Introduction to GPGPUs and to CUDA programming model

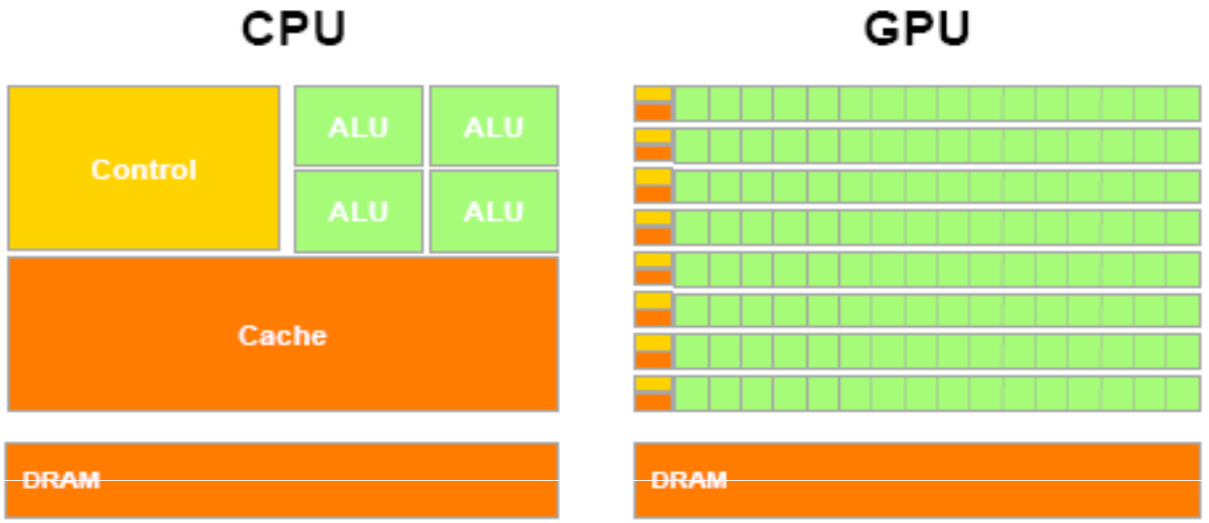
Marzia Rivi
m.rivi@cineca.it

www.cineca.it

Agenda

- *GPGPU architecture*
- *CUDA programming model*
- *CUDA efficient programming*
- *Debugging & profiling tools*
- *CUDA libraries*

GPU vs CPU: different philosophies



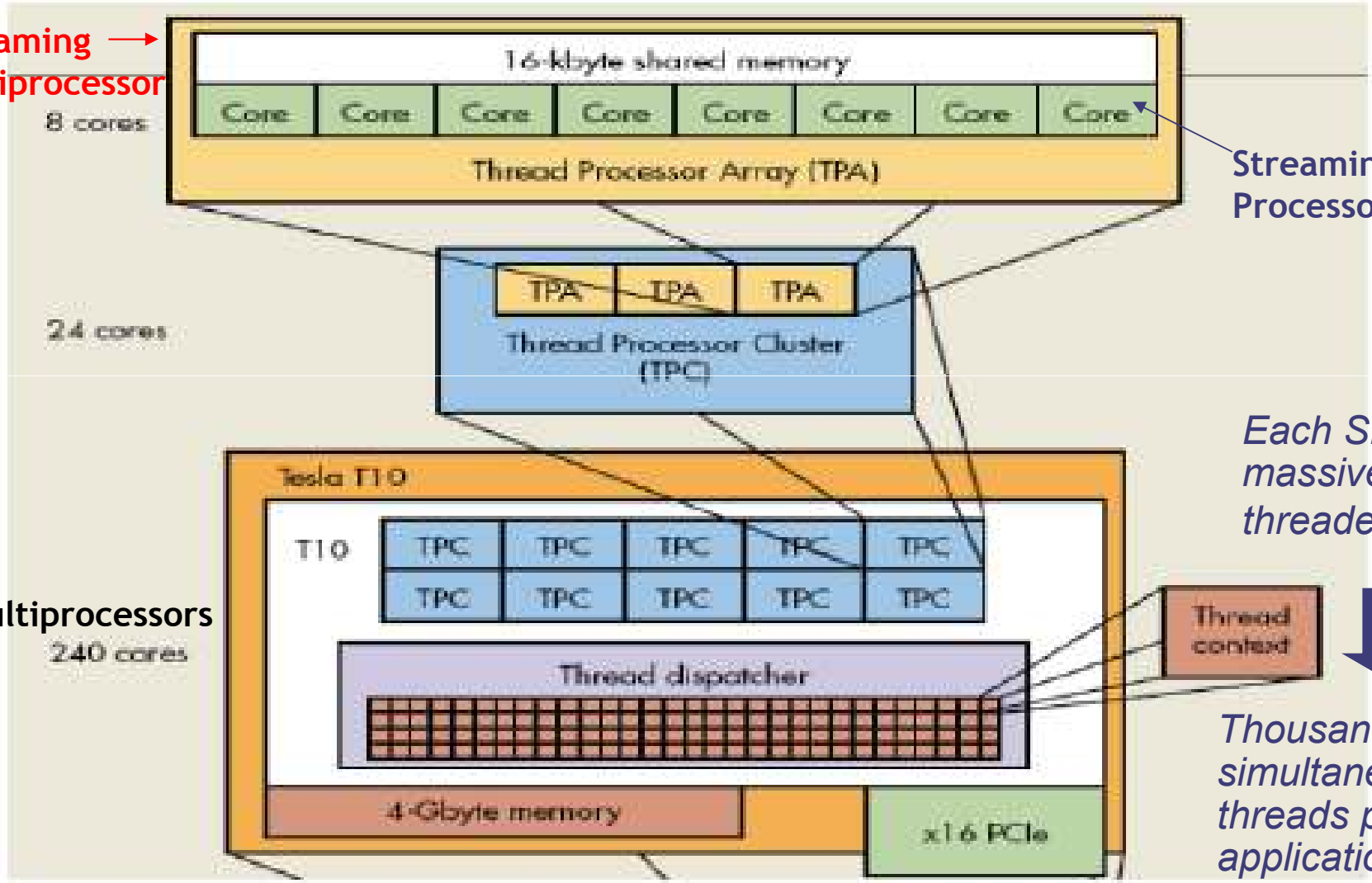
Design of CPUs optimized for sequential code performance:

- *multi-core*
- *sophisticated control logic unit*
- *large cache memories to reduce access latencies*

Design of GPUs optimized for the execution of large number of threads dedicated to floating-points calculations:

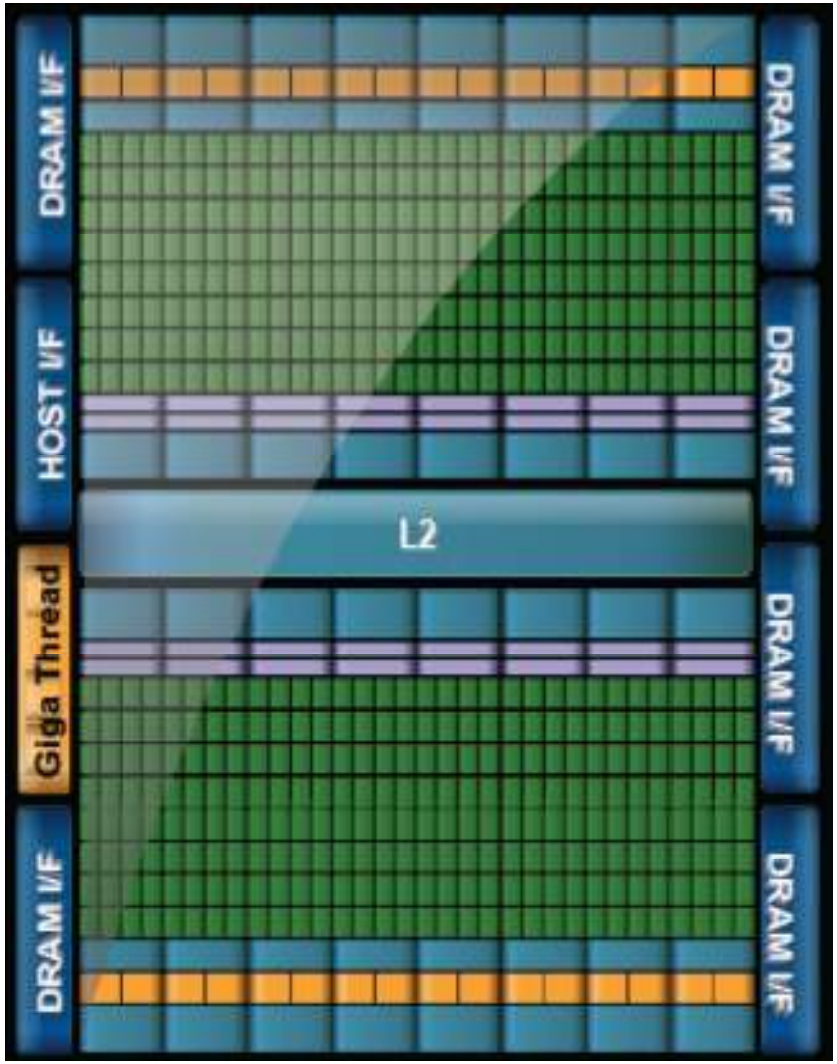
- *many-cores (several hundreds)*
- *minimized the control logic in order to manage leightweight threads and maximize execution throughput*
- *taking advantage of large number of threads to overcome long-latency memory accesses*

GPU NVIDIA Tesla T10 architecture



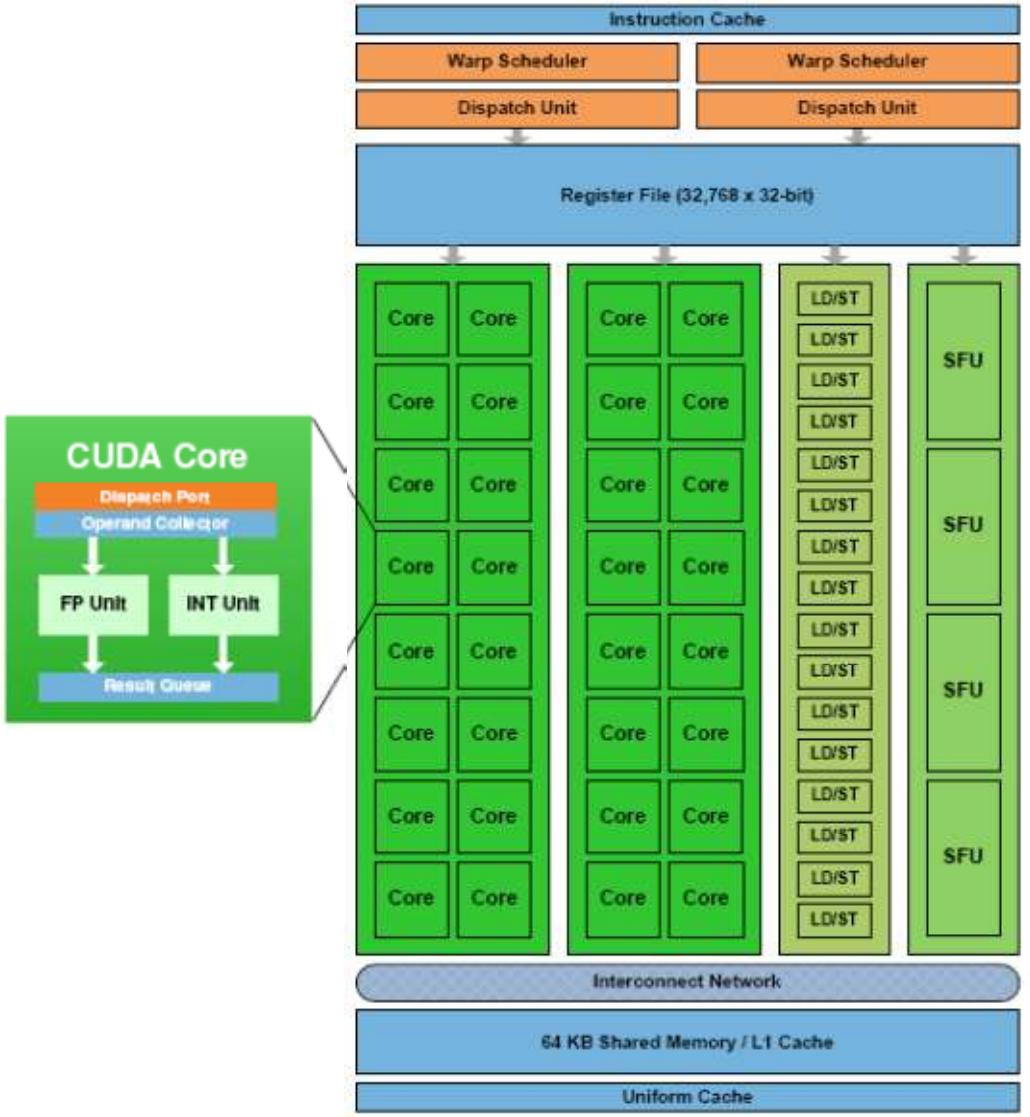
Fermi architecture

- *512 cores (16 SM x 32 SP)*
- *first GPU architecture to support a true cache hierarchy:*
  L1 cache per SM
  unified L2 caches (768 Kb)
- *1.5x Memory Bandwidth (GDDR5)*
- *6 Gb of global memory*
- *48Kb of shared memory*
- *Concurrent Kernels*
- *support C++*



CUDA core architecture

- New IEEE 754-2008 floating point standard
- Fused multiply-add (FMA) instruction for both single and double precision
- Newly designed integer ALU optimized for 64-bit and extended precision operations



Fermi Streaming Multiprocessor (SM)

