



Introduction to CUDA II

Ivan Girotto – igirotto@ictp.it

Information & Communication Technology Section (ICTS)
International Centre for Theoretical Physics (ICTP)



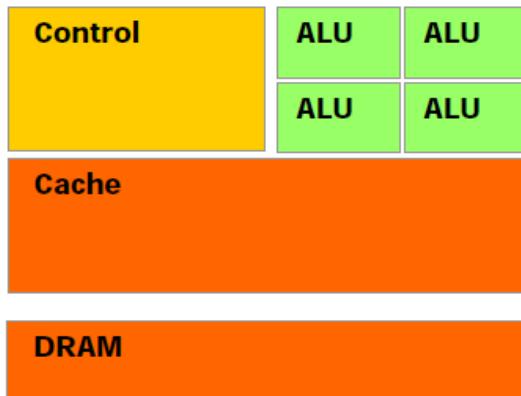
Scuola Internazionale Superiore
di Studi Avanzati



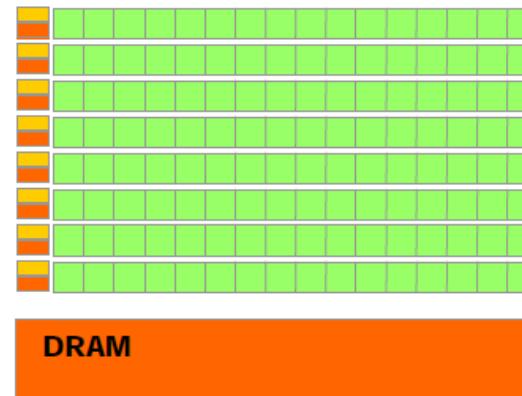
GPU Work Abstraction

- CUDA Kernels can be thought of as telling a GPU to compute all iterations of a set of nested loops concurrently
- Threads are dynamically scheduled onto hardware according to a hierarchy of thread groupings

CPU: Cache heavy, focused on individual thread performance

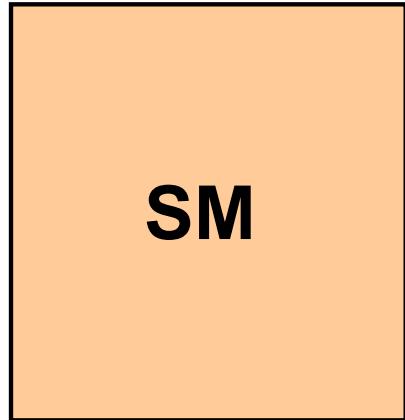


GPU: ALU heavy, massively parallel, throughput oriented

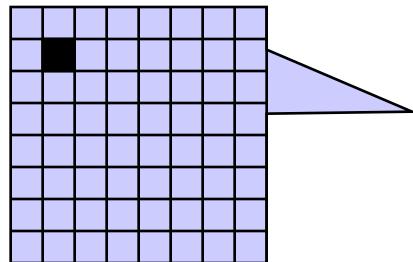


Grids, Thread Blocks, Threads

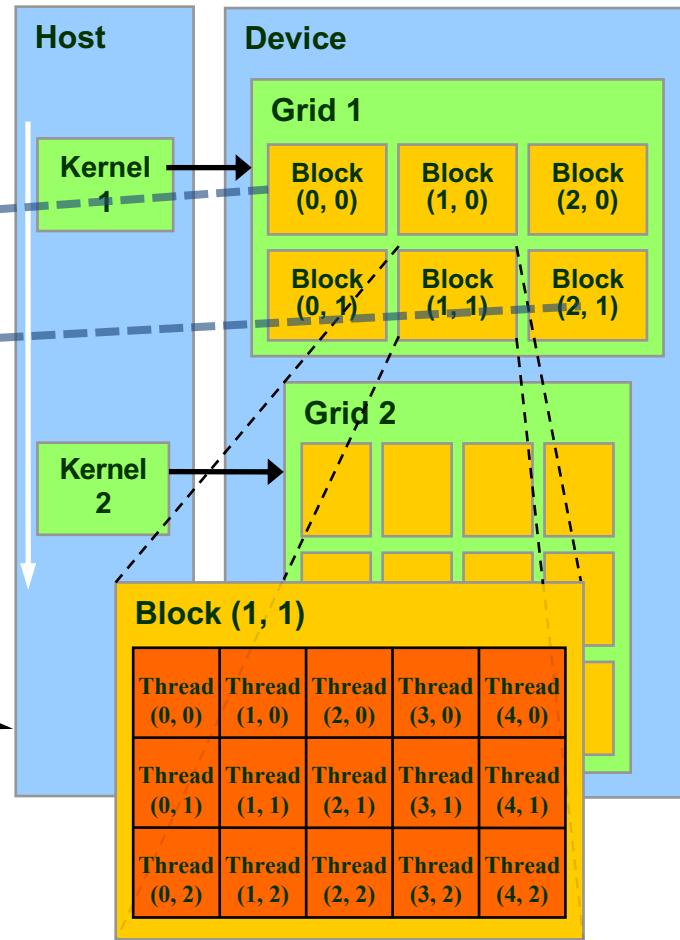
Thread blocks are scheduled onto pool of GPU SMs...



