Introduction to GPU Accelerators and CUDA Programming

25th Summer School on Parallel Computing

11-22 July 2016

Sergio Orlandini
s.orlandini@cineca.it

CINECA SCAI
SuperComputing Applications and Innovation

Agenda

Morning:

- Introduction to GPGPU
- GPU architectures
- GPU programming model
- Data transfers CPU/GPU

--- break ---

- Compiling CUDA programs
- Error Checking
- Measuring Performances
- Hands on

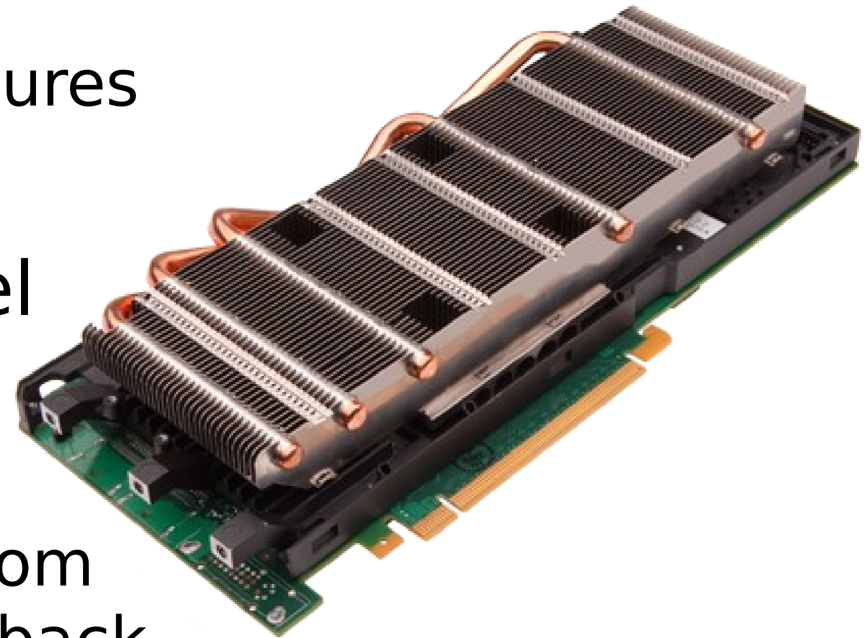
Afternoon:

- GPU Memory Hierarchy
- Concurrency
- CPU-GPU Interaction
- Working with multi-GPU
- Hands on

--- break ---

- CUDA-Toolkit
- CUDA Enabled Libraries
- OpenACC introduction
- Hands on

- Introducion to GPGPU
  - CPU vs GPGPU architecture
  - Programming approaches
  - nVIDIA GPU HPC architectures



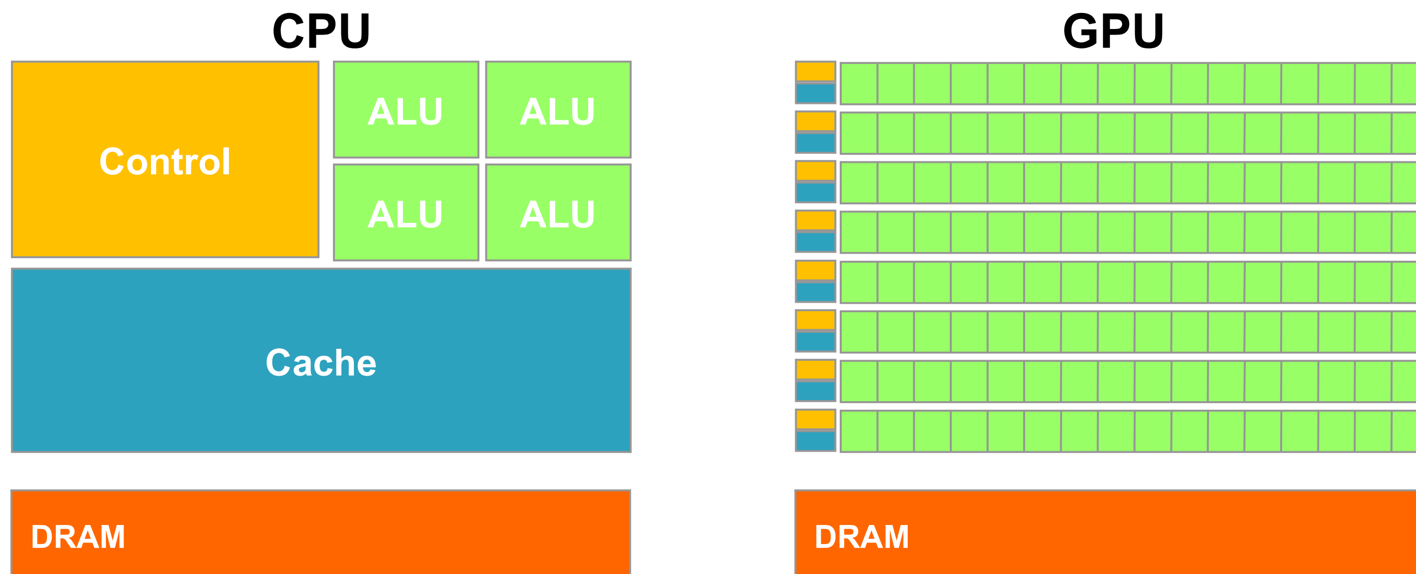
- GPU programming model
  - Thread indexing
  - Vector-Vector Add
  - handling data transfers from CPU to GPU memory and back
  - write and launch a CUDA program

What is a GPU ?

- **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



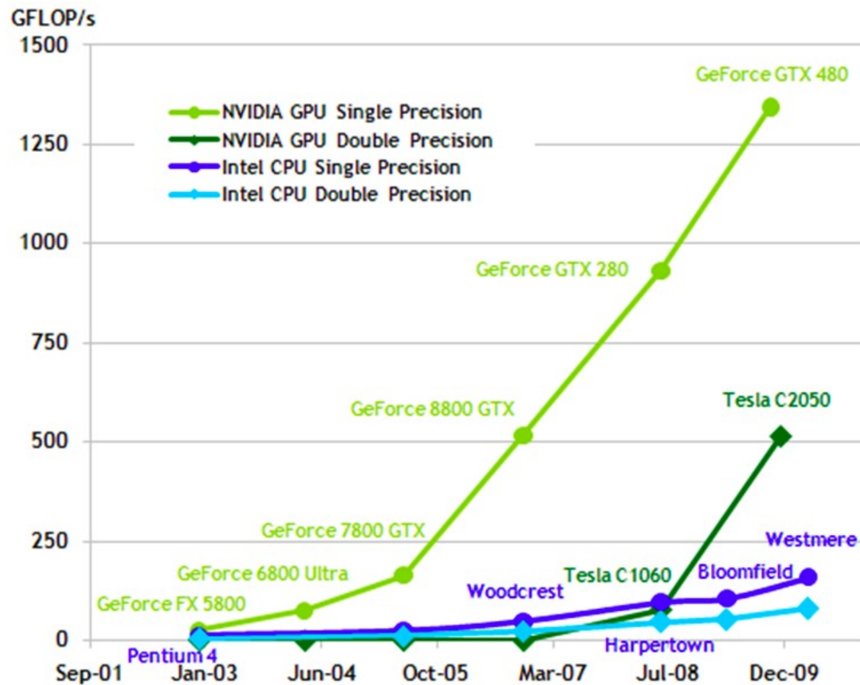
"The GPU devotes more transistors to Data Processing"
(NVIDIA CUDA Programming Guide)

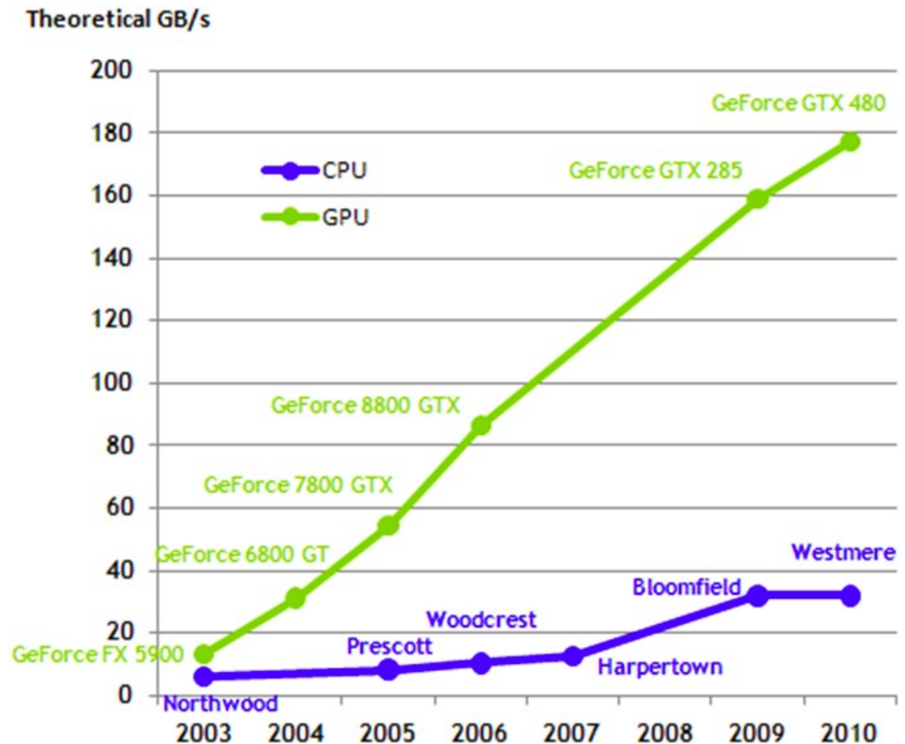
CINECA SCAI SuperComputing Applications and Innovation

