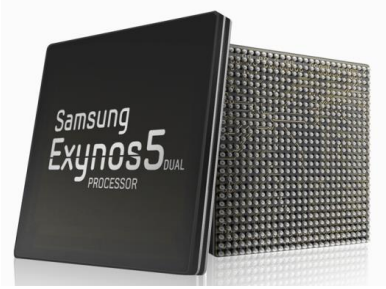
“As the two major programming frameworks for GPU computing, OpenCL and CUDA have been competing for mindshare in the developer community for the past few years. Until recently, CUDA has attracted most of the attention from developers, especially in the high performance computing realm. But OpenCL software 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?



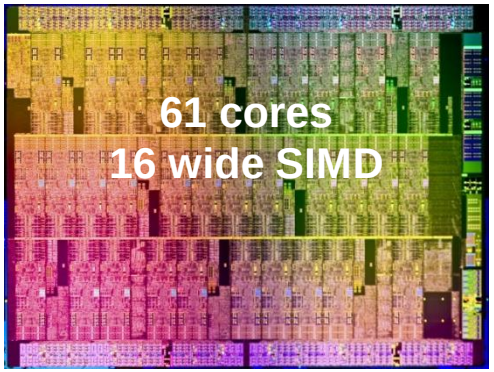
E.g. Samsung® Exynos 5:

- Dual core ARM A15 1.7GHz, Mali T604 GPU

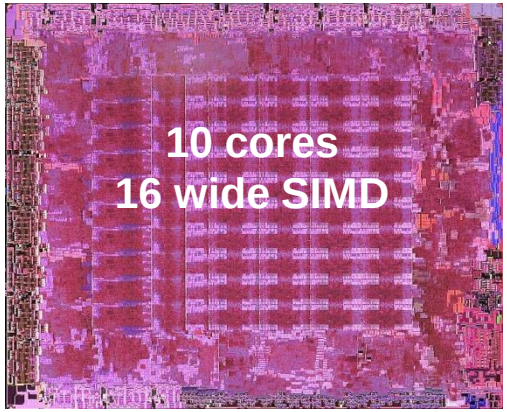
OpenCL lets Programmers write a single portable program that uses ALL resources in the heterogeneous platform

Microprocessor trends

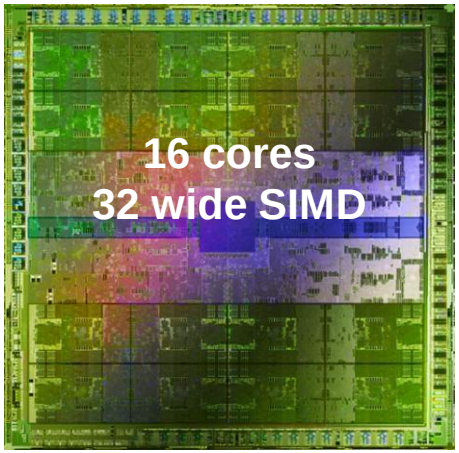
Individual processors have many (possibly heterogeneous) cores.