1-D, 2-D, 3-D thread block:



1-D, 2-D, or 3-D Grid of Ths blocks:



Built-in Variables to manage grids and blocks

dim3 => a new datatype defined by CUDA:

- **struct dim3 { unsigned int x, y, z };**
 - three unsigned ints where any unspecified component defaults to 1.
-
- **dim3 gridDim;**
 - Dimensions of the grid in blocks
 - **dim3 blockDim;**
 - Dimensions of the block in threads
 - **dim3 blockIdx;**
 - Block index within the grid
 - **dim3 threadIdx;**
 - Thread index within the block

Bi-dimensional threads configuration: set the elements of a square matrix

```
__global__ void kernel( int *a, int dimx, int dimy ) {
    int ix = blockIdx.x*blockDim.x + threadIdx.x;
    int iy = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx] = idx+1;
}
```

```
int main() {
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x = dimx / block.x;
    grid.y = dimy / block.y;

    kernel<<<grid, block>>>( d_a, dimx, dimy );

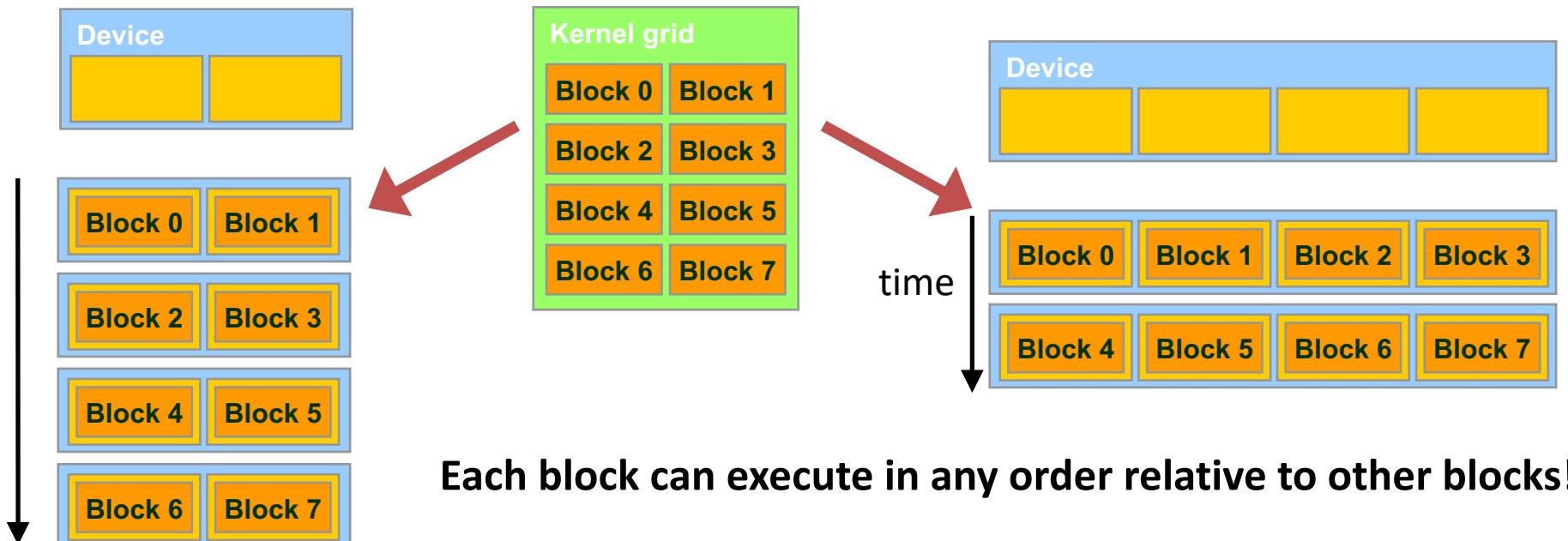
    cudaMemcpy(h_a,d_a,num_bytes,
              cudaMemcpyDeviceToHost);

    for(int row=0; row<dimy; row++) {
        for(int col=0; col<dimx; col++)
            printf("%d ", h_a[row*dimx+col] );
        printf("\n");
    }

    free( h_a );
    cudaFree( d_a );
    return 0;
}
```