The concurrency revolution

A new direction in microprocessor design roadmaps

- CPU vendors tend to increase the computational power of single processing unit by increasing the working frequency and adding more higher level control logic and pipelines
- GPU increased the number of processing units, less logic, lowering frequency and dropping down power consumption



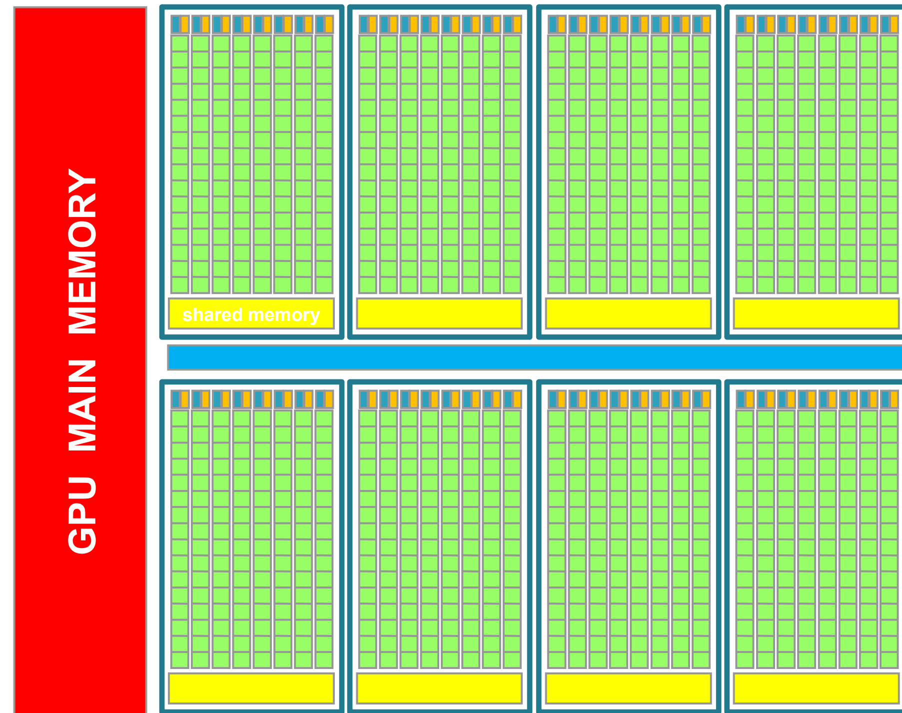


Peak GFlops (left) and bandwidth (right) trends of some nVIDIA GPU compared to Intel CPU products

CINECA SCAI
SuperComputing Applications and Innovation

GPU Architectures

- A typical GPU architecture consists of:
  - Main global memory
    - high bandwidth
  - Streaming Processor
    - grouping independent cores and control units
- Each SM unit has
  - Many ALU cores
  - Instruction scheduler dispatchers
  - Shared memory with very fast access to data



SCAI CINECA SuperComputing Applications and Innovation

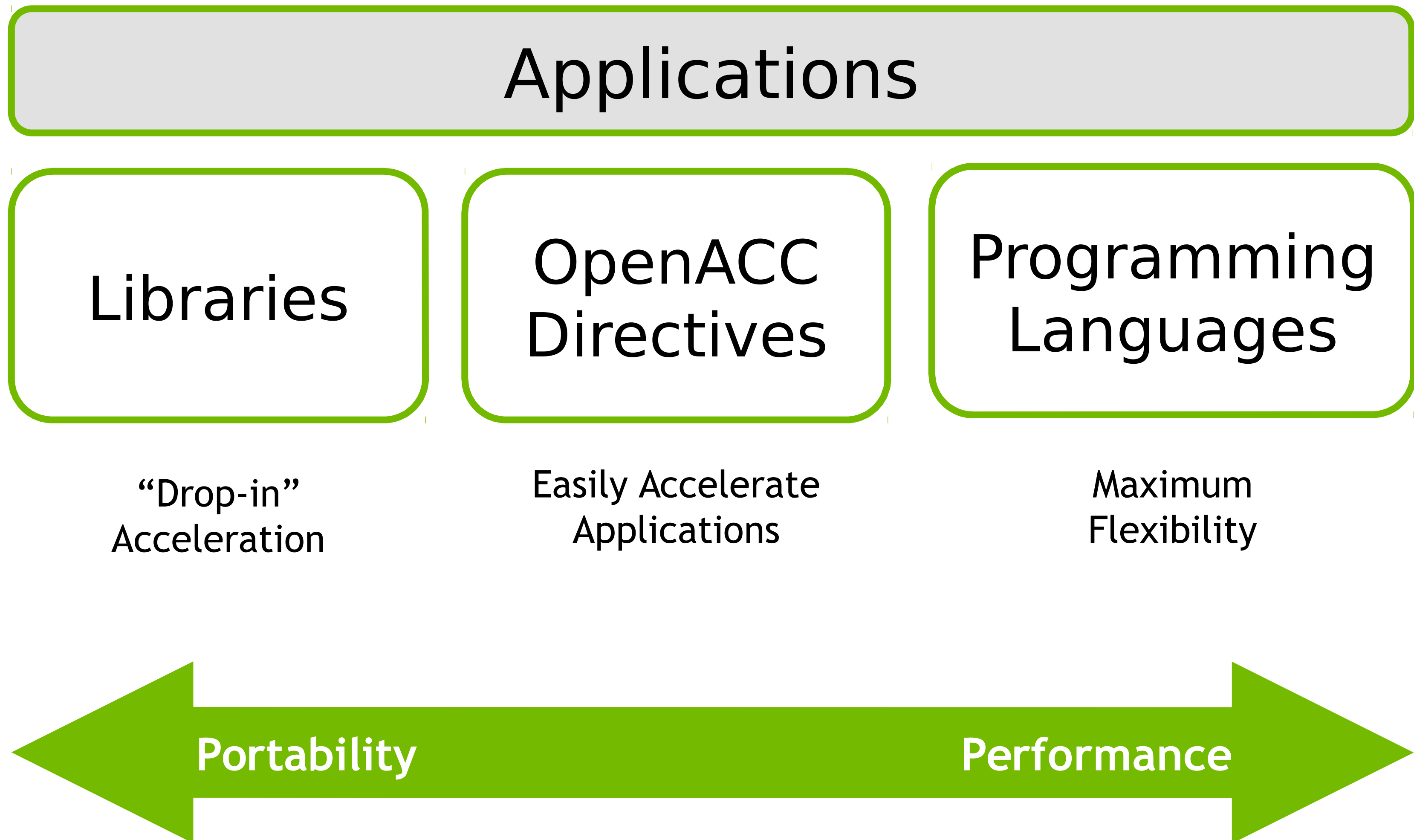
GPGPU (General Purpose GPU) and GPU computing

- many applications that process large data sets can use a data-parallel programming model to speed up the computations
- many algorithms outside the field of image rendering are accelerated by data-parallel processing
- ... so why not using GPU power for applications out of the 3D graphics domain?
- many attemps where made by brave programmers and researchers in order to force GPU APIs to threat their scientific data (atoms, signals, events, etc) as pixel or vertex in order to be computed by the GPU.
- not many survived, still the era of GPGPU computing was just begun ...

GPGPU Programming Approaches

- nVIDIA CUDA (Compute Unified Device Architecture)
  - a set of extensions to higher level programming language to use GPU as a coprocessor for heavy parallel task
  - a developer toolkit to compile, debug, profile programs and run them easily in a heterogeneous systems
- OpenCL (Open Computing Language):
  - a standard open-source programming model developed by major brands of hardware manufacters (Apple, Intel, AMD/ATI, nVIDIA).
    - like CUDA, provides extentions to C/C++ and a developer toolkit
    - extensions for specific hardware (GPUs, FPGAs, MICs, etc)
    - it’s very low level (verbose) programming
- Accelerator Directives Approach
  - OpenACC
  - OpenMP v4.x accelerator directives
  - you hope your compiler understand what you want, and do a good job
- Library Based:
  - MAGMA, CUDA Libraries, StarPu, ArrayFire, etc