Intel® Xeon Phi™ coprocessor



ATI™ RV770

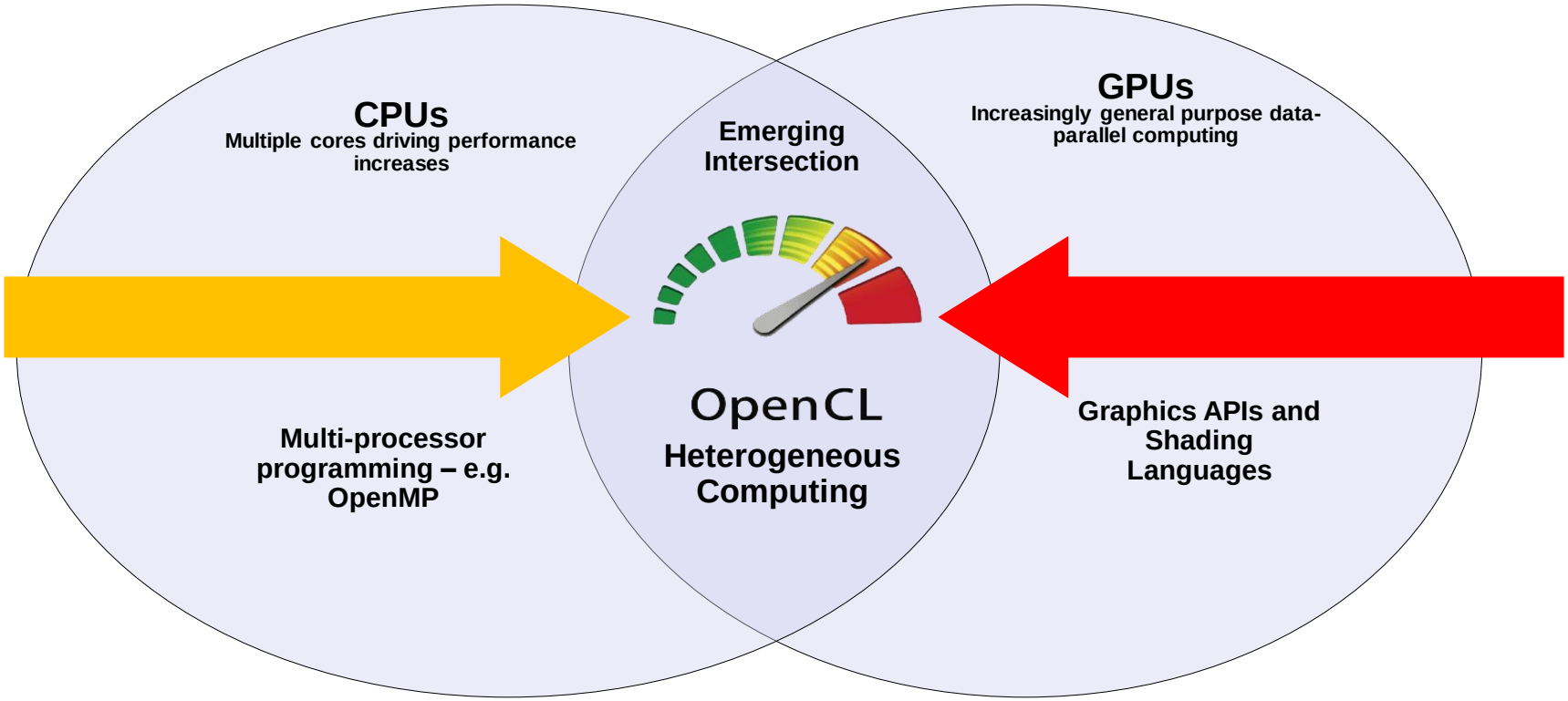


NVIDIA® Tesla® C2090

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



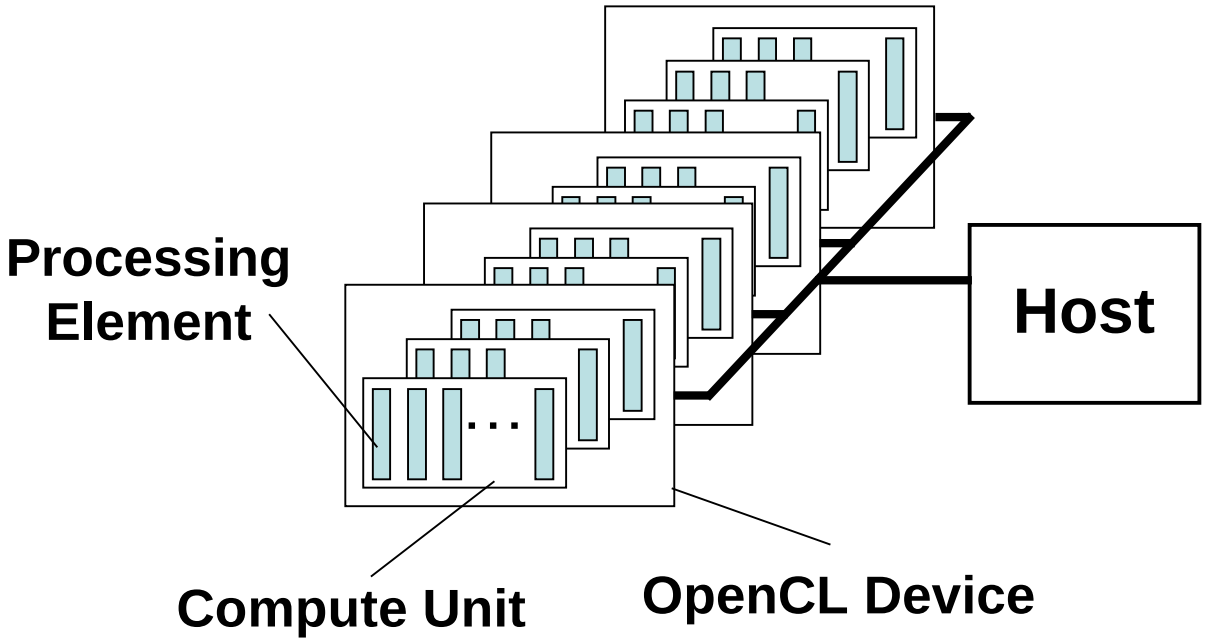
OpenCL – Open Computing Language

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

OpenCL 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.

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

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 – 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.