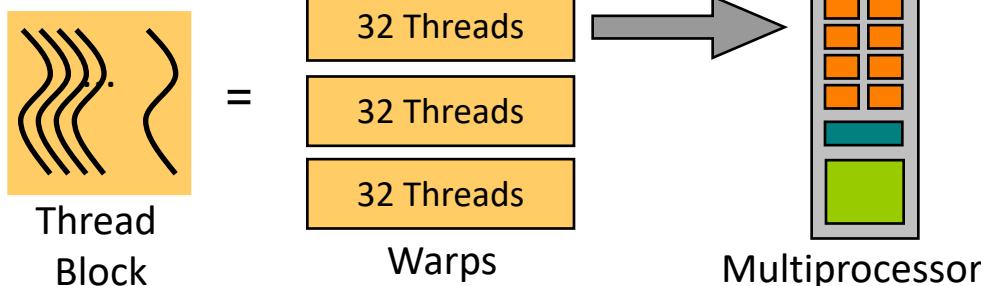
Transparent Scalability

- Hardware is free to assigns blocks to any processor at any time
 - A kernel scales across any number of parallel processors

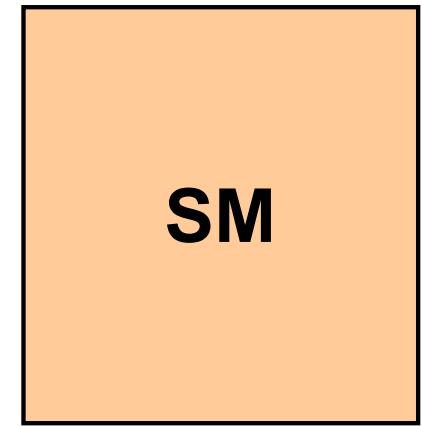


GPU Thread Block Execution

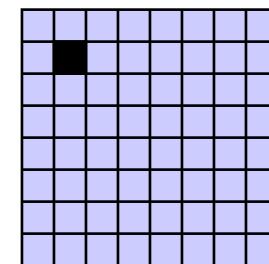
- Thread blocks are decomposed onto hardware in **32-thread “warps”**
- Hardware execution is scheduled in units of **warps**
 - an SM can execute warps from several thread blocks
- **Warps** run in SIMD-style execution:
 - All threads execute the same instruction in lock-step
 - If one thread stalls, the entire warp stalls...
 - A branch taken by a thread has to be taken by all threads...
(divergence is bad)



Thread blocks are multiplexed onto pool of GPU SMs...



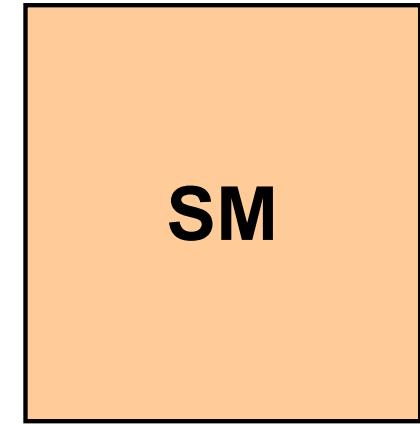
1-D, 2-D, 3-D
thread block:



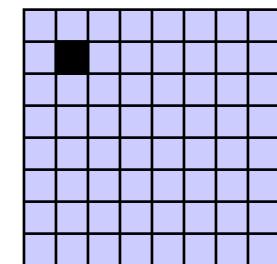
GPU Warp Branch Divergence

- Branch divergence: when not all threads take the same branch, the entire warp has to **execute both sides of the branch**
- Branch divergence issue not unique to GPUs, affects **all SIMD hardware platforms...**
- On GPUs, we get fast **hardware-based** implementation of predication/masking/etc...
- GPU blocks memory writes from disabled threads in the “if then” branch, then inverts all thread enable states and runs the “else” branch
- GPU hardware detects warp re-convergence and then runs with all threads enabled...