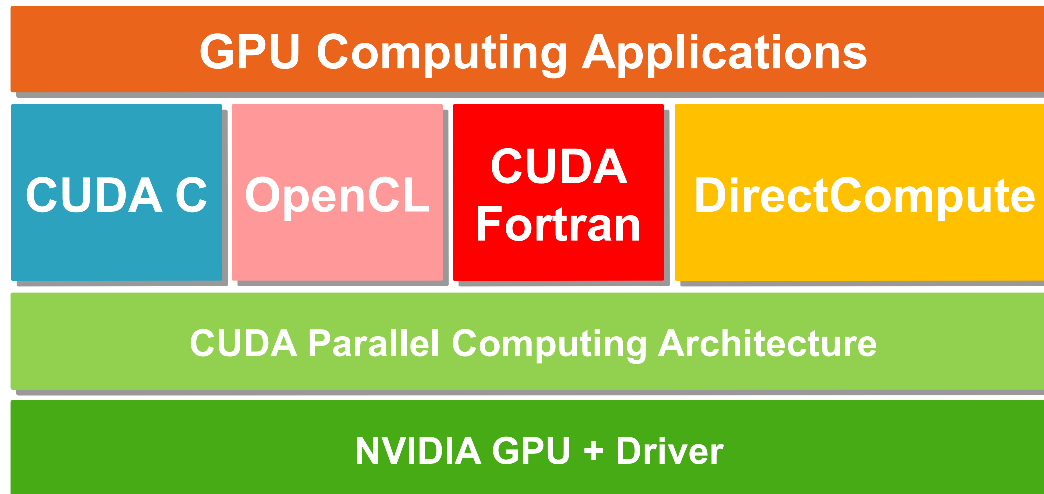
3 Ways to Accelerate Applications



General-Purpose Parallel Computing Architecture

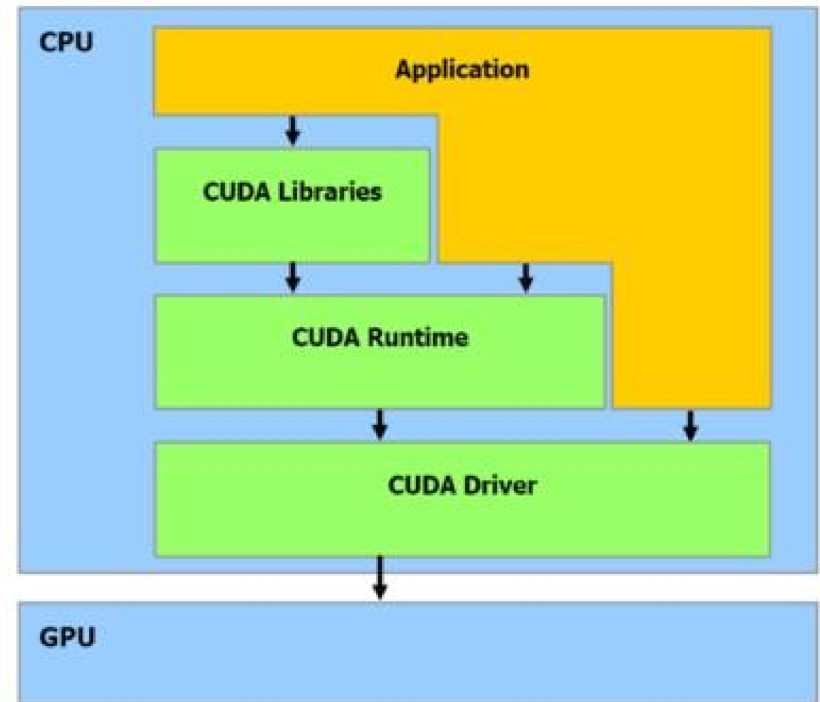
Compute Unified Device Architecture (CUDA)

- 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 Driver Vs Runtime API

- CUDA is composed of two APIs:
  - the CUDA runtime API
  - the CUDA driver API
- They are mutually exclusive
- Runtime API:
  - easier to program
  - it eases device code management: it's where the C-for-CUDA language lives
- Driver API:
  - requires more code: no syntax sugar for the kernel launch, for example
  - finer control over the device expecially in multithreaded application
  - doesn't need nvcc to compile the host code.



CUDA Driver API

- The driver API is implemented in the nvcuda dynamic library. All its entry points are prefixed with cu.
- It is a handle-based, imperative API: most objects are referenced by opaque handles that may be specified to functions to manipulate the objects.
- The driver API must be initialized with cuInit() before any function from the driver API is called. A CUDA context must then be created that is attached to a specific device and made current to the calling host thread.
- Within a CUDA context, kernels are explicitly loaded as PTX or binary objects by the host code**.
- Kernels are launched using API entry points.

- **by the way, any application that wants to run on future device architectures must load PTX, not binary code

NVIDIA Architectures naming

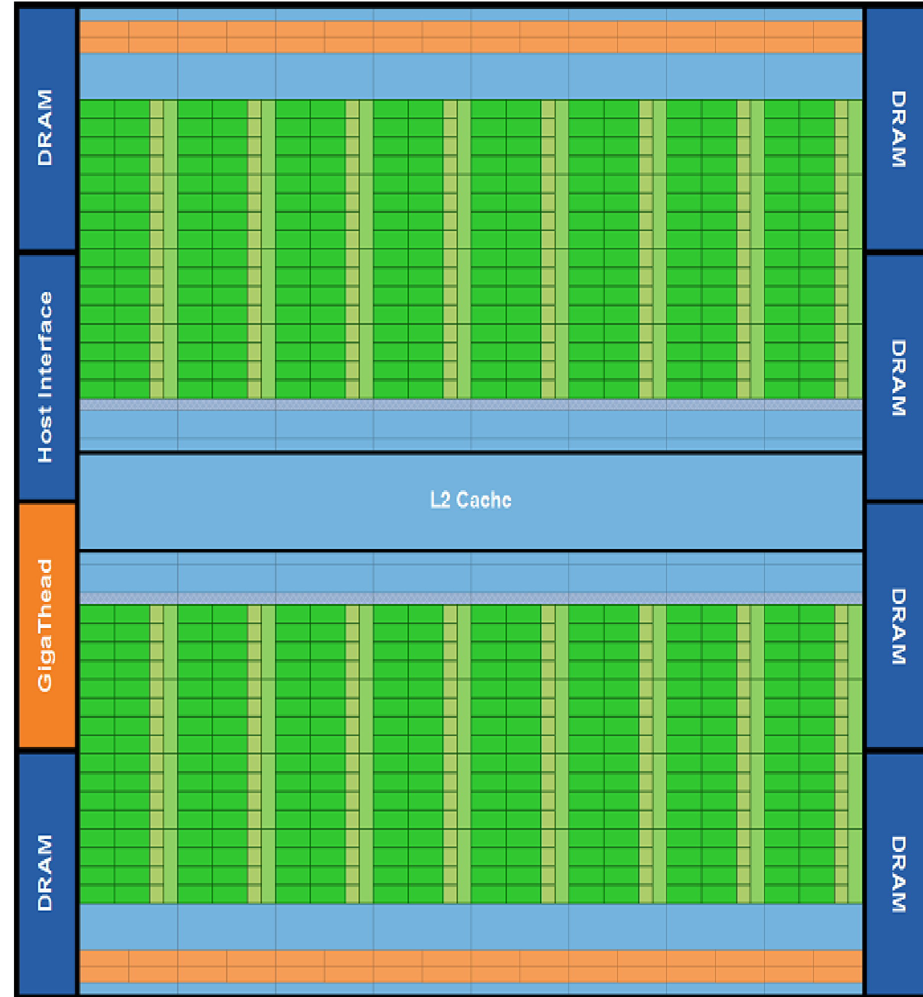
- Mainstream & laptops: GeForce
  - Target: videogames and multi-media

- Workstation: Quadro
  - Target: professional graphic applications such as CAD, modeling 3D, animation and visual effects

- GPGPU: Tesla
  - Target: High Performance Computing

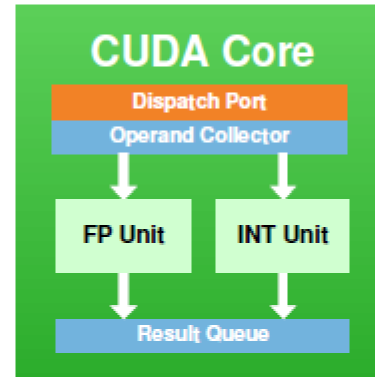
NVIDIA Fermi Architecture (2009)