Vector Addition – Host Program

Summer School on PARALLEL COMPUTING

```
// 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                                                  Y |
    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                                        LL,NULL);

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

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0]
                  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
                  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.

CINECA

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

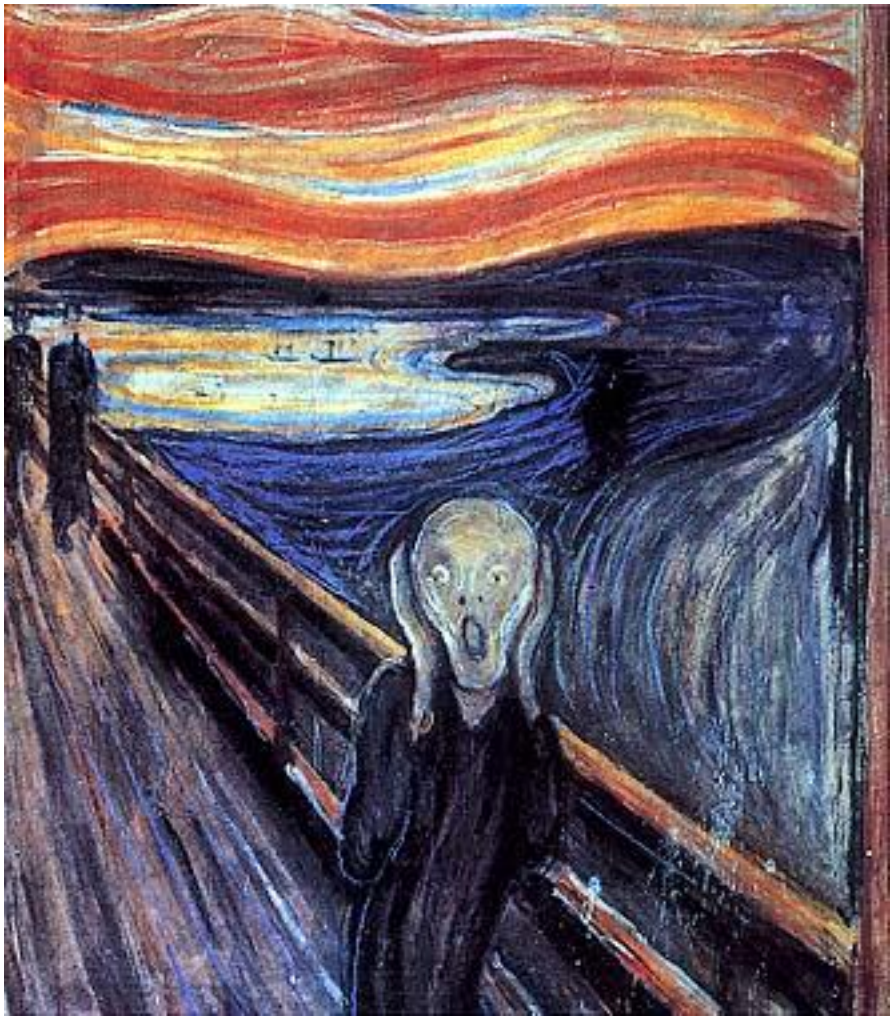
- Built-in functions — ***mandatory***
  - Work-Item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions
  - printf (v1.2 only, so not currently for NVIDIA GPUs)
- Built-in functions — ***optional*** (called “extensions”)
  - Double precision, atomics 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.



PORTING CUDA TO OPENCL

- 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

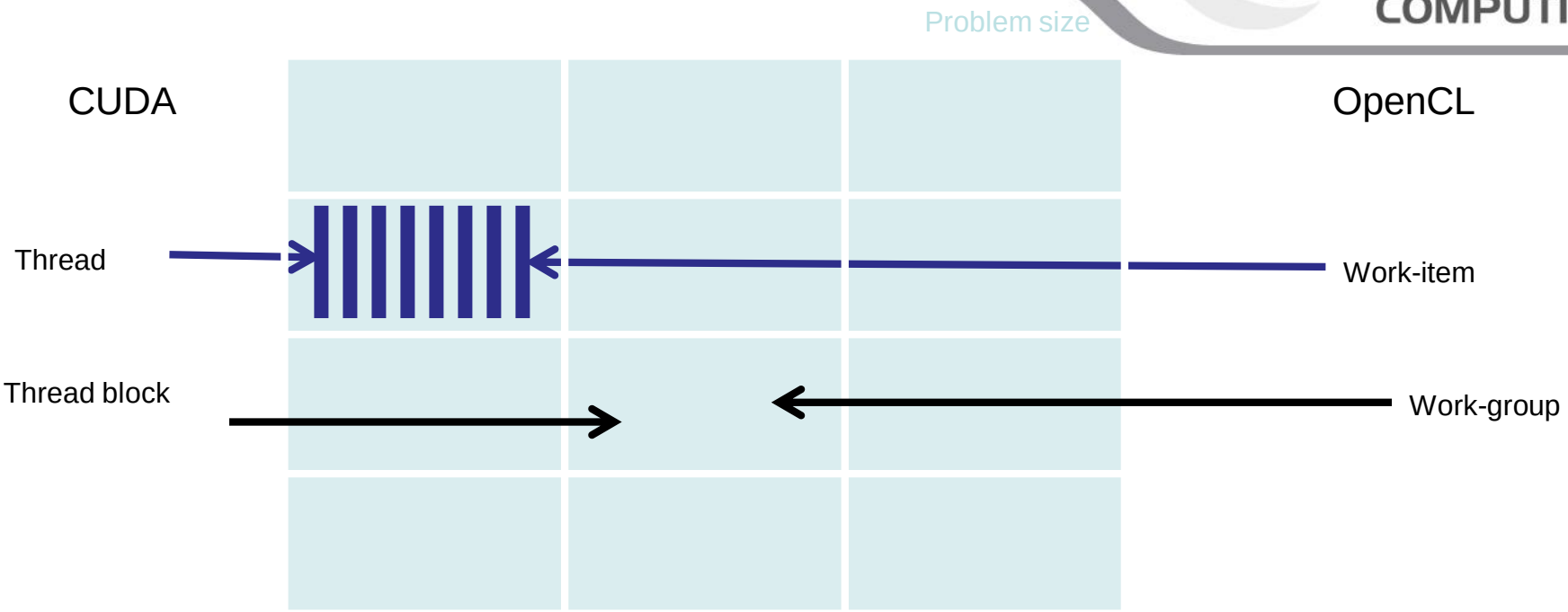
1. Have the kernel accept a local array as an argument