Thread blocks are multiplexed onto pool of GPU SMs...



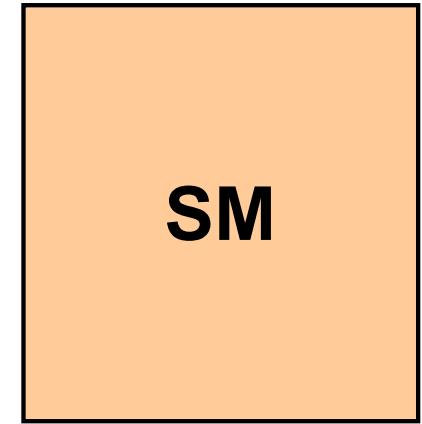
1-D, 2-D, 3-D
thread block:

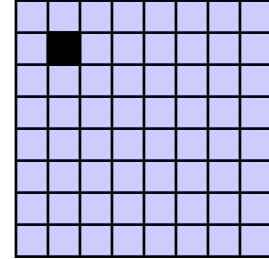


GPU Warp Branch Divergence

- Threads within the same thread block can communicate with each other in fast on-chip shared memory
- Once scheduled on an SM, thread blocks run until completion
- Because the order of thread block execution is arbitrary and blocks cannot be stopped, they cannot communicate or synchronize with other thread blocks (*)
- (*) Atomic memory ops are an exception wrt/ communication

Thread blocks are multiplexed onto pool of GPU SMs...



1-D, 2-D, 3-D
thread block:




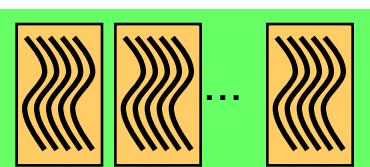
Execution Model

Software

Thread



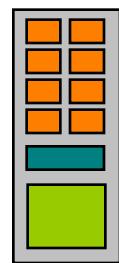
Thread
Block



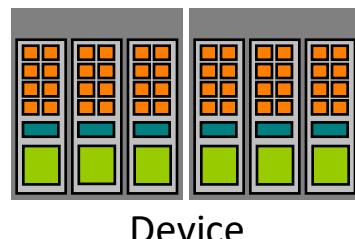
Grid

Hardware

Scalar
Processor



Multiprocessor



Device

Threads are executed by scalar processors

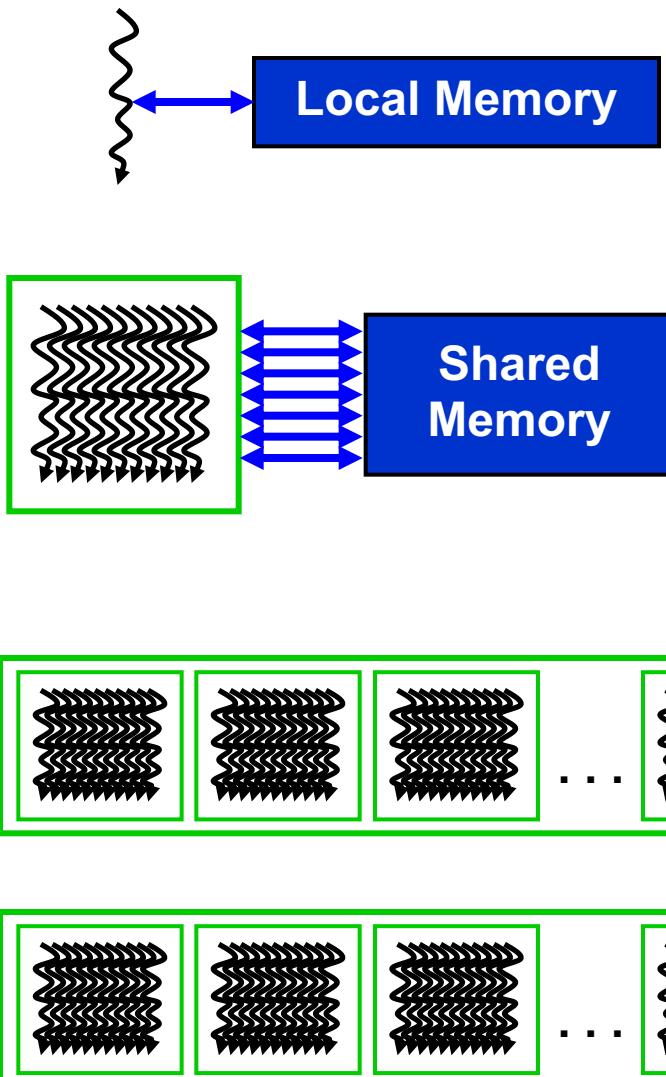
Thread blocks are executed on multiprocessors