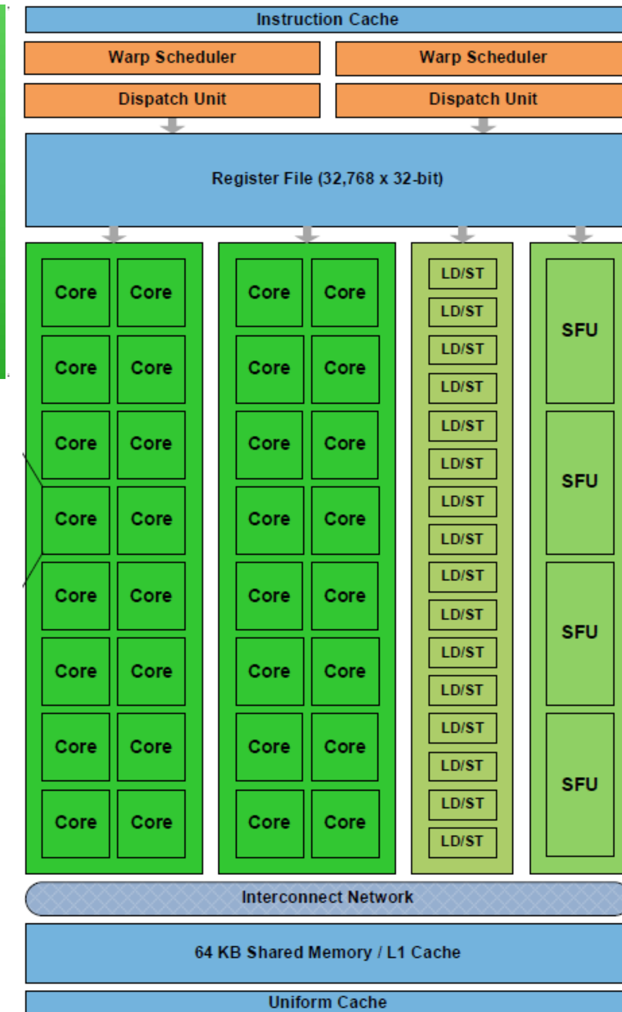
- 16 Streaming Multiprocessors (SM)
- DDR3 Memory
  - 4-6 GB global memory with ECC
- First model with a cache hierarchy:
  - L1 (16-48KB) per SM
  - L2 (768KB) shared among all SM
- 2 independent controllers for data transfer from/to host through PCI-Express
- Global thread scheduler (GigaThread global scheduler) which manage and distribute thread blocks to be processed on SM resources



Fermi Streaming Multiprocessor (SM)

- 32/48 CUDA cores with an arithmetic logic unit (ALU) and a floating point unit (FPU) fully pipelined
- floating point operations are fully IEEE 754-2008 a 32-bit e a 64-bit
- fused multiply-add (FMA) for both single and double precision
- 32768 registers (32-bit)
- 64KB configurable L1
- shared-memory/cache
- 48-16KB or 16-48KB shared/L1 cache
- 16 load/store units
- 4 Special Function Unit (SFU) to handle trascendental mathematical functions (sin, sqrt, recp-sqrt,..)

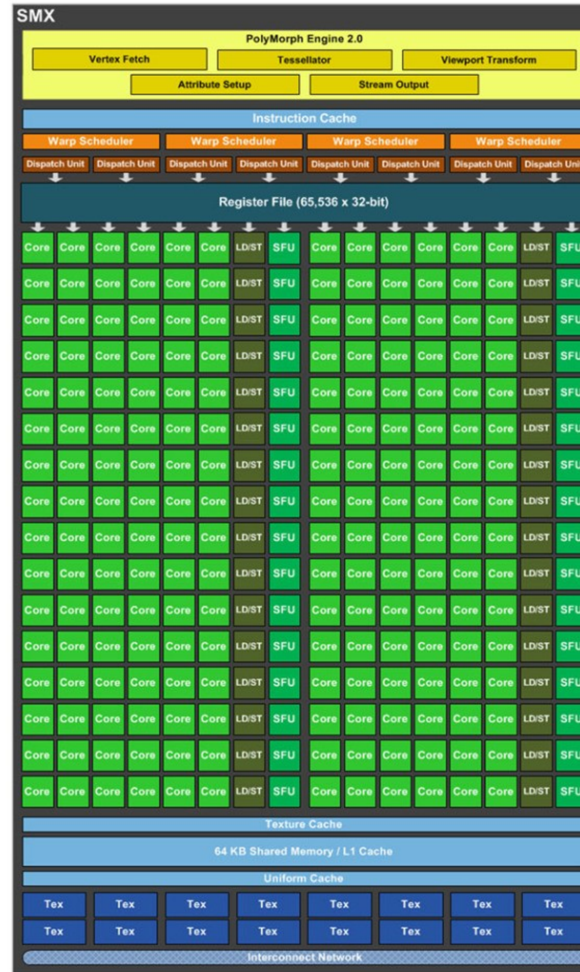


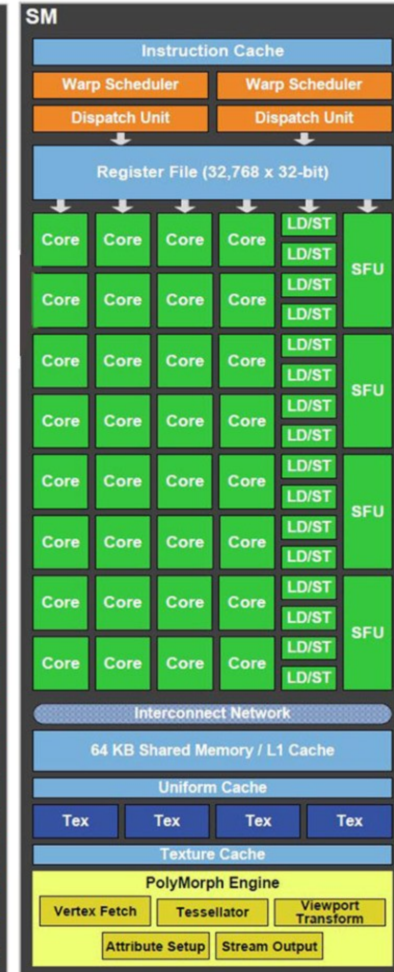


SCAI
SuperComputing Applications and Innovation

NVIDIA Kepler Architecture (2012)

- x3 performance/watt with respect to FERMI
- 28nm litography
- 192 CUDA cores
- 4 warp scheduler (2 dispatcher)
  - 2 independent instruction/warp
- 65536 registers per SM (32-bit)
- 32 load/store units
- 32 Special Function Unit
- 1534KB L2 cache (x2 vs Fermi)
- 64KB shared-memory/cache + 48KB read-only L1 cache
- 16 texture units (x4 vs Fermi)
  - Read-only cache
- Hyper-Q technology
  - Enable dynamic parallelism





NVIDIA Pascal Architecture (2016)

- 6 Compute Graphic Clusters (CGC) with 10 SM each
- 16nm litography
  - 2X Watt/Flop respect Kepler architecture
- 4MB L2 cache
- 3D stacked RAM HDDR5
- High Bandwidth Memory
  - 16GB RAM
  - 760 GB/s bandwidth
- NVLink tecnology
  - 80GB/s bandwidth to host data transfers
  - 5X respect PCIe Gen3 16x

Peak Performance: 5,7 TFlops

NVIDIA Pascal Architecture (2016)

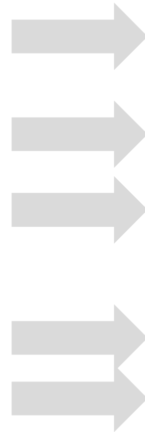
- SM composed of two independent blocks
- Each block sports:
  - 1 warps x 2 dispatchers
  - 32 ALU SIMD units
  - 16FP64 units
  - 8 Load/Store units
  - 8 SFU units
  - 32768 32bits registers
- Each block accesses:
  - 64KB shared memory
  - L1 64KB cache
  - 4 texture units

Compute Capability

- the *compute capability* of a device describes its architecture
  - *registers, memory sizes, features and capabilities*
- the compute capability is identified by a code like "`compute_Xy`"
  - major number (X): identifies base line chipset architecture
  - minor number (y): indentifies variants and releases of the base line chipset
- a compute capability select the set of usable PTX instructions

compute capability	***feature support***
compute_20	FERMI architecture
compute_30	KEPLER K10 architecture (only single precision)
compute_35	KEPLER K20, K20X, K40 architectures
compute_37	KEPLER K80 architecture (two K40 on a single board)
compute_53	MAXWELL GM200 architecture (only single precision)
compute_60	PASCAL GP100 architecture

Capability: resources constraints



Technical Specifications	Compute Capability						
	1.0	1.1	1.2	1.3	2.x	3.0	3.5
Maximum dimensionality of grid of thread blocks	2				3		
Maximum x-dimension of a grid of thread blocks	65535					2^{31}-1	
Maximum y- or z-dimension of a grid of thread blocks	65535						
Maximum dimensionality of thread block	3						
Maximum x- or y-dimension of a block	512				1024		
Maximum z-dimension of a block	64						
Maximum number of threads per block	512				1024		
Warp size	32						
Maximum number of resident blocks per multiprocessor	8					16	
Maximum number of resident warps per multiprocessor	24		32		48	64	
Maximum number of resident threads per multiprocessor	768		1024		1536	2048	
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K	
Maximum number of 32-bit registers per thread	128				63		255
Maximum amount of shared memory per multiprocessor	16 KB				48 KB		
Number of shared memory banks	16				32		
Amount of local memory per thread	16 KB				512 KB		
Constant memory size	64 KB						
Cache working set per multiprocessor for constant memory	8 KB						
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB						
Maximum width for a 1D texture reference bound to a CUDA array	8192				65536		