```
__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

NVIDIA Tesla K20

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

Intel MIC Xeon Phi

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

Setting up OpenCL on 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

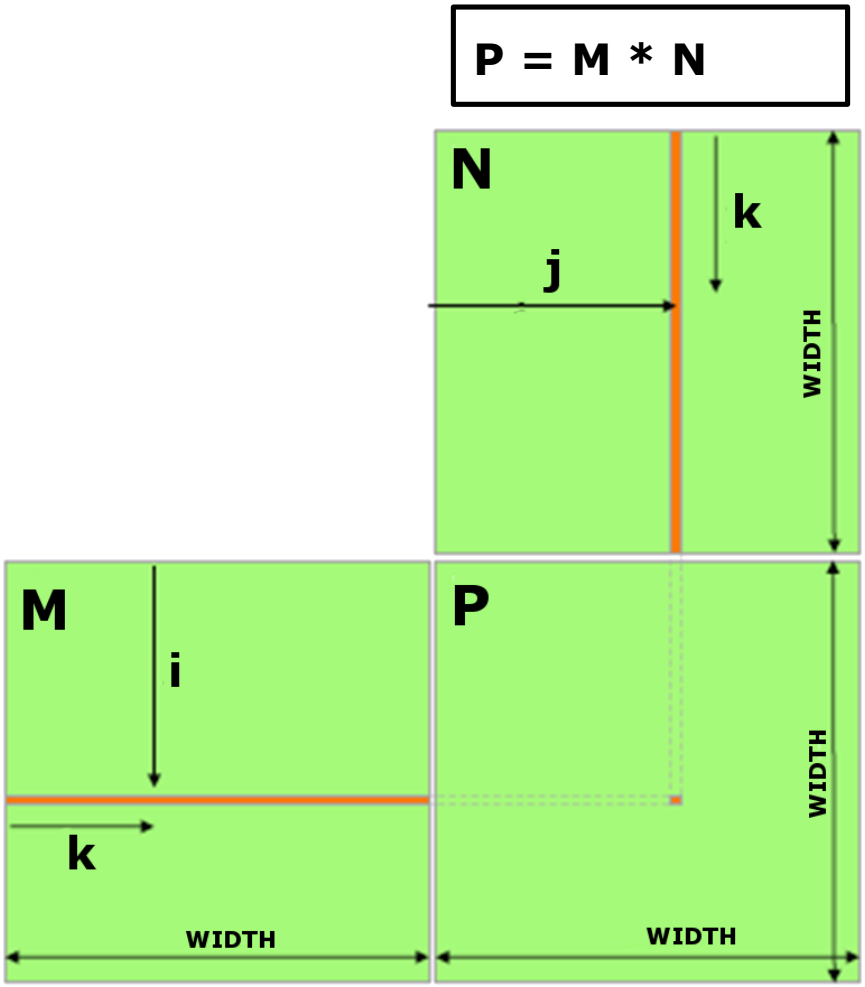
Results are OK no matter what performances

Exercise

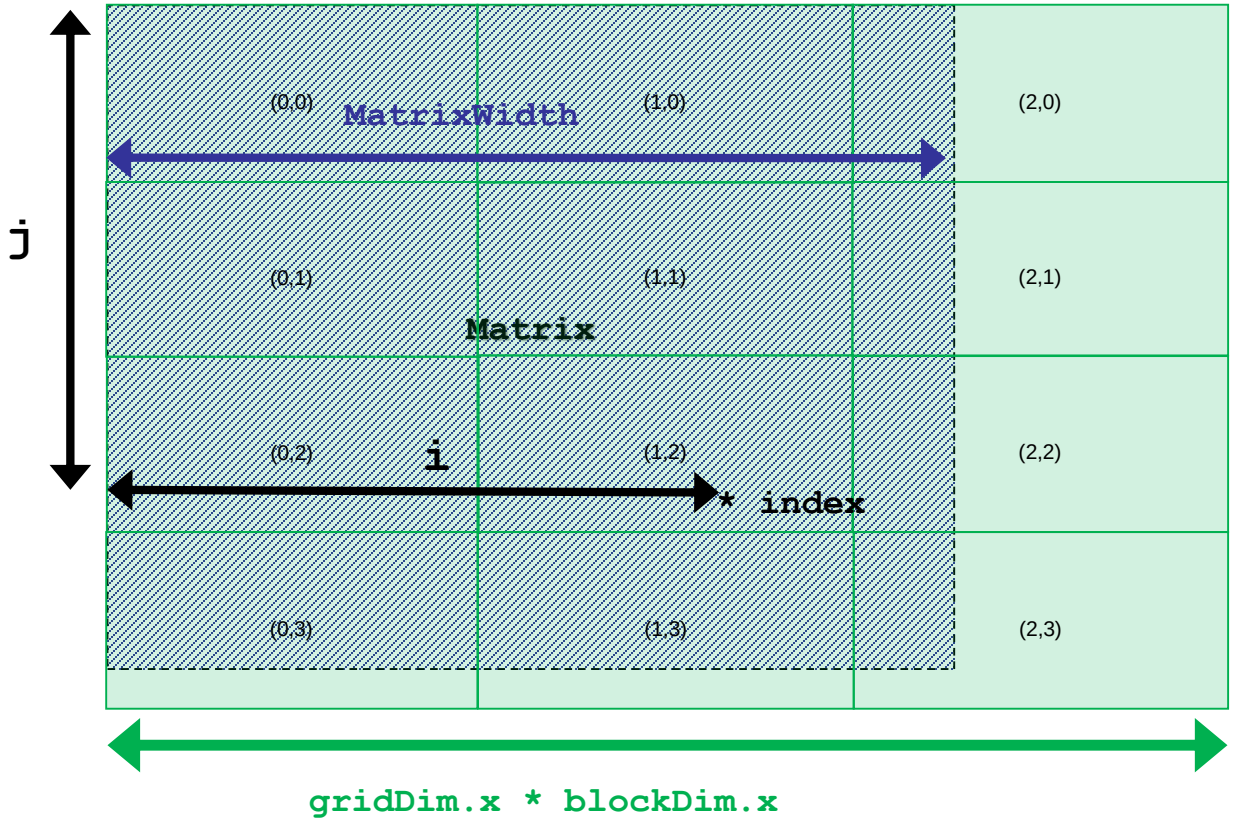
- Goal:
  - To inspect and verify that you can run an OpenCL kernel on Eurora machines
- Procedure:
  - Take the provided C ***vadd.c*** and ***vadd.cl*** source programs from VADD directory
  - Compile and link ***vadd.c***
  - Run on NVIDIA or Intel platform.
- Expected output:
  - A message verifying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