NVIDIA naming

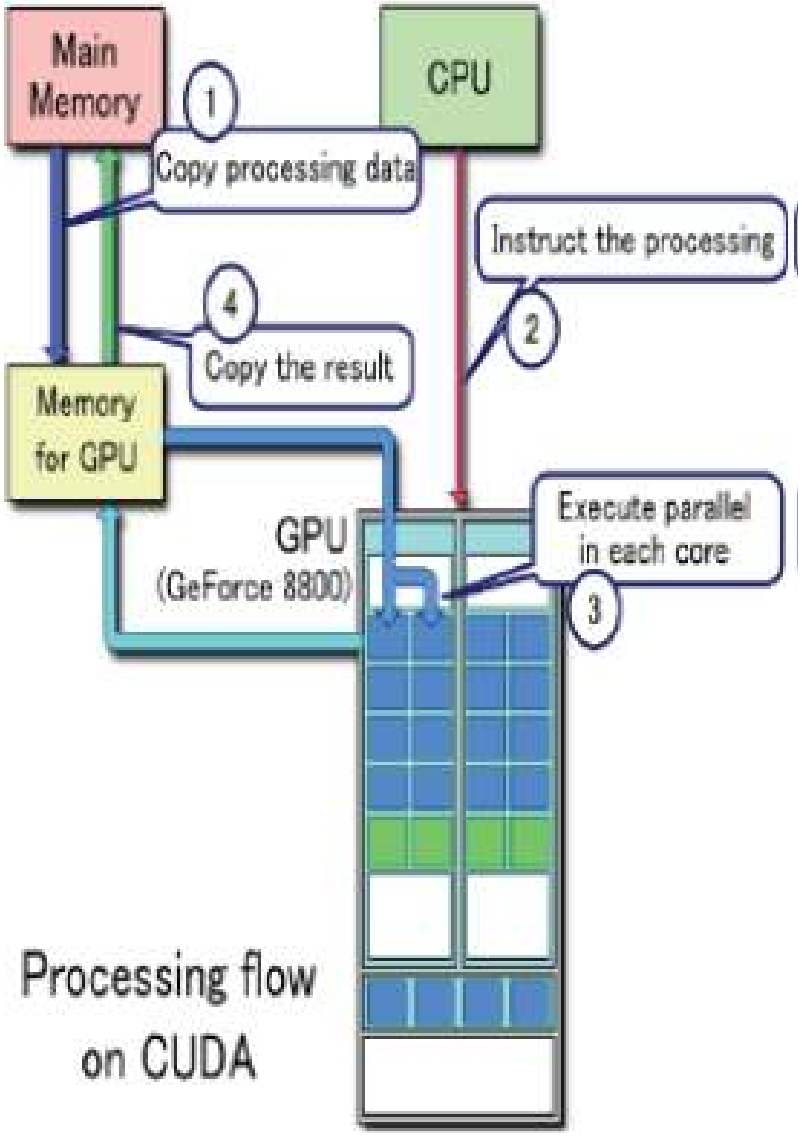
- *Mainstream & laptops: GeForce*
  - Target: videogames and multi-media
- *Workstation: Quadro*
  - Target: graphic professionals who use CAD and 3D modeling applications
  - The surcharge is due to more memory and especially the specific drivers for accelerating applications
- *GPGPU: Tesla*
  - Target: High Performance Computing

There cannot be a GPU without a CPU

GPUs are designed as numeric computing engines, therefore they will not perform well on other tasks.

Applications should use both CPUs and GPUs, where the latter is exploited as a coprocessor in order to speed up numerically intensive sections of the code by a massive fine grained parallelism.

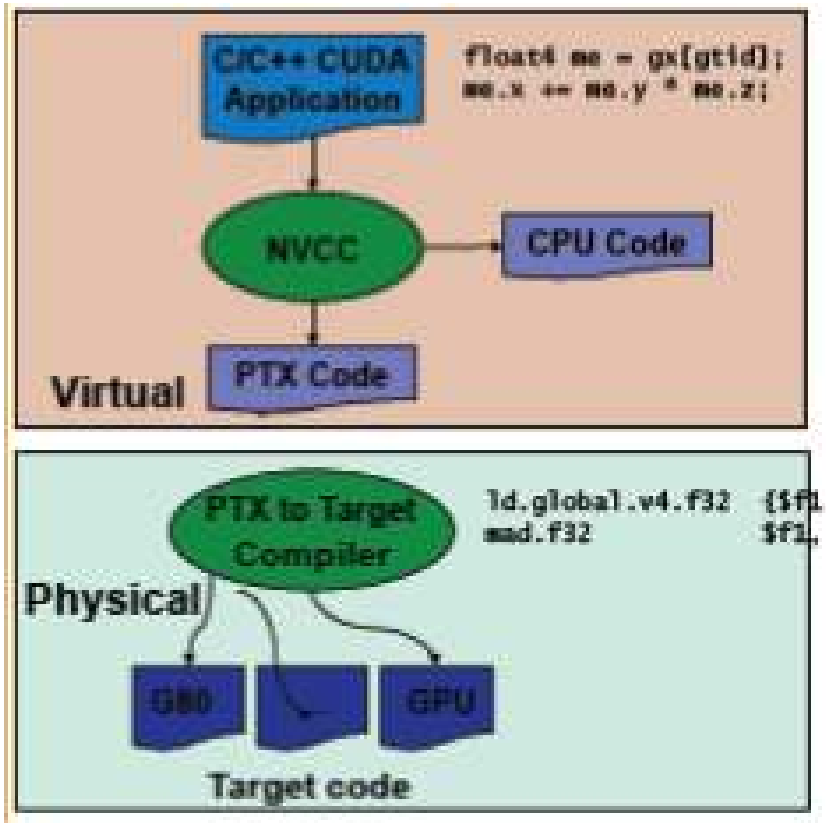
CUDA programming model introduced by NVIDIA in 2007, is designed to support joint CPU/GPU execution of an application.



NVIDIA C compiler

nvcc front-end for compilation:

- *separates GPU code from CPU code*
- *CPU code -> C/C++ compiler (Microsoft, GCC, ecc.)*
- *GPU code is converted in an intermediate language: PTX, then in assembler*
- *link all executables*



CUDA programming model

C*ompute* **U***nified* **D***evice* **A***rchitecture:*

- *extends ANSI C language with minimal extensions*
- *provides application programming interface (API) to manage host and device components*

CUDA program:

- *Serial sections of the code are performed by CPU (host)*
- *The parallel ones (that exhibit rich amount of data parallelism) are performed by GPU (device) in the SPMD mode as CUDA kernels.*
- *host and device have separate memory spaces: programmers need to transfer data between CPU and GPU in a manner similar to "one-sided" message passing.*

Device management

- *Application can query and select GPUs*
  - cudaGetDeviceCount(int *count)
  - cudaSetDevice(int device)
  - cudaGetDevice(int *device)
  - cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
- *Multiple threads can share a device*
- *A single thread can manage multiple devices*
  - cudaSetDevice(i) to select current device
  - cudaMemcpy(…) for peer-to-peer copies

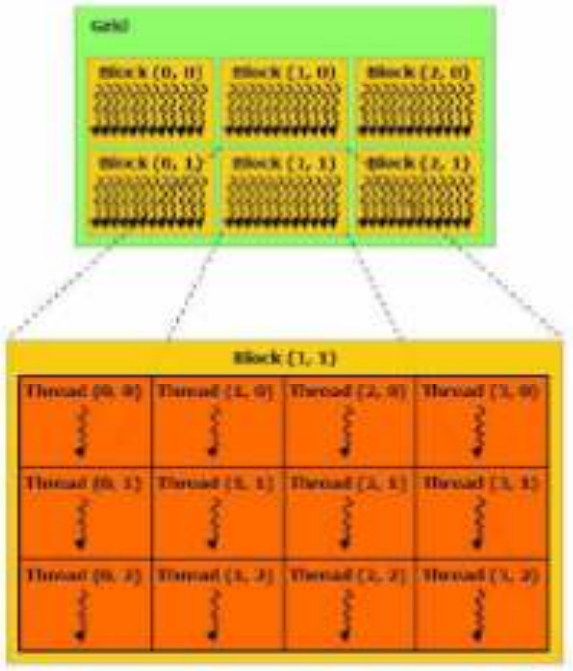
Device management (sample code)

```
int cudadevice;
struct cudaDeviceProp prop;
cudaGetDevice( &cudadevice );
cudaGetDeviceProperties (&prop, cudadevice);
mpc=prop.multiProcessorCount;
mtpb=prop.maxThreadsPerBlock;
shmsize=prop.sharedMemPerBlock;
printf("Device %d: number of multiprocessors
   %d\n , max number of threads per block
   %d\n, shared memory per block %d\n",
   cudadevice, mpc, mtpb, shmsize);
```

CUDA threads organization

A kernel is executed as a **grid** *of many parallel threads.*
They are organized into a two-level hierarchy:

- *a grid is organized as up to 3-dim array of thread blocks*
- *each block is organized into up to 3-dim array of threads*
- *all blocks have the same number of threads organized in the same manner.*



Block of threads:
set of concurrently executing threads that can cooperate among themselves through

- *barrier synchronization,by using the function* `__syncthreads();`
- *shared memory.*

CUDA threads organization

Because all threads in a grid execute the same code, they rely on unique coordinates assigned to them by the CUDA runtime system as built-in preinitialized variables

- *Block ID up to 3 dimensions:*
  (blockIdx.x, blockIdx.y, blockIdx.z)
- *Thread ID within the block up to 3 dimensions:*
  (threadIdx.x, threadIdx.y, threadIdx.z)

The exact organization of a grid is determined by the execution configuration provided at kernel launch.