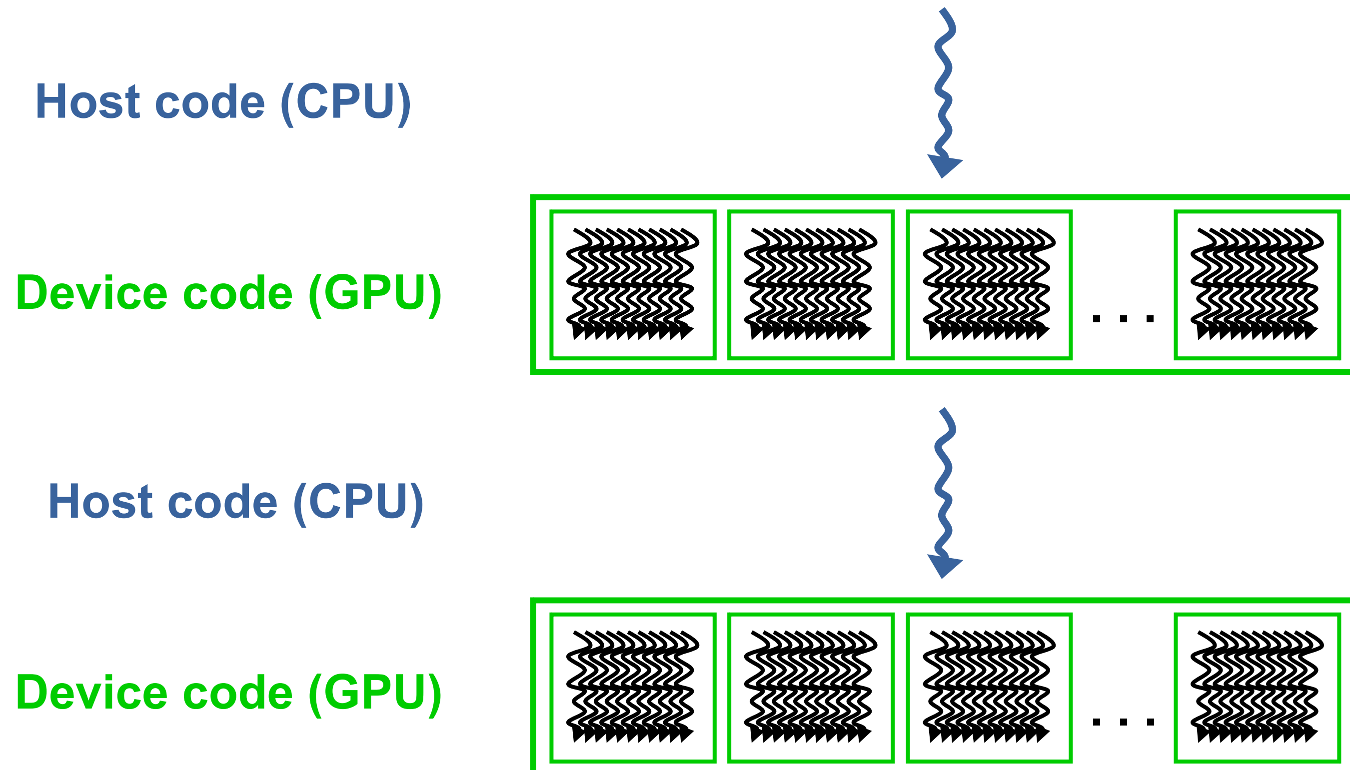
- CUDA programming model
  - Heterogeneous execution
  - Writing a CUDA Kernels
  - Thread Hierarchy
- Getting started with CUDA programming:
  - Vector-Vector Add
  - handling data transfers from
    CPU to GPU and back
  - write and launch a CUDA program

CUDA Programming Model

- GPU is seen as an auxilirary 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

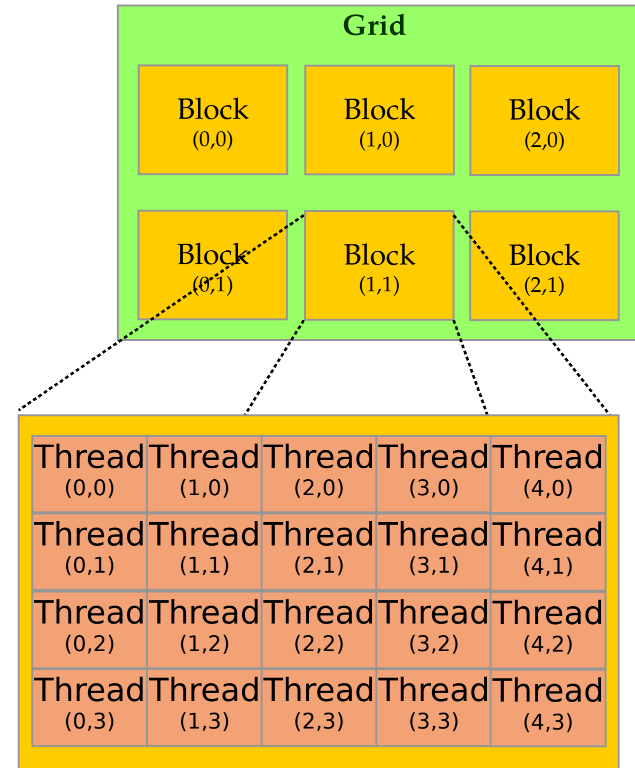
- Serial portions of a program, or those with low level of parallelism, keep running on the CPU (host)
- Data-parallel , computational intensive portions of the program are isolated into CUDA kernel function. The CUDA kernel are executed onto the GPU (device)
- Required data is moved on GPU memory and back to HOST memory



CINECA SCAI SuperComputing Applications and Innovation

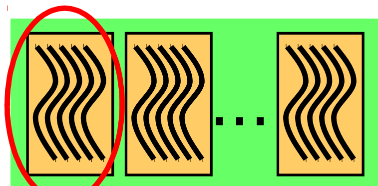
GPU Thread Hierarchy

- In order to compute N elements on the GPU in parallel, at least N concurrent threads must be created on the device
- GPU threads are grouped together in *teams* or *blocks* of threads
- Threads belonging to the same block or team can cooperate together exchanging data through the shared memory area

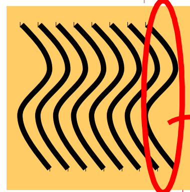


more on the CUDA Execution Model

Software | **Hardware**



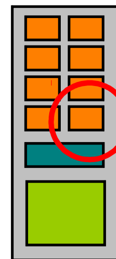
Griglia



Blocco di Thread

Thread

GPU



Streaming Multiprocessor

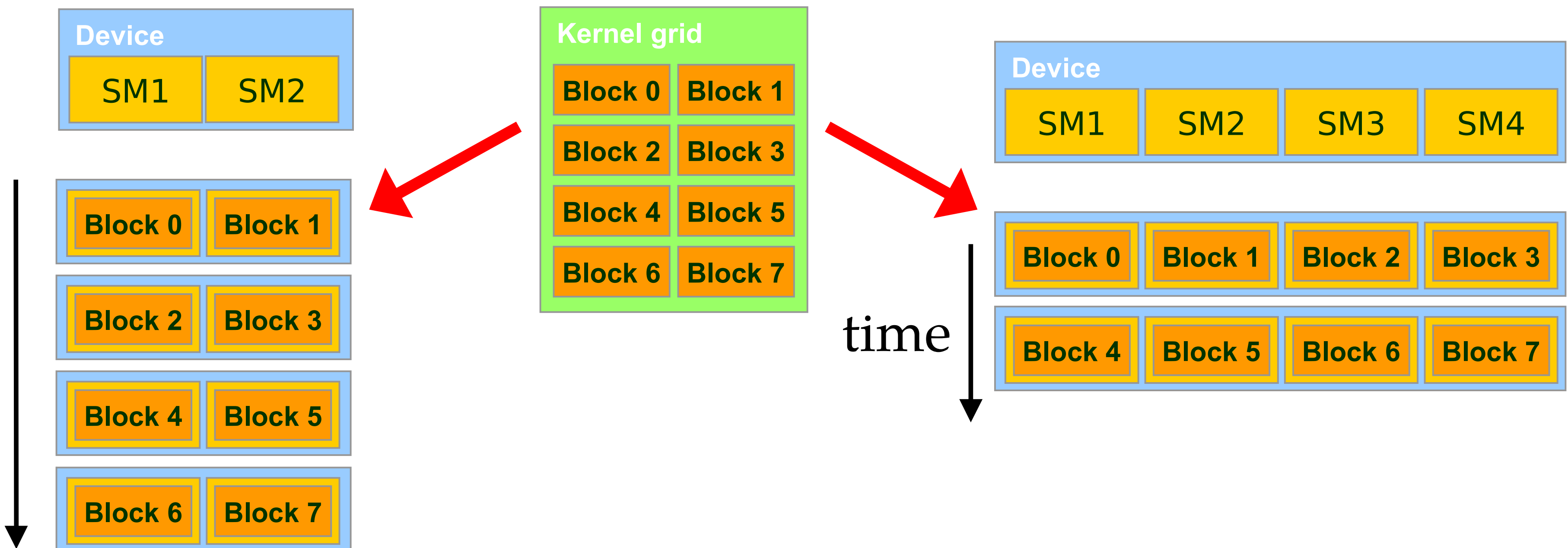
CUDA core

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 a 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