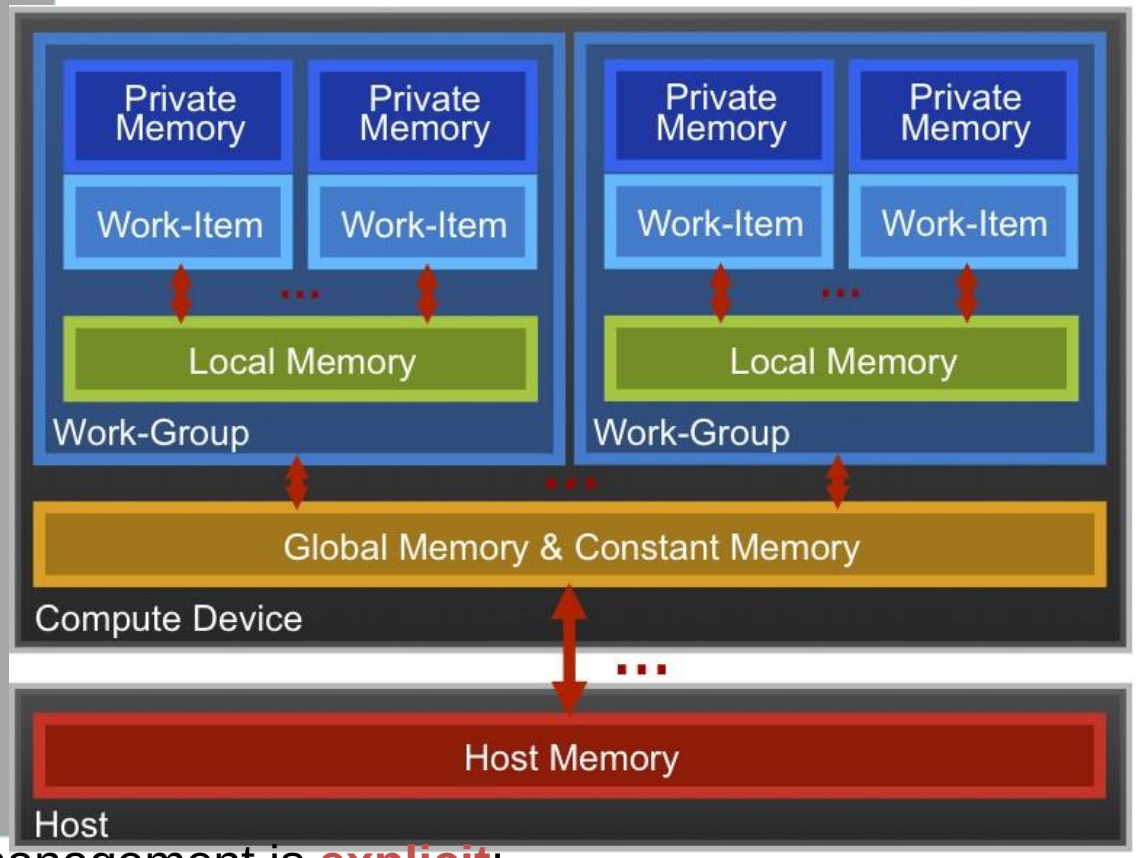

```
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;

index = j * MatrixWidth + i;
```

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

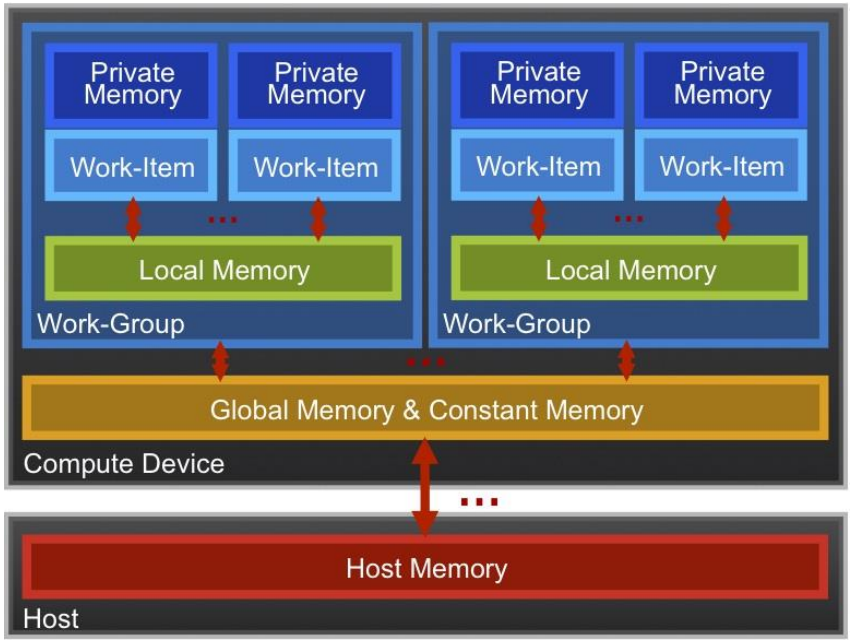
- Private Memory
  - Per work-item
- Local Memory
  - Shared within a work-group
- Global/Constant Memory
  - Visible to all work-groups
- Host memory
  - On the CPU