Thread blocks **do not migrate**

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

Memory Hierarchy



- Registers (fast up to availability)
 - Limited num. registers available per block
- Shared Memory: per-block
 - Shared by threads of the same block
 - *Fast* inter-thread communication
- Global Memory: per-application
 - Shared by all threads
 - Inter-Grid communication

**Sequential Grids
Execution in
Time**

Synchronization

- GPU blocks do not synchronize each other (except using atomic operations)
- Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses
- one can specify synchronization points in the kernel by calling the `__syncthreads()` intrinsic function
- `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed
- `cudaDeviceSynchronize()` can be used on host to synchronize the host with a GPU.
- Kernels are asynchronous to the host, `cudaMemcpy` is not (blocking)

Atomic Operations

- Terminology: Read-modify-write uninterruptible when *atomic*
- Many *atomic operations* on memory available with CUDA C
 - `atomicAdd()`
 - `atomicSub()`
 - `atomicMin()`
 - `atomicMax()`
 - `atomicInc()`
 - `atomicDec()`
 - Other `atomic` ...
- Predictable result when simultaneous access to memory required

Multiblock Dot Product: dot ()

```
__global__ void dot( int *a, int *b, int *c ) {  
  
    __shared__ int temp[THREADS_PER_BLOCK];  
  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
  
    temp[threadIdx.x] = a[index] * b[index];  
  
    __syncthreads();  
  
    if( 0 == threadIdx.x ) {  
        int sum = 0;  
        for( int i = 0; i < THREADS_PER_BLOCK; i++ ) sum += temp[i];  
        atomicAdd( c, sum );  
    }  
}
```

- We need to atomically add **sum** to **c** in our multiblock dot product



GPU On-Board Global Memory

GPU arithmetic rates dwarf memory bandwidth

For Kepler K40 hardware:

~4.3 SP TFLOPS vs. ~288 GB/sec

The ratio is roughly **60 FLOPS per memory reference**
for single-precision floating point

Peak performance achieved with “**coalesced**” memory access patterns –
patterns that result in a single hardware memory transaction for a SIMD
“warp” – a contiguous group of 32 threads



Scuola Internazionale Superiore
di Studi Avanzati

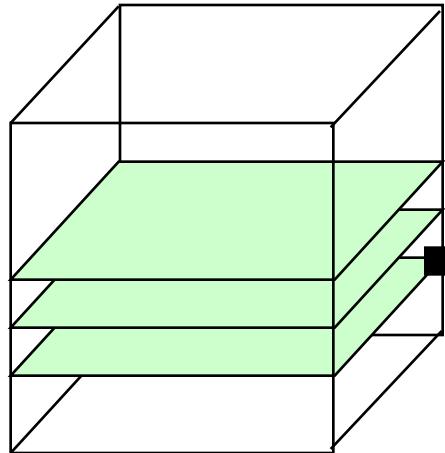


Memory Coalescing (Oversimplified explanation)

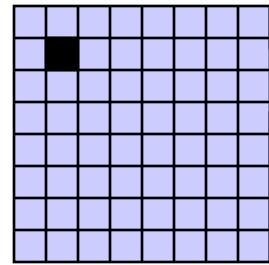
- Threads in a warp perform a read/write operation that can be serviced in a single hardware transaction
- Rules vary slightly between hardware generations, but new GPUs are much more flexible than old ones
- If all threads in a warp read from a contiguous region that's 32 items of 4, 8, or 16 bytes in size, that's an example of a coalesced access
- Multiple threads reading the same data are handled by a hardware broadcast
- Writes are similar, but multiple writes to the same location yields undefined results