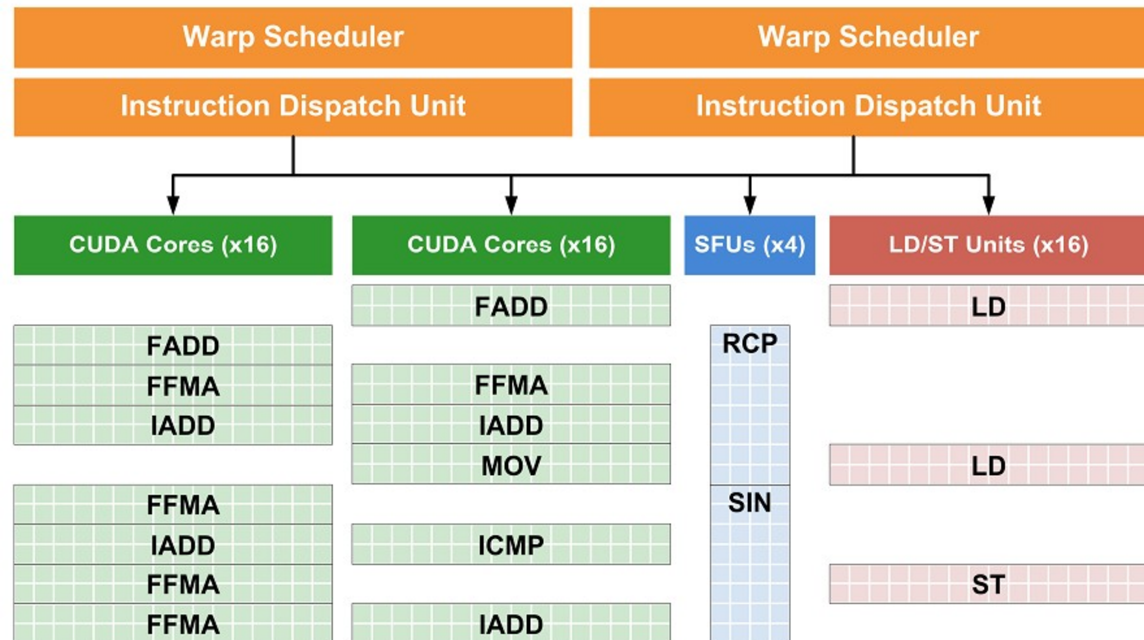
- CUDA runtime system can execute blocks in any order relative to each other.
- This flexibility enables to execute the same application code on hardware with different numbers of SM



CINECA SCAI SuperComputing Applications and Innovation

Warps

- The GPU multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*.
- Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently
- each *warp* can execute instructions on
  - SM cores
  - load/store units
  - SFUs units

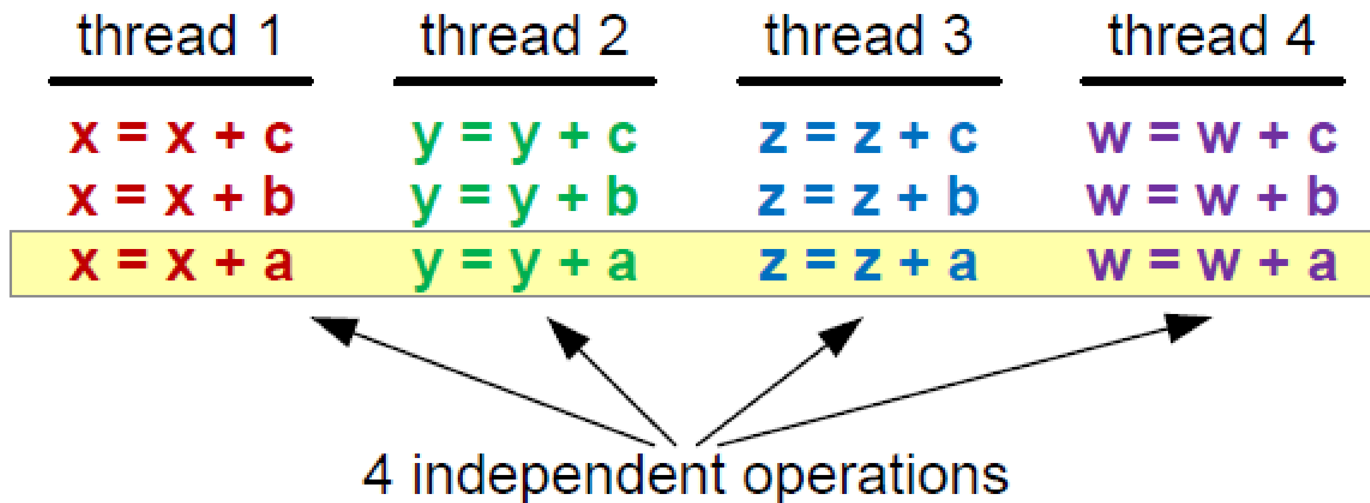


Hiding Latencies

- What is latency?
  - the number of clock cycles needed to complete an istruction
  - ... that is, the number of cycles I need to wait for before another **dependent operation** can start
    - arithmetic latency (~ 18-24 cycles)
    - memory access latency (~ 400-800 cycles)
- We cannot discard latencies (it's an hardware design effect), but we can lesser their effect and hide them.
  - saturating computational pipelines in computational bound problems
  - saturating bandwidth in memory bound problems
- We can organize our code so to provide the scheduler a sufficient number of **independent operations,** so that the more the warp are available, the more content-switch can hide latencies and proceed with other useful operations
- There are two possible ways and paradigms to use (can be combined too!)
  - Thread-Level Parallelism (TLP)
  - Instruction-Level Parallelism (ILP)

Thread-Level Parallelism (TLP)

- Strive for high SM occupancy: that is try to provide as much threads per SM as possible, so to easy the scheduler find a warp ready to execute, while the others are still busy
- This kind of approach is effective when there is a low level of independet operations per CUDA kernels



SuperComputing Applications and Innovation

Instruction-Level Parallelism (ILP)

- Strive for multiple independent operations inside you CUDA kernel: that is, let your kernel act on more than one data
- this will grant the scheduler to stay on the same warp and fully load each hardware pipeline

- note: the scheduler will not select a new warp untill there are eligible instructions ready to execute on the current warp

thread

w = w + b
z = z + b
y = y + b
x = x + b
w = w + a
z = z + a
y = y + a
x = x + a

4 independent operations

Three steps for a CUDA porting

1. identify data-parallel, computational intensive portions
   1. isolate them into functions (CUDA kernels candidates)
   2. identify involved data to be moved between CPU and GPU

2. translate identified CUDA kernel candidates into real CUDA kernels
   1. choose the appropriate thread index map to access data
   2. change code so that each thead acts on its own data

3. modify code in order to manage memory and kernel calls
   1. allocate memory on the device
   2. transfer needed data from host to device memory
   3. insert calls to CUDA kernel with execution configuration syntax
   4. transfer resulting data from device to host memory

Identify data-parallel intensive portions

```c
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;
}
```

```fortran
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
```

CINECA SCAI
SuperComputing Applications and Innovation

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 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

Manage kernel calls

Insert calls to CUDA kernels using the execution configuration syntax:

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

specifying the thread/block hierarchy you want to apply:

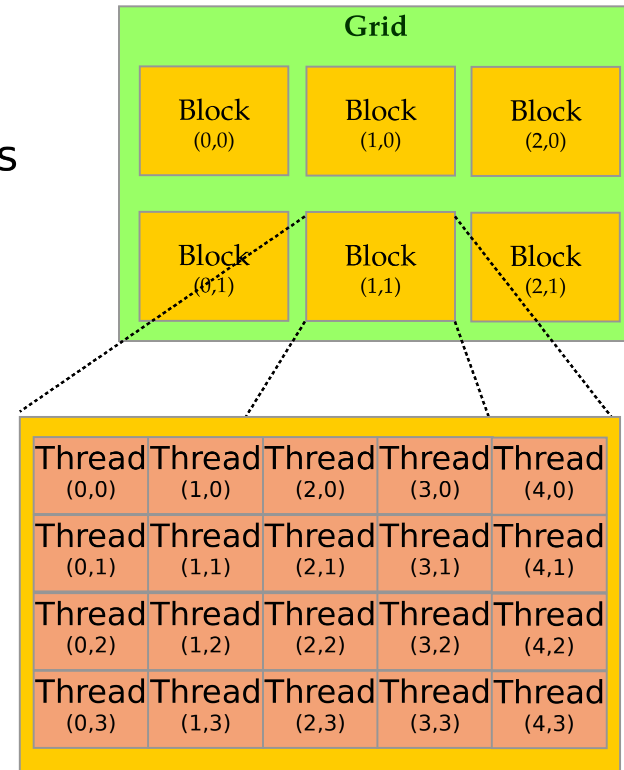
- **numBlocks:** specify grid size in terms of thread blocks along each dimension
- **numThreads:** specify the block size in terms of threads along each dimension