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

OpenCL Memory model

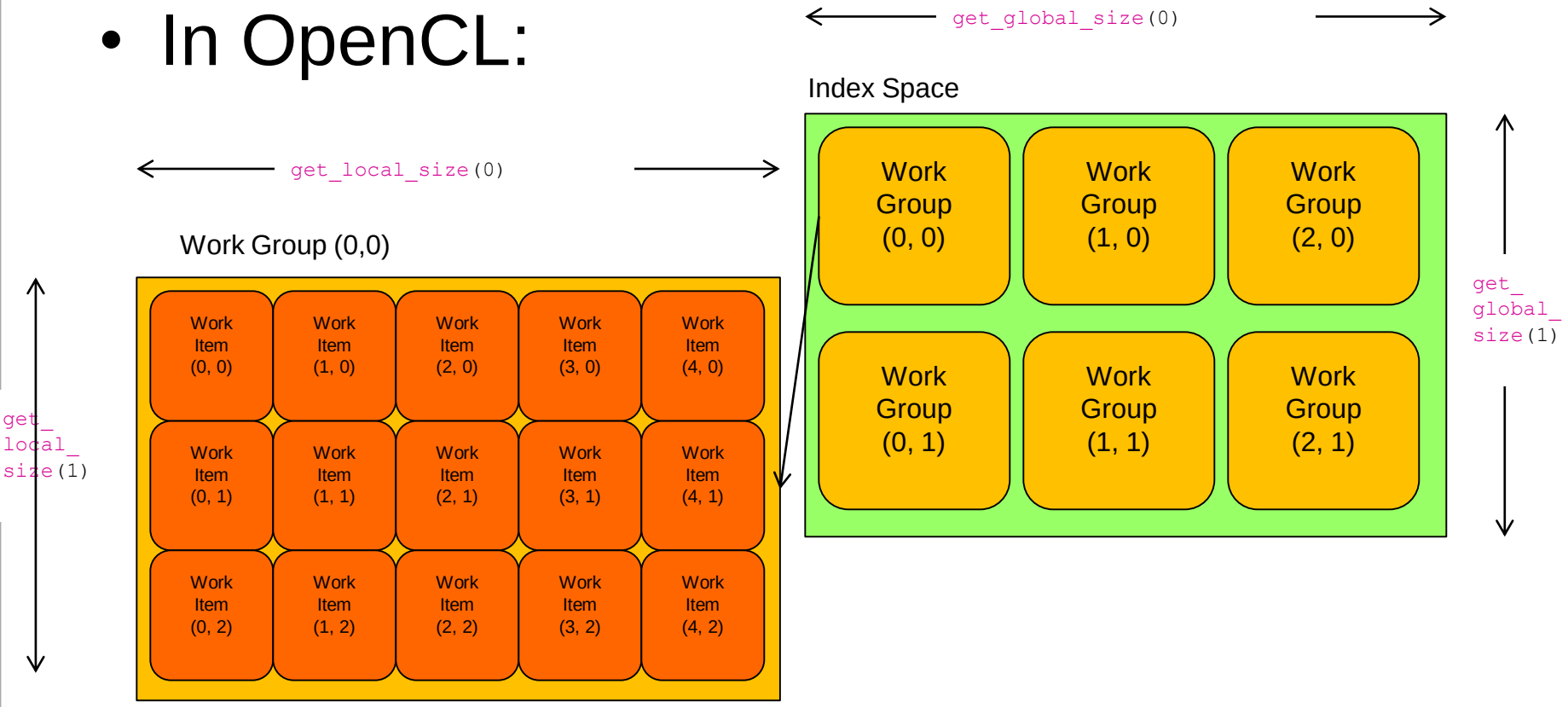
- Private Memory
  - Fastest & smallest: O(10) words/WI
- Local Memory
  - Shared by all WI's in a work-group
  - But not shared between work-groups!
  - O(1-10) Kbytes per work-group
- Global/Constant Memory
  - O(1-10) Gbytes of Global memory
  - O(10-100) Kbytes of Constant memory
- Host memory
 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



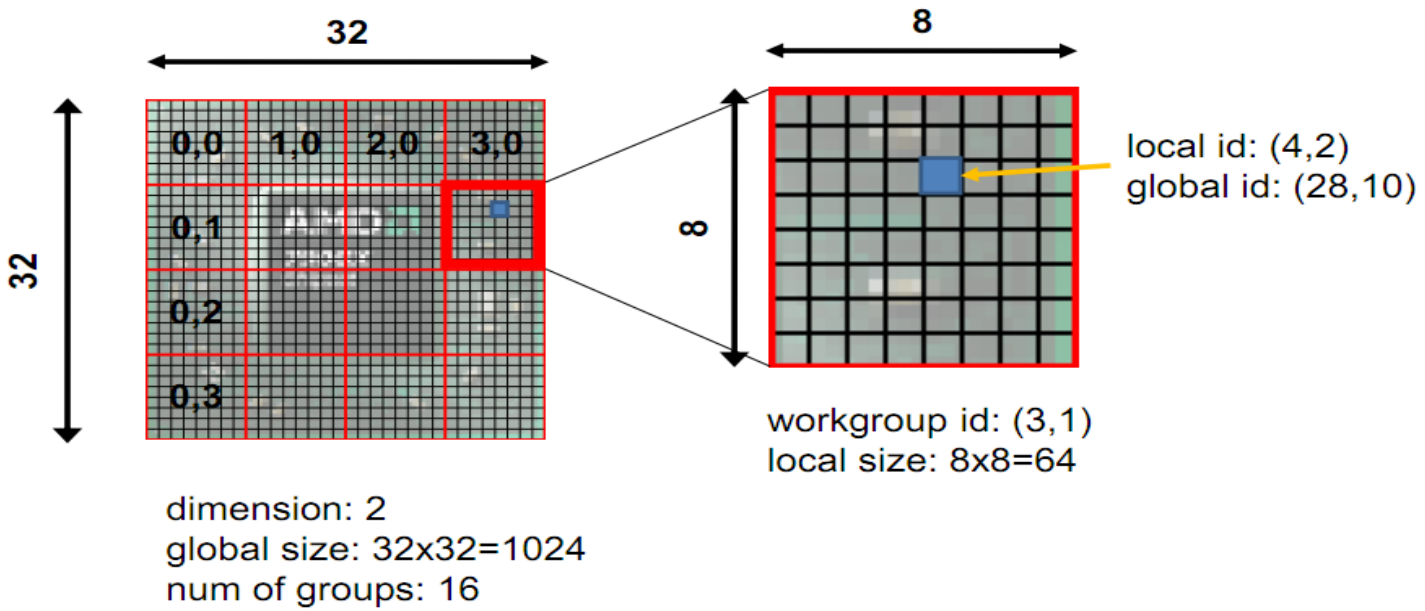
Memory management is **explicit**:
O(1-10) Gbytes/s bandwidth to discrete GPUs for
Host <-> Global transfers