Two additional variables of type **`dim3`** *(C struct with 3 unsigned integer fields) are declared:*

- *gridDim* ⟶ *dimensions of the grid in terms of number of blocks*
- *blockDim* ⟶ *dimensions of the block in terms of number of threads*

Launching a kernel

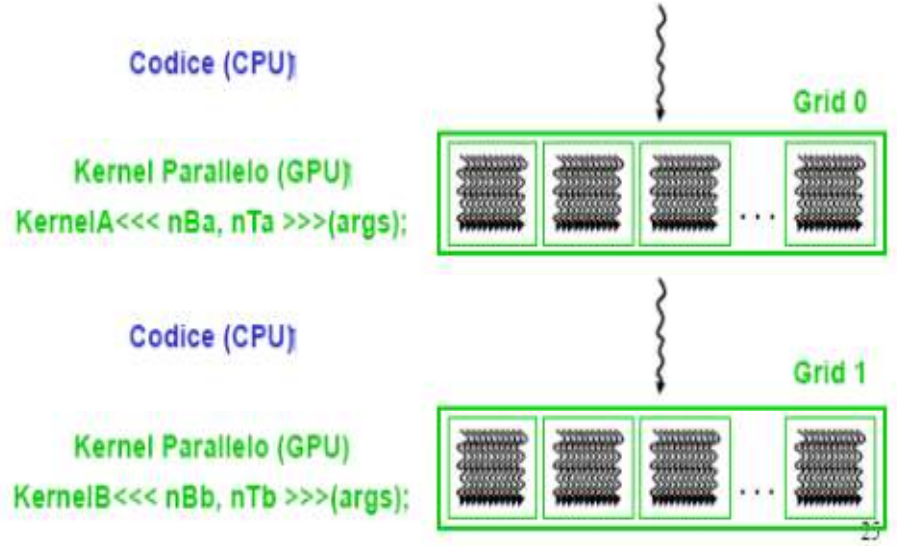
A kernel must be called from the host with the following syntax:

```
__global__ void KernelFunc(…);
dim3 gridDim(100, 50);  // 5000 thread blocks
dim3 blockDim(8, 8, 4); // 256 threads per block

//call the kernel
KernelFunc<<< gridDim, blockDim >>>(<arguments>);
```

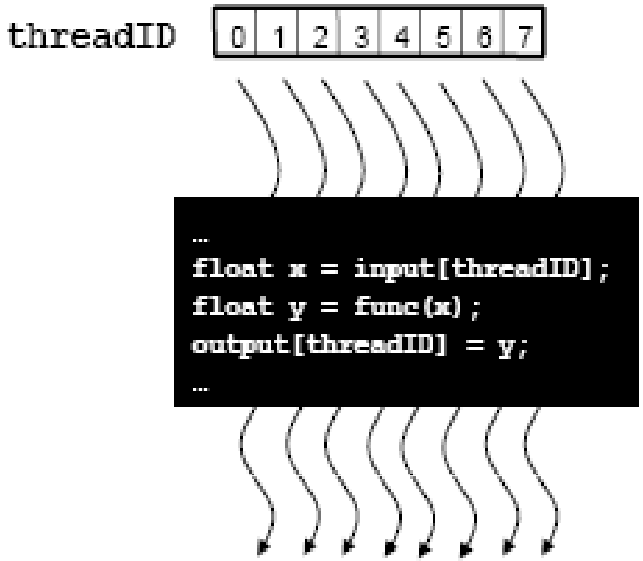
Typical CUDA grids contain thousands to millions of threads.

All kernel calls are asynchronous!



Thread ID computation

The build-in variables are used to compute the global ID of the thread, in order to determine the area of data that it is designed to work on.



- *1D:*
  - int id = blockDim.x * blockIdx.x + threadIdx.x;
- *2D:*
  - int iy = blockDim.y * blockIdx.y + threadIdx.y;
  - int ix = blockDim.x * blockIdx.x + threadIdx.x;
  - int id = iy * dimx + ix;

Kernel example

CPU code

```
void increment_cpu(float *a, float b, int
N)
{
 for (int idx – 0; idx<N; idx++)
 a[idx] = a[idx] + b;
}
void main()
{
  .....
 increment_cpu(a, b, 16);
}
```

CUDA code

```
__global__ void increment_gpu(float *a,
float b, int N)
{
 int idx – blockIdx.x * blockDim.x +
threadIdx.x;
 if( idx < N)
 a[idx] = a[idx] + b;
}
void main()
{
  .....
 increment_gpu<<< 4, 4 >>>(a, b, 16);
}
```

Function modifiers

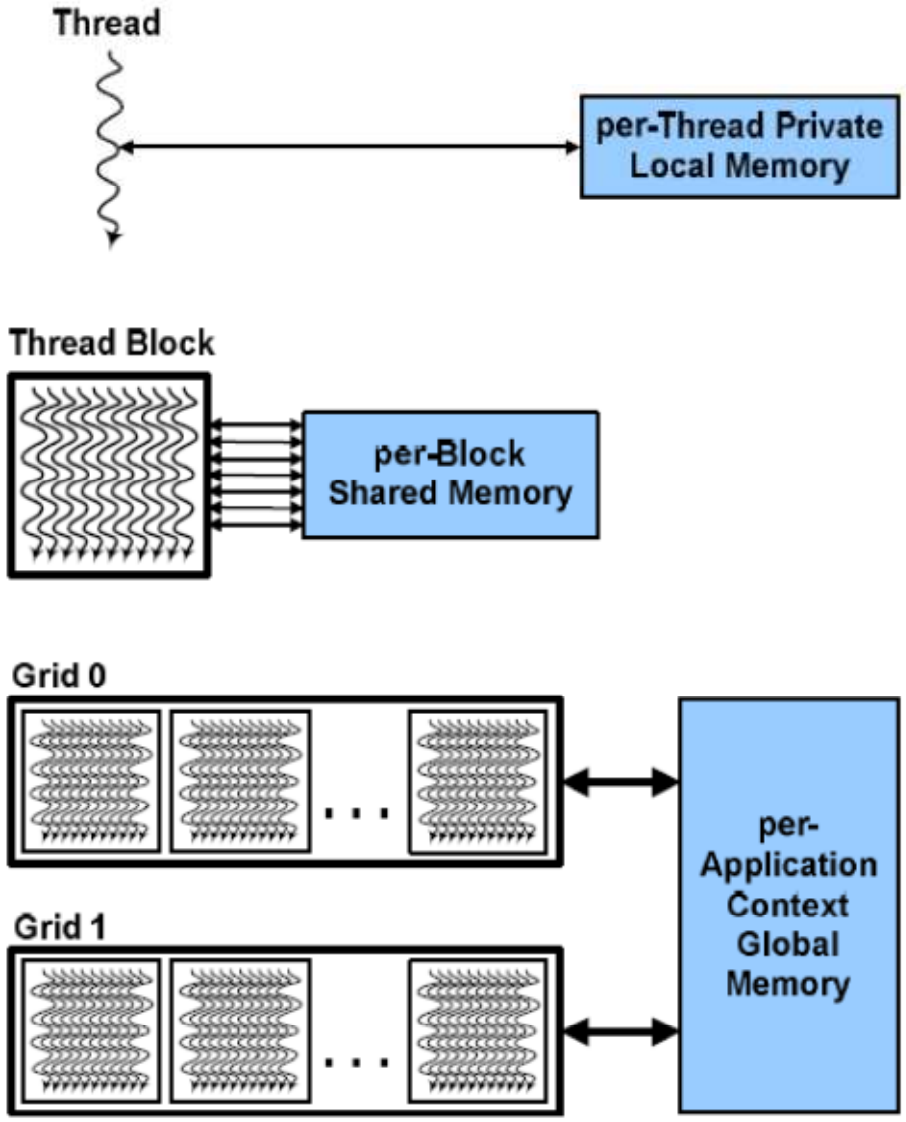
CUDA extends C function declarations with three qualifier keywords.