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

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

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



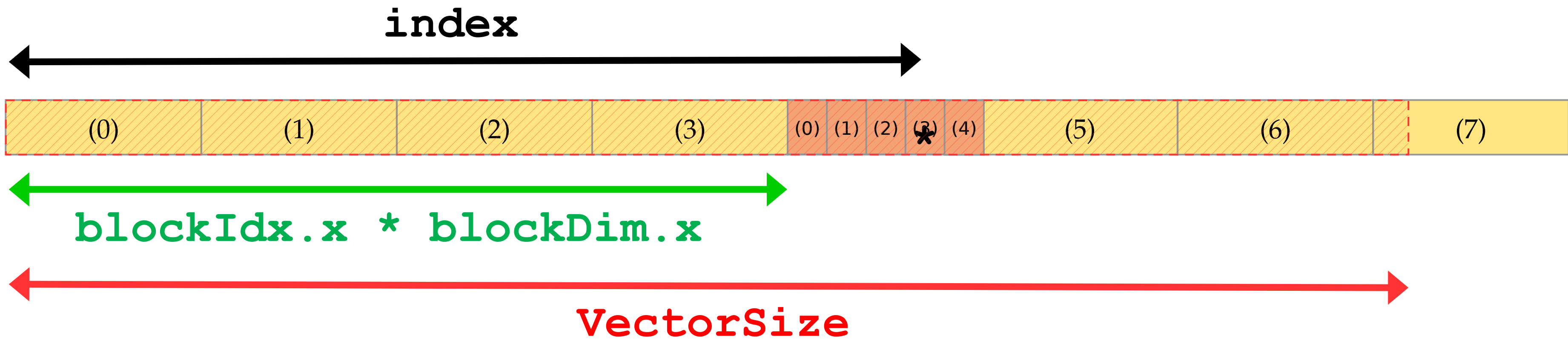
`threadIdx:`
thread coordinates inside a block

`blockIdx:`
block coordinates inside the grid

`blockDim:`
block dimensions in thread units

`gridDim:`
grid dimensions in block units

Composing 1D CUDA Thread Indexing




```
index = blockIdx.x * blockDim.x + threadIdx.x;
```

`threadIdx`: thread coordinates inside a block

`blockIdx`: block coordinates inside the grid

`blockDim`: block dimensions in thread units

`gridDim`: grid dimensions in block units

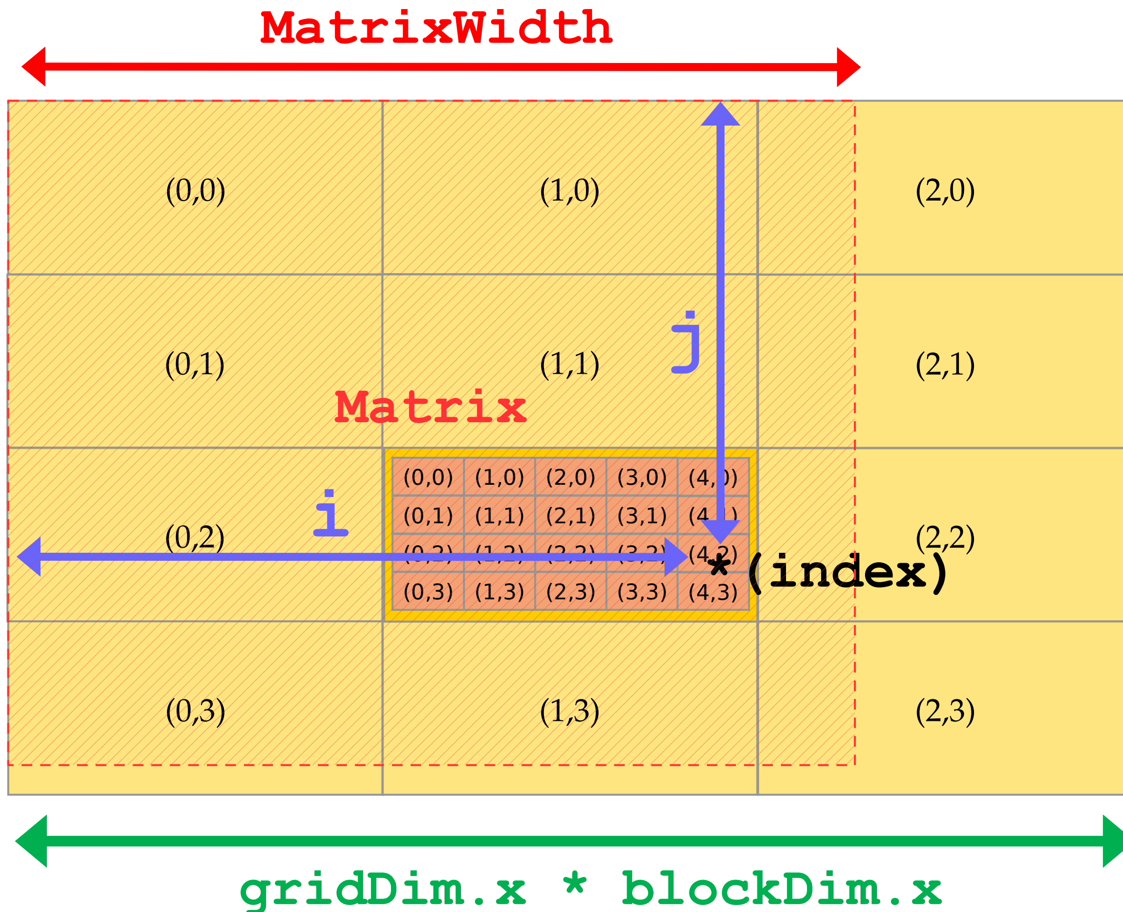
CINECA SCAI SuperComputing Applications and Innovation

CUDA Vector add – 1D thread grid

```
__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 - 1) / blockSize.x + 1;

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

CINECA SCAI SuperComputing Applications and Innovation

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

CINECA SCAI
SuperComputing Applications and Innovation

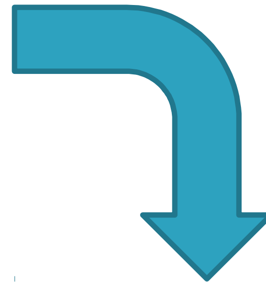
CUDA Matrix add - 2D thread grid

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

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

  if ( i < N && j < N )
    C[index] = A[index] + B[index];
}

int main(int argc, char *argv[]) {
  ...
  // use 2D block threads
  dim3 blockSize(32,32);
  // use 2D grid blocks
  dim3 gridSize( (N-1)/block.x + 1, (N-1)/block.y + 1 );
  // add NxN matrices on GPU
  matrixAdd <<< gridSize, blockSize >>> (N, A, B, C);
  ...
}
```

Translate parallel portions into kernels

- 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

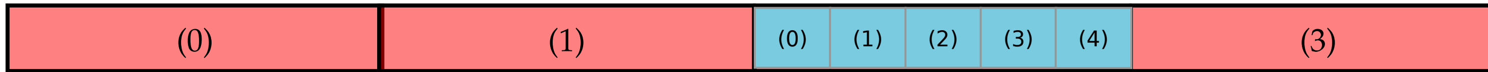
```
const int N = 1000;
double u[N], v[N], z[N];

// 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];
}
```

CINECA SCAI
SuperComputing Applications and Innovation

Translate parallel portions into kernels



^(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

Translate parallel portions into kernels

```
module vector_algebra_cuda
use cudafor
contains
attributes(global) subroutine gpuVectAdd (N, u, v, z)
  implicit none
  integer, intent(in), value :: N
  real, intent(in) :: u(N), v(N)
  real, intent(inout) :: z(N)
  integer :: i

   i = ( blockIdx%x - 1 ) * blockDim%x + threadIdx%x

   if (i .gt. N) return

    z(i) = u(i) + v(i)
end subroutine
end module vector_algebra_cuda
```

Translate parallel portions into kernels

```
attributes(global) subroutine gpuVectAdd (N, u, v, z)
  ...
end subroutine

program vectorAdd
use cudafor
implicit none
interface
  attributes(global) subroutine gpuVectAdd (N, u, v, z)
    integer, intent(in), value :: N
    real, intent(in) :: u(N), v(N)
    real, intent(inout) :: z(N)
    integer :: i
  end subroutine
end interface
  ...

end program vectorAdd
```

If the kernels are not defined within a module, then an explicit interface must be provided for each kernel you want to launch within a program unit.

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

CINECA SCAI SuperComputing Applications and Innovation

Memory allocation on GPU device

```
double *u_dev;

cudaMalloc((void **) &u_dev, N*sizeof(double));
```

- &u_dev
  - u_dev it's a variable defined on the *host* memory
  - u_dev contains an address of the *device* memory
  - C pass arguments to function by value
    - we need to pass u_dev by reference to let its value be modified by the cudaMalloc function
    - this has nothing to do with CUDA, it's a C common idiom
    - if you don't understand this, probably you are not ready for this course
- (void **) is a cast to force cudaMalloc to handle pointer to memory of any kind
    - again, if you don't understand this…

Memory allocation on GPU device

- CUDA C API: `cudaMalloc(void **p, size_t size)`
  - allocates size bytes of GPU global memory
  - p is a valid device memory address (i.e. SEGV if you dereference p on the host)

```
double *u_dev, *v_dev, *z_dev;

cudaMalloc((void **)&u_dev, N * sizeof(double));
cudaMalloc((void **)&v_dev, N * sizeof(double));
cudaMalloc((void **)&z_dev, N * sizeof(double));
```

- in CUDA Fortran the attribute **device** needs to be used while declaring a GPU array. The array can be allocated by using the Fortran statement **allocate**:

```
real(kind(0.0d0)), device, allocatable, dimension(:,:) :: u_dev, v_dev, z_dev

allocate( u_dev(N), v_dev(N), z_dev(N) )
```

CINECA SCAI SuperComputing Applications and Innovation

Memory Initialization on GPU device

- `cudaMemset(void* devPtr, int value, size_t count)`

It fills the first count <u>bytes</u> of the memory area pointed to by devPtr with the constant byte of the int value converted to unsigned char.