OpenCL mapping

- In OpenCL:



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: 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];
    }
}
```

Matrix multiplication: OpenCL kernel improved

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

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

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

```
{
  int k;
  int j = get_global_id(0);
  int i = get_global_id(1);
  float tmp = 0.0f;
  for (k = 0; k < Pdim; k++)
   tmp += A[i*Ndim+k]*B[k*Pdim+j];
  }
  C[i*Ndim+j] += tmp;
}
```

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

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

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

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!

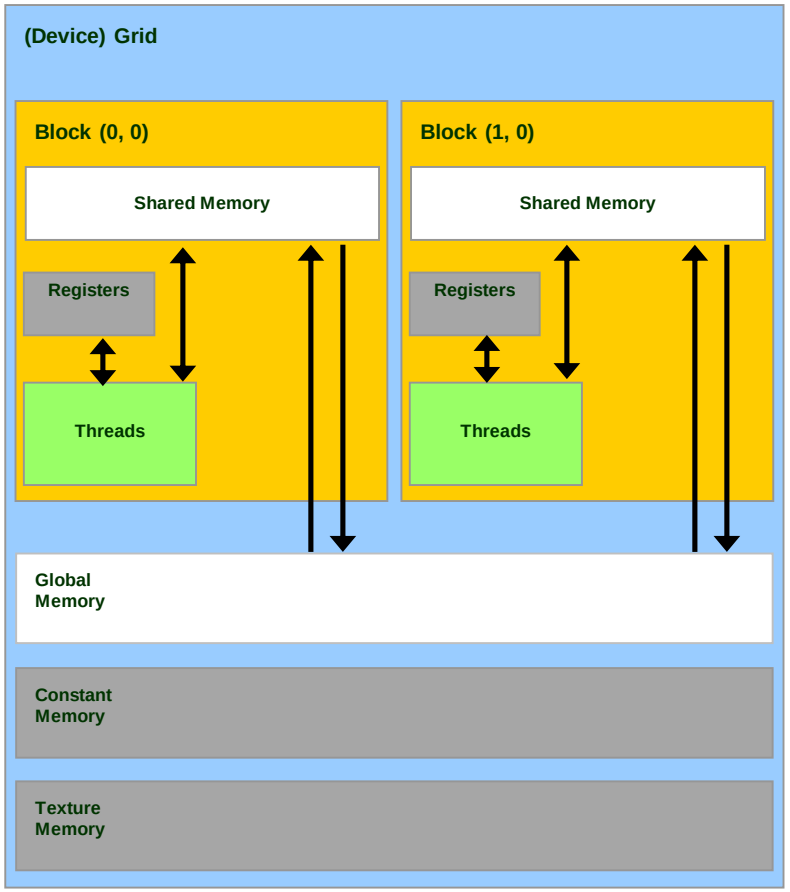
Using *Local/Shared Memory* for Thread Cooperation

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

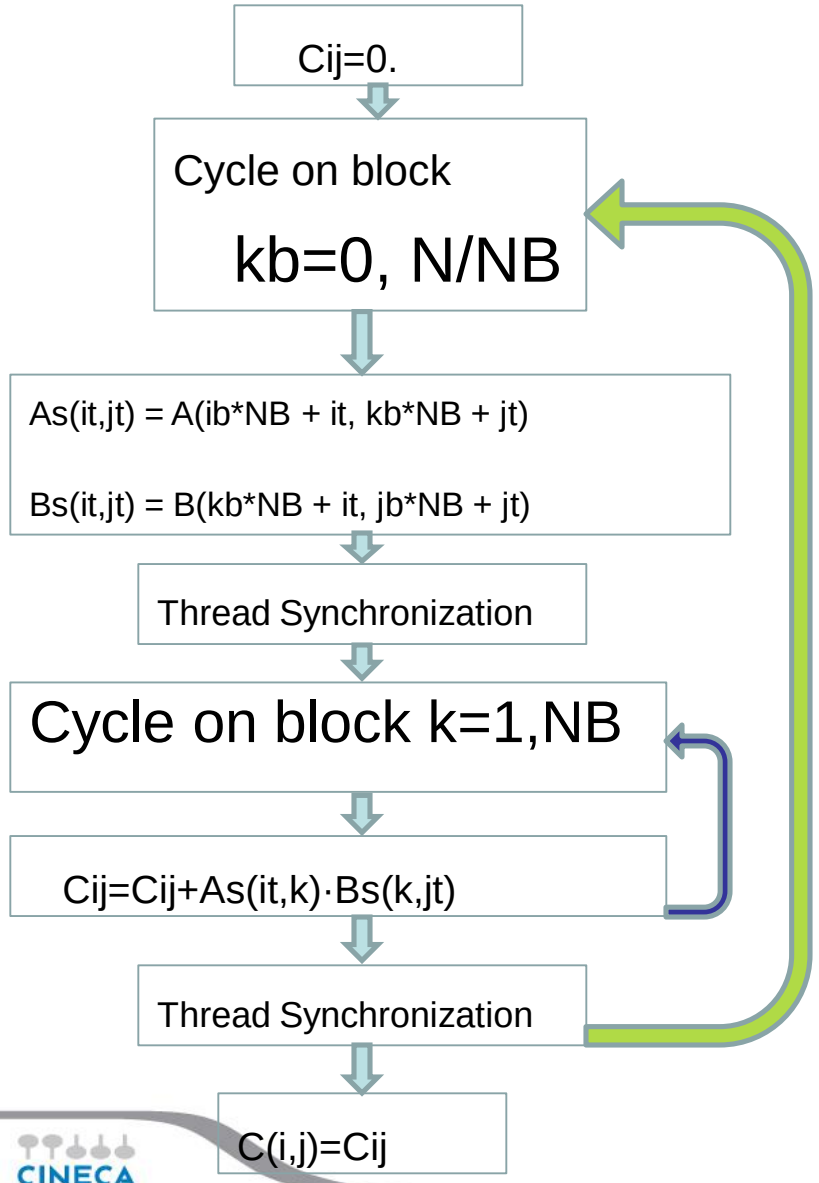
if a thread needs some data which has been already retrived by another thread in the same block, this data can be shared using the shared memory

Typical Shared Memory usage:

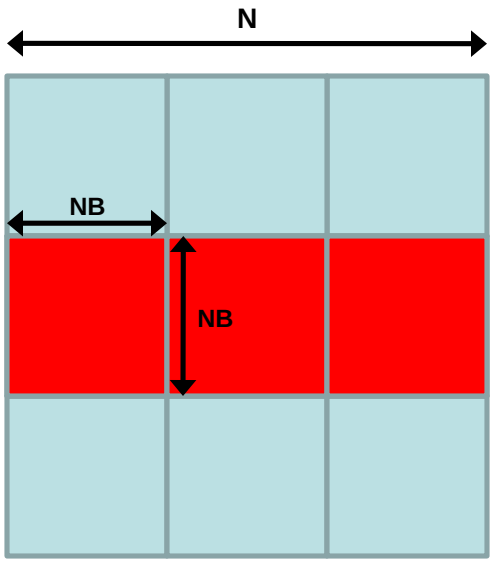
1. declare a buffer residing on shared memory (this buffer is per block)
2. load data into shared memory buffer
3. synchronize threads so to make sure all needed data is present in the buffer
4. performe operation on data
5. synchronize threads so all operations have been performed
6. write back results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

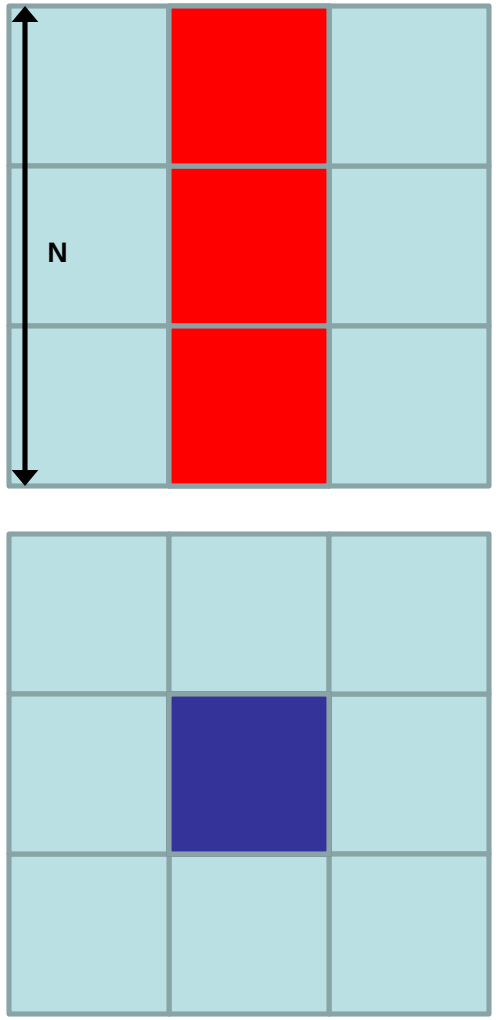


Matrix-matrix using Shared Memory




```
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;

}
```

Matrix-matrix using Shared Memory: OpenCL Kernel

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

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

  // Block row and column
  int ib = get_group_id(1);
  int jb = get_group_id(0);

  // Thread row and column within Csub
  int it = get_local_id(1);
  int jt = get_local_id(0);

  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
    barrier(CLK_LOCAL_MEM_FENCE);

    // 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
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  // 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;

}
```

Matrix-matrix using Shared Memory: OpenCL Kernel


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

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

  // Block row and column
  int ib = get_group_id(1);
  int jb = get_group_id(0);

  // Thread row and column within Csub
  int it = get_local_id(1);
  int jt = get_local_id(0);

  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
  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
    barrier(CLK_LOCAL_MEM_FENCE);

    // 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
    barrier(CLK_LOCAL_MEM_FENCE);
```

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).

- 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

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?

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?

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?

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?

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