CUDA Grid/Block/Thread Decomposition

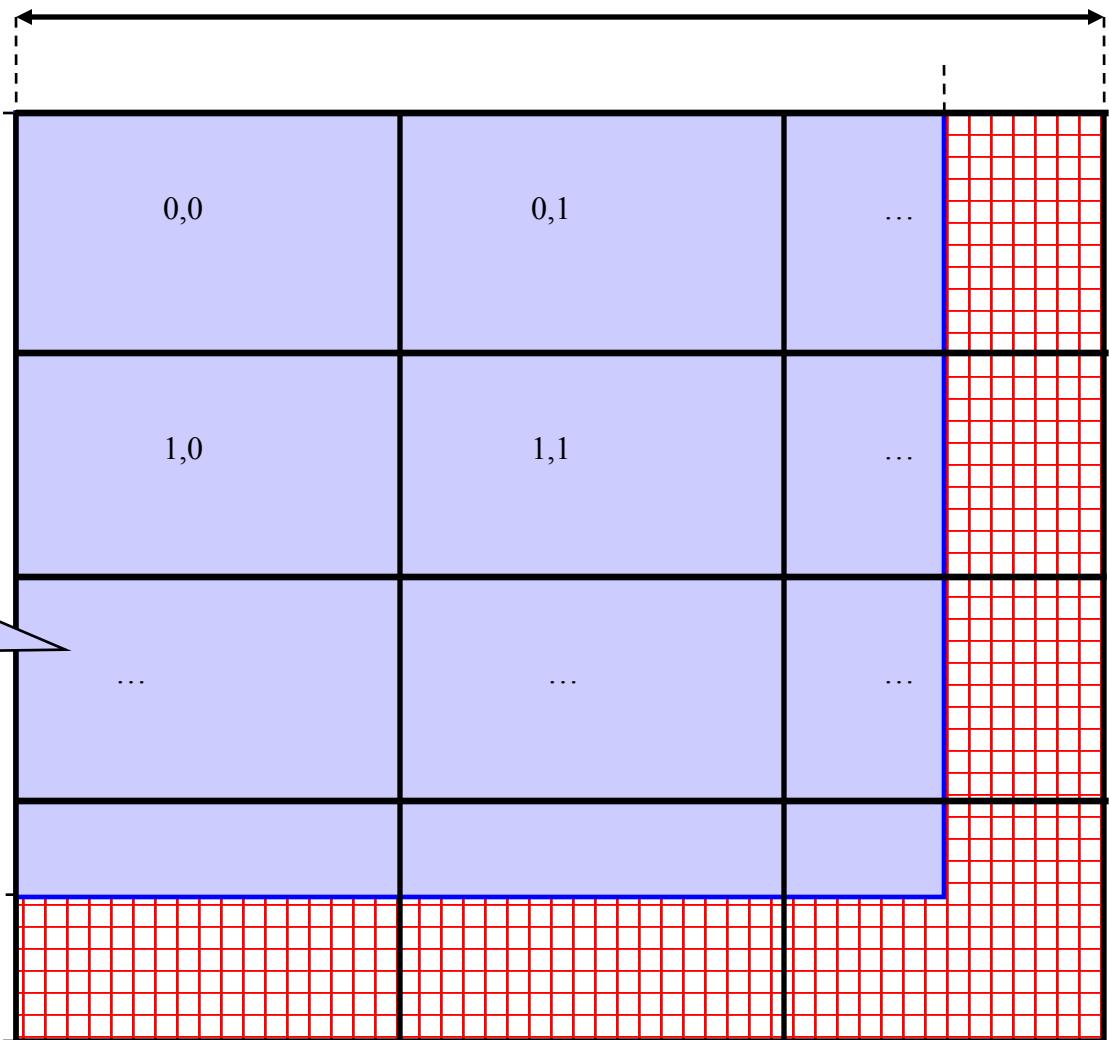
**1-D, 2-D, or 3-D
Computational Domain**



**1-D, 2-D, 3-D
thread block:**



**1-D, 2-D, or 3-D (SM >= 2.x)
Grid of thread blocks:**



Padding arrays out to full blocks
optimizes global memory performance
by guaranteeing memory coalescing

CUDA Compiler: nvcc basic options

- **-arch=sm_35 →** enable code for a given capability
- **-G →** enable debug for device code
- **--ptxas-options=-v →** show register and memory usage
- **--maxrregcount <N> →** limit the number of registers
- **-use_fast_math →** use fast math library
- **-O3 →** Enables compiler optimization
- **-ccbin *compiler_path* →** use a different C compiler
- **--compiler-options →** Specify options directly to the compiler/preprocessor.