Function declaration	Executed on the	Only callable from the
`__device__` (*device functions*)	*device*	*device*
`__global__` (*kernel function*)	*device*	*host*
`__host__` (*host functions*)	*host*	*host*

Hierarchy of device memories

CUDA's hierarchy of threads maps to a hierarchy of memories on the GPU:

- *Each thread has some* **registers**, *used to hold automatic scalar variables declared in kernel and device functions, and a* **per-thread private memory space** *used for register spills, function calls, and C automatic array variables*
- *Each thread block has a* **per-block shared memory space** *used for inter-thread communication, data sharing, and result sharing in parallel algorithms*
- *Grids of thread blocks share results in* **global memory space**



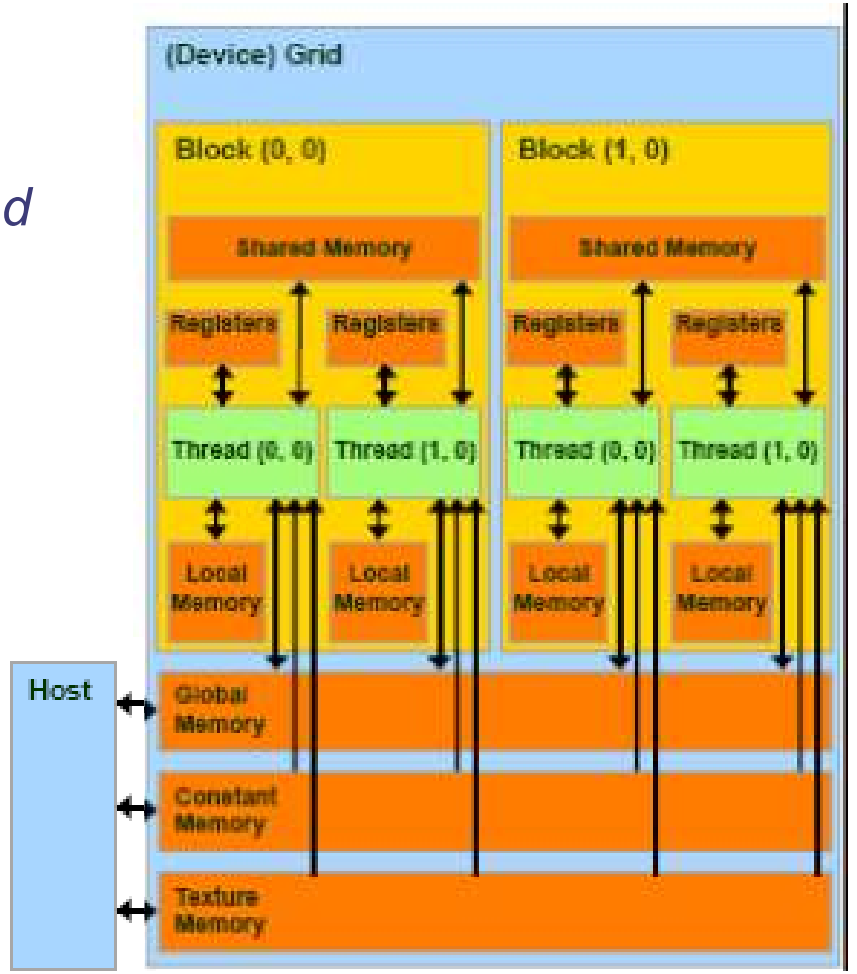
CUDA device memory model

on-chip memories:

- *registers (~8kB) → SP*
- *shared memory (~16kB) → SM*
- *they can be accessed at very high speed in a highly parallel manner.*

per-grid memories:

- *global memory (~4GB)*
  - *long access latencies (hundreds of clock cycles)*
  - *finite access bandwith*
- *constant memory (~64kB)*
  - *read only*
  - *short-latency (cached) and high bandwith when all threads simultaneously access the same location*
- *texture memory (read only)*
- *CPU can transfer data to/from all per-grid memories.*



Local memory is implemented as part of the global memory, therefore has a long access latencies too.

CUDA variable qualifiers

Variable declaration	memory	lifetime	scope
Automatic scalar variables	*register*	*kernel*	*thread*
Automatic array variables `__device__ __local__`	*local*	*kernel*	*thread*
`__device__ __shared__`	*shared*	*kernel*	*block*
`__device__`	*global*	*application*	*grid*
`__device__ __constant__`	*constant*	*application*	*grid*

- *Global variables are often used to pass information from one kernel to another.*
- *Constant variables are often used for providing input values to kernel functions.*

Shared memory allocation

- **Static modality**
  inside the kernel:
  __shared__ float f[100];

- **Dynamic modality**
  in the execution configuration of the kernel, define the number of bytes to be allocated per block in the shared memory :

  kernel<<<DimGrid, DimBlock, SharedMemBytes>>>(…);

  while inside the kernel:
  extern __shared__ float f[ ];