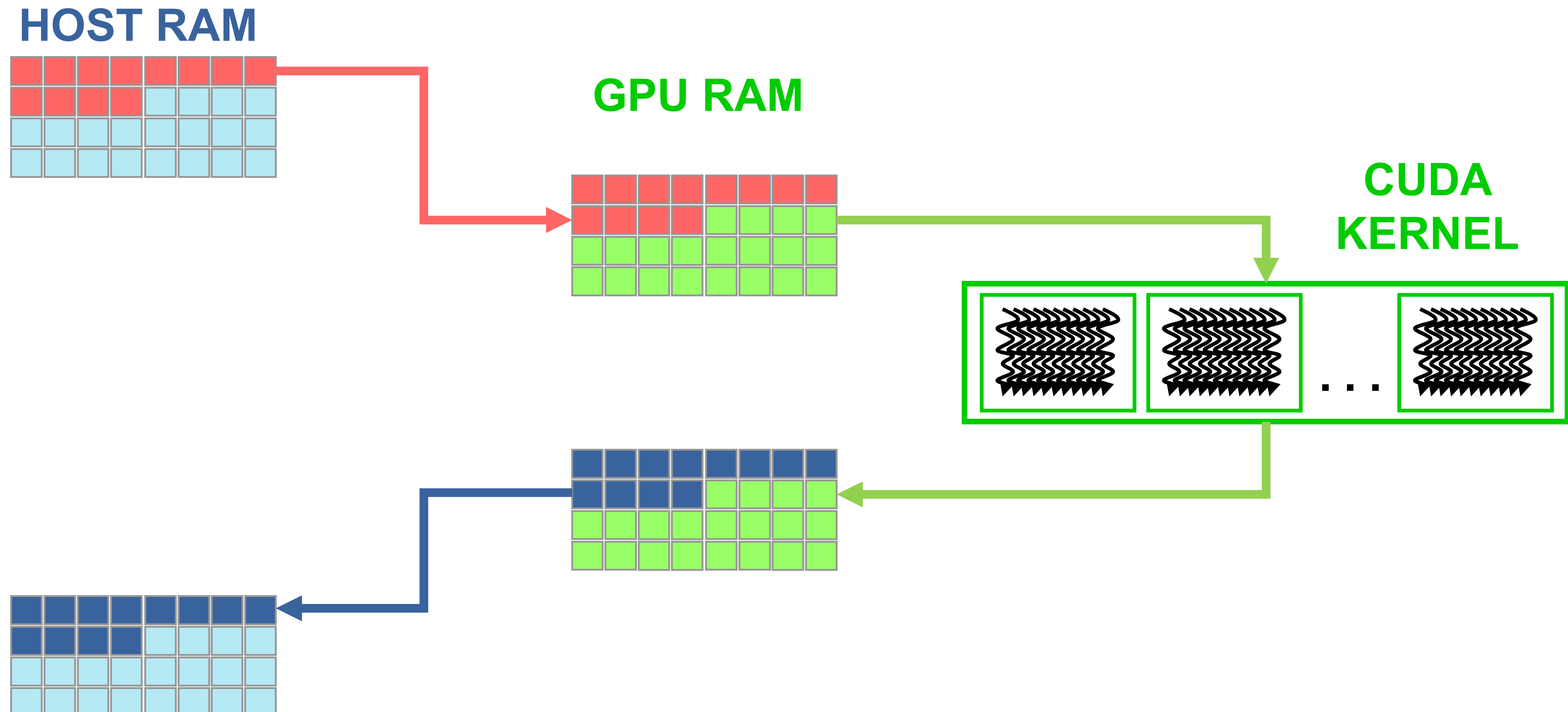
  - it's like the standard library C memset() function
  - devPtr - Pointer to device memory
  - value - Value to set for each byte of specified memory
  - count - Size in bytes to set

- REM: to initialize an array of double (float, int, ...) to a specific value you need to execute a CUDA kernel.

CINECA SCAI SuperComputing Applications and Innovation

Data movement

- Data must be moved from HOST to DEVICE memory in order to be processed by a CUDA kernel
- When data is processed, and no more needed on the GPU, it is transferred back to HOST



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

Manage memory transfers

- CUDA C API:
  `cudaMemcpy(void *dst, void *src, size_t size, direction)`
  - copy size bytes from the src to dst buffer

```
cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);
```

- in CUDA Fortran you can use the array syntax

```
u_dev = u ; v_dev = v
```

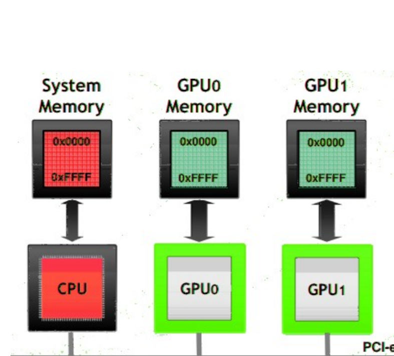
CINECA SCAI SuperComputing Applications and Innovation

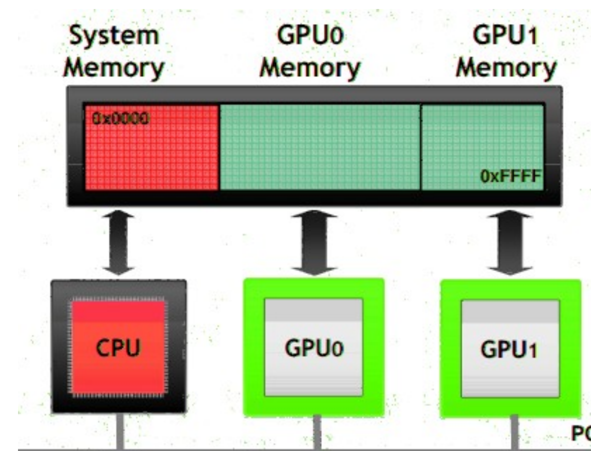
CUDA 4.x - Unified Virtual Addressing

- CUDA 4.0 introduces a unique virtual address space for memory (Unified Virtual Address) shared between GPU and HOST:
  - the actual memory type a data resides is automatically understood at runtime
  - greatly simplify programming model
  - allow simple addressing and transfer of data among GPU devices

Pre-UVA	UVA
A macro for each combination of source/destination	The system keeps track of the buffer location.

`cudaMemcpyHostToHost`
`cudaMemcpyHostToDevice`
`cudaMemcpyDeviceToHost`
`cudaMemcpyDeviceToDevice`
	`cudaMemcpyDefault`





CUDA 6.x - Unified Memory

- Unified Memory creates a pool of memory with an address space that is shared between the CPU and GPU. In other word, a block of Unified Memory is accessible to both the CPU and GPU by using the same pointer;
- the system automatically *migrates* data allocated in Unified Memory mode between the host and device memory
  - no need to explicitly declare device memory regions
  - no need to explicitly copy back and forth data between CPU and GPU devices
  - greatly simplifies programming and speeds up CUDA ports
- REM: it can result in performances degradation with respect to an explicit, finely tuned data transfer.

Sample code using CUDA Unified Memory

CPU code

```
void sortfile (FILE *fp, int N) {
  char *data;

  data = (char *) malloc (N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);

  use_data(data);

  free(data)
}
```

GPU code

```
void sortfile(FILE *fp, int N) {
  char *data;

  cudaMallocManaged(&data, N);

  fread(data, 1, N, compare);

  qsort<<< ... >>> (data, N, 1, compare);

  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

Vector Sum: the complete CUDA code

```
double *u_dev, *v_dev, *z_dev;
cudaMalloc((void **)&u_dev, N * sizeof(double));
cudaMalloc((void **)&v_dev, N * sizeof(double));
cudaMalloc((void **)&z_dev, N * sizeof(double));

cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);

dim3 numThreads( 256 ); // 128-512 are good choices
dim3 numBlocks( (N + numThreads.x - 1) / numThreads.x );
gpuVectAdd<<<numBlocks, numThreads>>>( N, u_dev, v_dev, z_dev );
cudaMemcpy(z, z_dev, N * sizeof(double), cudaMemcpyDeviceToHost);
```

```
real(kind(0.0d0)), device, allocatable, dimension(:,:) :: u_dev, v_dev, z_dev
type(dim3) :: numBlocks, numThreads
allocate( u_dev(N), v_dev(N), z_dev(N) )
u_dev = u; v_dev = v
numThreads = dim3( 256, 1, 1 ) ! 128-512 are good choices
numBlocks = dim3( (N + numThreads%x - 1) / numThreads%x, 1, 1 )
call gpuVectAdd<<<numBlocks,numThreads>>>( N, u_dev, v_dev, z_dev )
z = z_dev
```