Scuola Internazionale Superiore
di Studi Avanzati



Getting Performance From GPUs

- Don't worry (much) about counting arithmetic operations...at least until you have nothing else left to do
- GPUs provide tremendous memory bandwidth, but even so, **memory bandwidth often ends up being the performance limiter**
- Keep/reuse data in **registers** as long as possible
- The main consideration when programming GPUs is **accessing memory efficiently**, and storing operands in the **most appropriate memory system** according to data size and access pattern



Scuola Internazionale Superiore
di Studi Avanzati





Getting Performance From GPUs

- Don't worry (much) about counting arithmetic operations...at least until you have nothing else left to do
- GPUs provide tremendous memory bandwidth, but even so, **memory bandwidth often ends up being the performance limiter**
- Keep/reuse data in **registers** as long as possible
- The main consideration when programming GPUs is **accessing memory efficiently**, and storing operands in the **most appropriate memory system** according to data size and access pattern



Scuola Internazionale Superiore
di Studi Avanzati





Why Does GPU Accelerate Computing?

Highly scalable design

Higher aggregate memory bandwidth

Huge number of low frequency cores

Higher aggregate computational power

Massively parallel processors for data processing



Scuola Internazionale Superiore
di Studi Avanzati





Why Does GPU Accelerate Computing?

Highly scalable design

Higher aggregate memory bandwidth

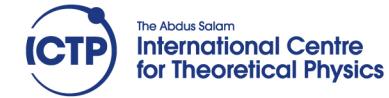
Huge number of low frequency cores

Higher aggregate computational power

Massively parallel processors for data processing



Scuola Internazionale Superiore
di Studi Avanzati





Why Does GPU Not Accelerate Computing?

PCI Bus bottleneck

Synchronization weakness

Extremely slow serialized execution

High complexity, SPMD + SIMD + Memory Model

People forget about the Amdahl's law: accelerating only the 50% of the original code, the expected speedup can get at most a value of 2!!