Device memory allocation

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

*cudaMalloc(void** pointer, size_t nbytes)*

- *It allocates a part of the device global memory*
- *The first parameter is the address of a generic pointer variable that must point to the allocated object*
  - ✓ *it should be cast to (void**)!*
- *The second parameter is the size of the object to be allocated, in terms of bytes*

cudaFree(void pointer)*

- *It frees the storage space of the object*

cudaMemset

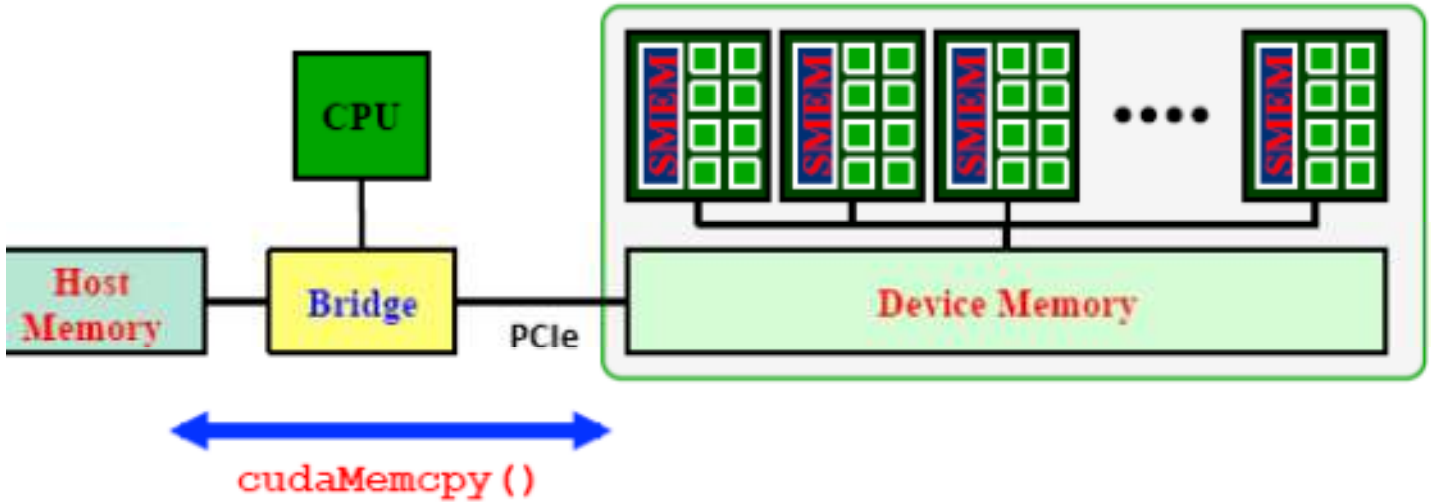
*cudaMemset (void * devPtr, int value, size_t count)*

Fills the first count bytes of the memory area pointed to by devPtr with the constant byte value value.

Cuda version of the C memset() function.

devPtr - Pointer to device memory
value - Value to set for each byte of specified memory
count - Size in bytes to set

Data transfer CPU-GPU



API blocking functions for data transfer between memories:

```
cudaMemcpy(dM, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(M, dM, size, cudaMemcpyDeviceToHost);
```

Destination — source data — number of bytes — symbolic constant indicating the direction

Constant memory

cudaMemcpyToSymbol(const char * symbol,
const void * src,
size_t count,
size_t offset,
enum cudaMemcpyKind kind)

symbol - symbol destination on device, it can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space.

src - source memory address

count - size in bytes to copy

offset - offset from start of symbol in bytes

kind - type of transfer, it can be either
cudaMemcpyHostToDevice or
cudaMemcpyDeviceToDevice

Minimize transfers

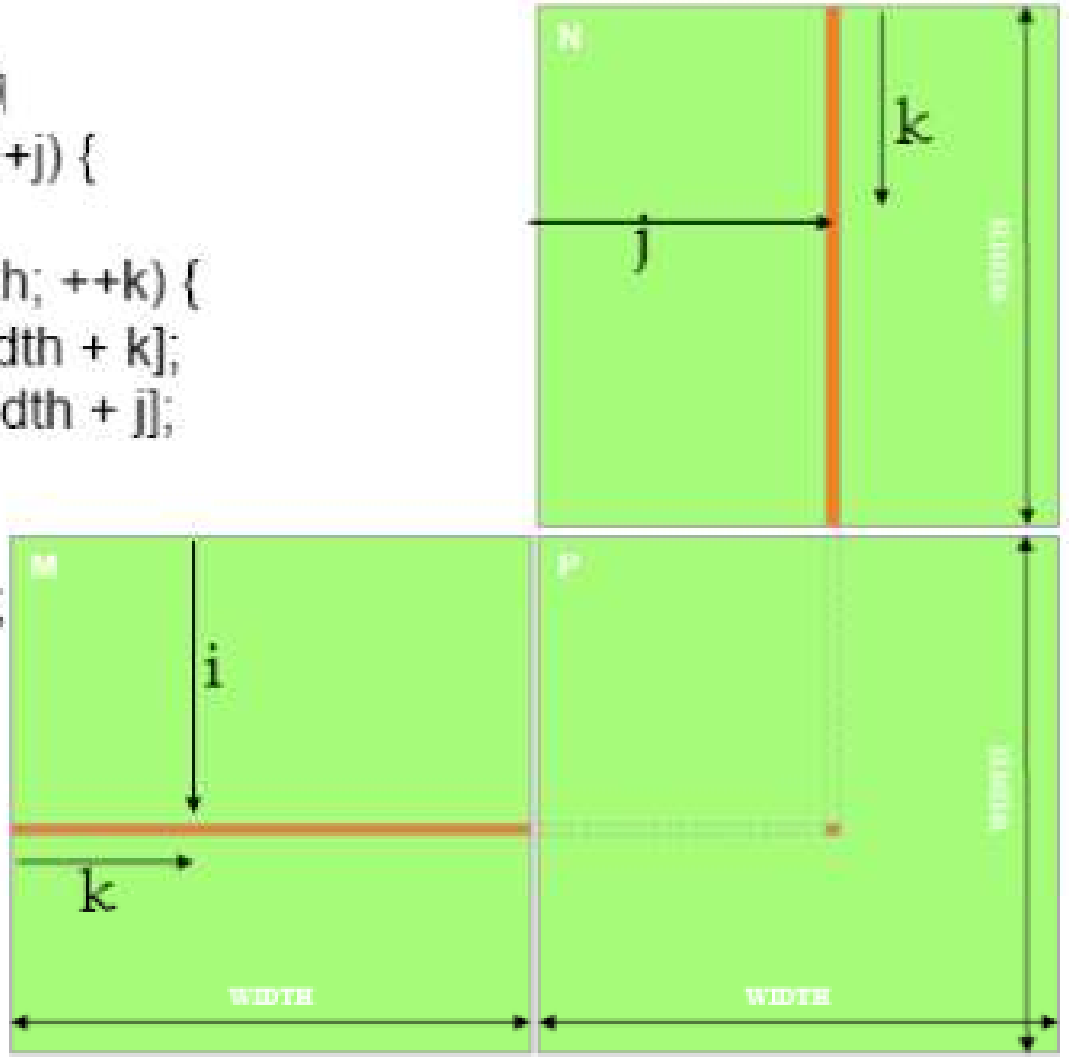
memory bandwith on GPU = 150 GB/s
memory bandwith on PCIe ~= 6 GB/s ⇨ *Minimize number of copies!*

- *Keep as much data as possible on the GPU memory*
- *Sometimes for the GPU is cheaper to recalculate some data rather than transfer them from the CPU*

Matrix- matrix multiplication example

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```

P = M*N



CUDA parallelization:
each thread computes an element of P

Matrix- matrix multiplication host code

```
void MatrixMultiplication(float* M, float *N, float *P, int width)
{
  size_t size = width*width*sizeof(float);
  float* Md, Nd, Pd;
  // transfer M and N to the device memory
  cudaMalloc((void**)&Md, size);
  cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
  cudaMalloc((void**)&Nd, size);
  cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
  // allocate P on the device
  cudaMalloc((void**)&Pd, size);
  // kernel invocation
  dim3 gridDim(1,1);
  dim3 blockDim(width,width);
  MNKernel<<<dimGrid, dimBlock>>>(Md,Nd,Pd,width);
  // transfer P from the device to the host
  cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
  // free device matrices
  cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

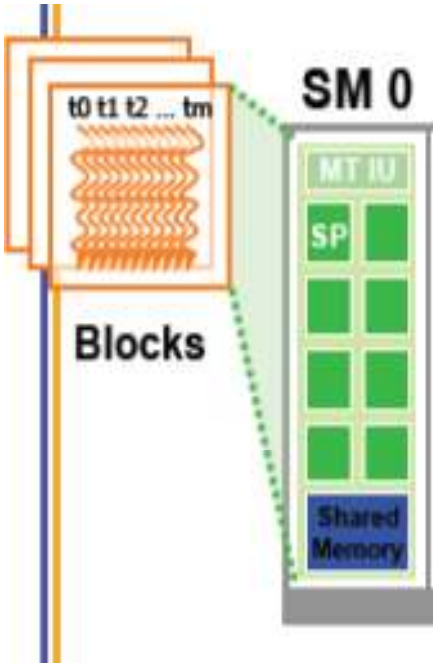
Matrix- matrix multiplication device code