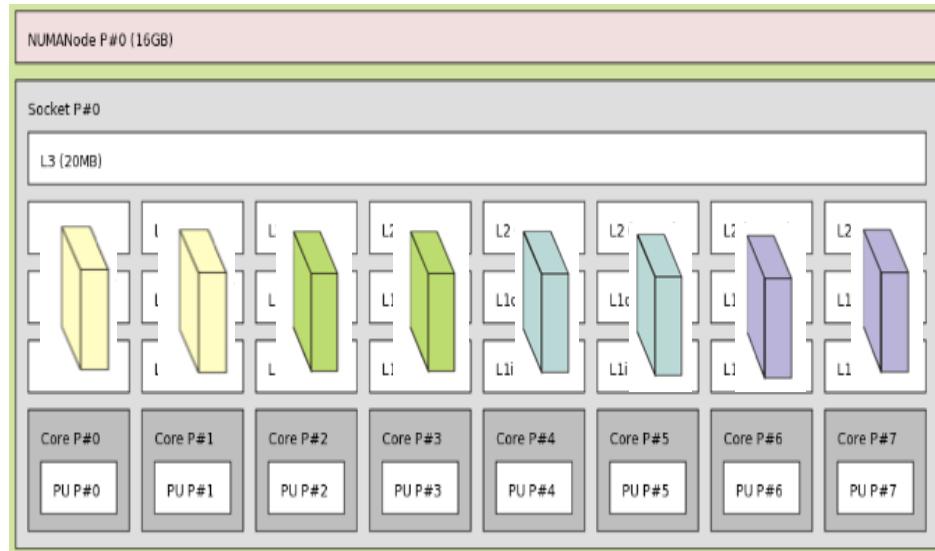


Scuola Internazionale Superiore
di Studi Avanzati



The Abdus Salam
International Centre
for Theoretical Physics

CPU & GPU



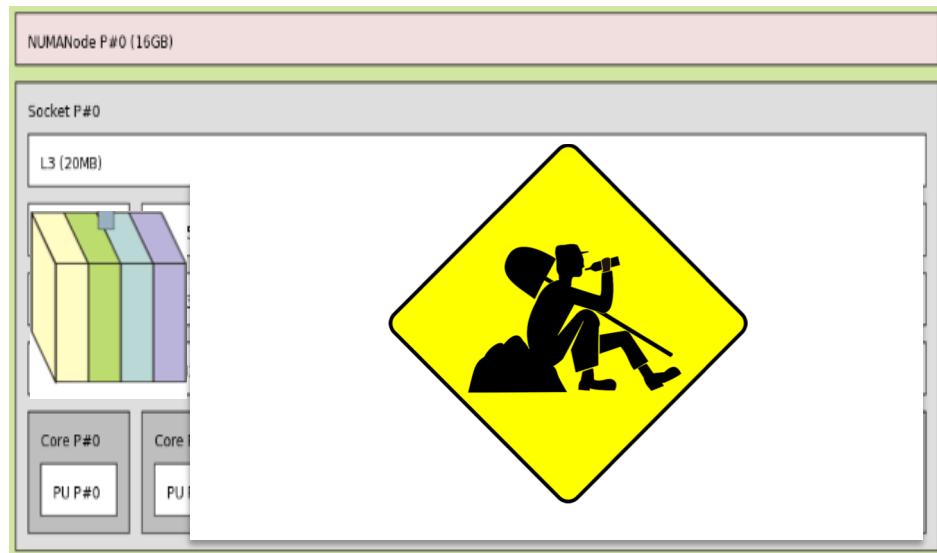
The Intel Xeon E5-2665
Sandy Bridge-EP 2.4GHz

~ 8 GBytes



Scuola Internazionale Superiore
di Studi Avanzati

CPU & GPU

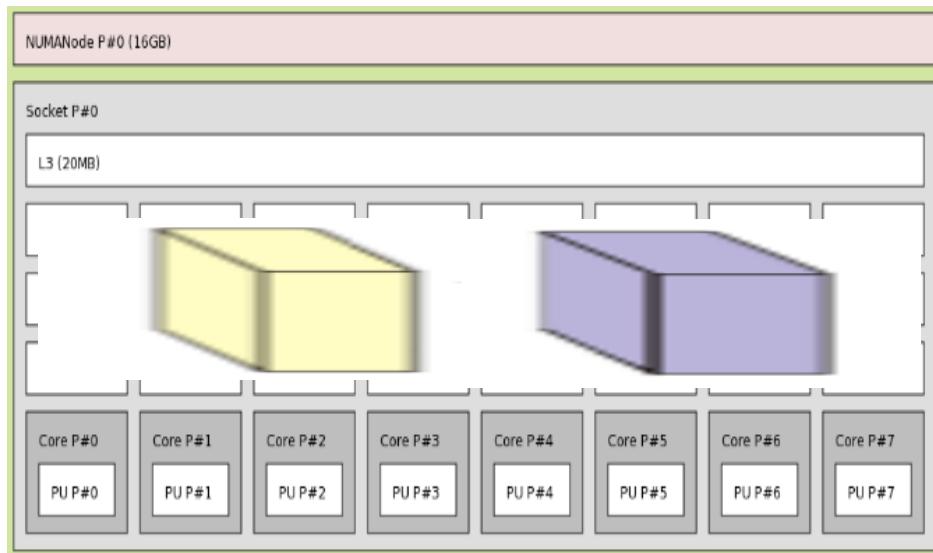


The Intel Xeon E5-2665
Sandy Bridge-EP 2.4GHz



Scuola Internazionale Superiore
di Studi Avanzati

CPU & GPU



The Intel Xeon E5-2665
Sandy Bridge-EP 2.4GHz

~ 8 GBytes



Scuola Internazionale Superiore
di Studi Avanzati

References

- <http://www.ks.uiuc.edu/Research/gpu/>
- <http://indico.ictp.it/event/a14302/other-view?view=ictptimetable> (ICTP - SMR2760)
- <http://www.iac.rm.cnr.it/~massimo/PMC.html>
- CUDA Zone: <https://developer.nvidia.com/cuda-zone>
- CUDA by Example



Scuola Internazionale Superiore
di Studi Avanzati