```
__global__ void MNKernel(float* Md, float *Nd, float *Pd,
  int width)
{
  // 2D thread ID
  int col = threadIdx.x;
  int row = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the
  // thread
  float Pvalue = 0;
  for (int k=0; k < width; k++)
     Pvalue += Md[row * width + k] * Nd[k * width + col];

  // write the matrix to device memory (each thread
  // writes one element)
  Pd[row * width + col] = Pvalue;
}
```

Threads execution



CUDA's hierarchy of threads/memories maps to the hierarchy of processors on the GPU:

- *a GPU executes one or more kernel grids;*
- *a streaming multiprocessor (SM) executes one or more thread blocks;*
- *a streaming processor (SP) in the SM executes threads.*

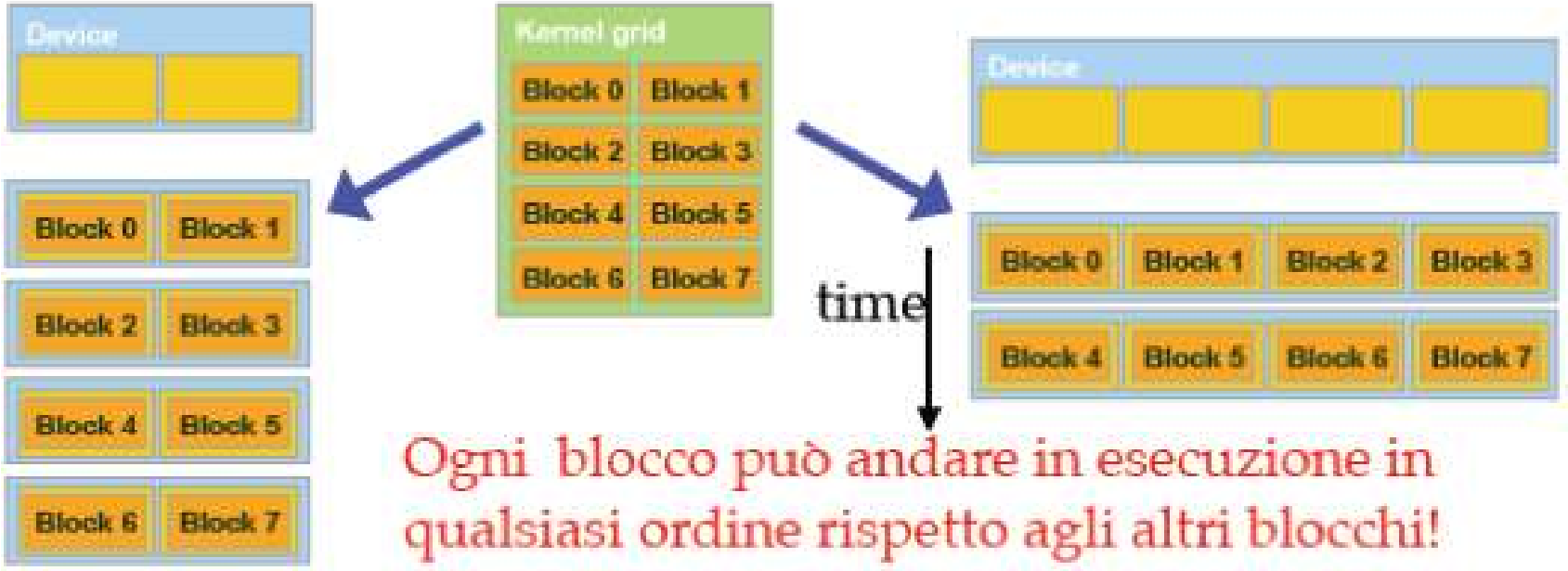
A maximum number of blocks can be assigned to each SM (8 for Tesla T10)

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.

Transparent scalability

By not allowing threads in different blocks to synchronize with each other, 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.



Resource limitations

Tesla T10: 1 miliardo e mezzo di transistor (più *cores*, meno *cache*…)

Informazioni riportate dal comando *deviceQuery*:

Major revision number:	1
Minor revision number:	3
Total amount of global memory:	4294770688 bytes
Number of multiprocessors:	30
Number of cores:	240
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	16384 bytes
Total number of registers available per block:	16384
Warp size:	32
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 *x* 512 *x* 64
Maximum sizes of each dimension of a grid:	65535 *x* 65535 *x* 1
Clock rate:	1.30 GHz
Concurrent copy and execution:	Yes

Memory Bandwidth: 102 Gbyte/sec.!

Matrix- matrix multiplication example

Limitation: a block can have up to 512 threads (for Tesla T10). Therefore the previous implementation can compute square matrices of order less than 32.

Improvement:

- *use more blocks by breaking matrix Pd into square tiles*
- *all elements of a tile are computed by a block of threads*
- *each thread still calculates one Pd element but it uses its blockIdx values to identify the tile that contains its element.*

Matrix- matrix multiplication example

```
__global__ void MNKernel(float* Md, float *Nd, float *Pd, int
  width)
{
   // 2D thread ID
   int col = blockIdx.x*TILE_WIDTH + threadIdx.x;
   int row = blockIdx.y*TILE_WIDTH + threadIdx.y;

   // Pvalue stores the Pd element that is computed by the
  thread
   float Pvalue = 0;
   for (int k=0; k < width; k++)
     Pvalue += Md[row * width + k] * Nd[k * width + col];

   Pd[row * width + col] = Pvalue;
}
```

Kernel invocation:

```
dim3 gridDim(width/TILE_WIDTH,width/TILE_WIDTH);
dim3 blockDim(TILE_WIDTH,TILE_WIDTH);
MNKernel<<<dimGrid, dimBlock>>>(Md,Nd,Pd,width);
```

Matrix- matrix multiplication example

Which is the optimal dimension of the block (i.e. TILE_WIDTH)?

Knowing that each SM of a Tesla T10 can have up to 1024 threads, we have

- *8x8 = 64 threads ⇨ 1024/64 = 12 blocks to fully occupy an SM; but we are limited to 8 blocks in each SM therefore we will end up with only 64x8 = 512 threads in each SM.*
- *16x16 = 256 threads ⇨ 1024/256 = 4 blocks*

  we will have full thread capacity in each SM.
- *32x32 = 1024 threads per block which exceed the limitation of up to 512 threads per block.*

➡ *TILE_WIDTH = 16 !*

Global memory access efficiency

Although having many threads available for execution can theoretically tolerate long memory access latency, one can easily run into a situation where traffic congestion prevents all but few threads from making progress, thus rendering some SM idle!

A common strategy for reducing global memory traffic (i.e. increasing the number of floating-point operations performed for each access to the global memory) is to partition the data into subsets called tiles such that each tile fits into the shared memory and the kernel computations on these tiles can be done independently of each other.

In the simplest form, the tile dimensions equal those of the block.

Matrix- matrix multiplication example

In the previous kernel:

thread(x,y) of block(0,0) access the elements of **`Md`** *row x and* **`Nd`** *column y from the global memory.*

⇨ *thread(0,0) and thread(0,1) access the same* **`Md`** *row 0*

What if these threads collaborate so that the elements of this row are only loaded from the global memory once? We can reduce the total number of accesses to the global memory by half!

Basic idea:

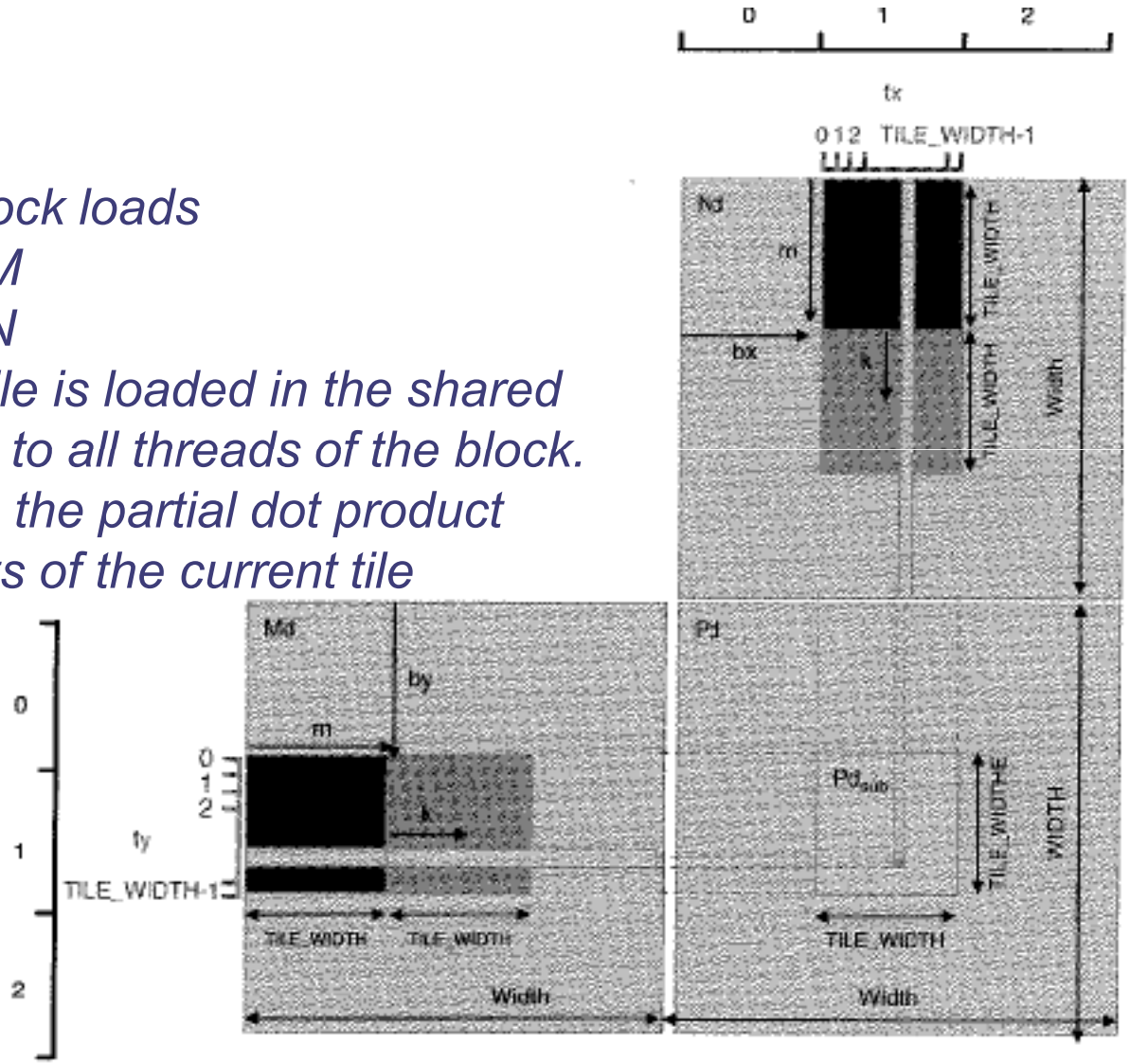
- *to have the threads within a block collaboratively load* **`Md`** *and* **`Nd`** *elements into the shared memory before they individually use these elements in their dot product calculation.*
- *The dot product performed by each thread is now divided into phases: in each phase all threads in a block collaborate to load a tile of Md and a tile of Nd into the shared memory and use these values to compute a partial product. The dot product would be performed in* **`width/TILE_WIDTH`** *phases.*
- *the reduction of the accesses to the global memory is by a factor of TILE_WIDTH.*

Matrix- matrix multiplication example

1 phase → 1 tile

For each phase:

- *Each thread of the block loads*
  1 element of a tile of M
  1 element of a tile of N
  at the end the entire tile is loaded in the shared memory and is visible to all threads of the block.
- *Each thread compute the partial dot product involving the elements of the current tile*



Matrix- matrix multiplication example

```
__global__ void MNKernel(float* Md, float *Nd, float *Pd, int width)
{
   __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
   __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    // 2D thread ID
    int tx = threadIdx.x; int ty = threadIdx.y;
    int col = blockIdx.x*TILE_WIDTH + tx;
    int row = blockIdx.y*TILE_WIDTH + ty;

    float Pvalue = 0;
    // Loop over the Md and Nd tiles required to compute the Pd element
    for (int m=0; m < width/TILE_WIDTH; m++)
    {//collaborative loading of Md and Nd tiles into shared memory
       Mds[ty][tx] = Md[row*width + (m*TILE_WIDTH + tx)];
       Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*width + col];
       __syncthreads();

       for (int k=0; k < TILE_WIDTH; k++)
           Pvalue += Mds[ty][k] * Nds[k][tx];
       __syncthreads();
    }

    Pd[row * width + col] = Pvalue;
}
```

Memory as a limiting factor to parallelism

The limited amount of CUDA memory limits the number of threads that can simultaneously reside in the SM!

For the matrix multiplication example, shared memory can become a limiting factor:

TILE_WIDTH = 16 ⟹ *each block requires 16x16x4 = 1kB of storage for* **`Mds`**
+ 1kB for **`Nds`**
⟹ *2kB of shared memory per block*

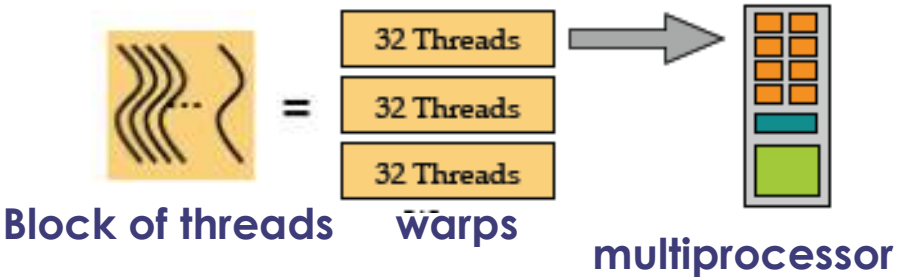
The 16-kB shared memory allows 8 blocks to simultaneously reside in an SM. Ok!
But the maximum number of threads per SM is 1024 (for Tesla T10)

➡ *only 1024/256 = 4 blocks are allowed in each SM*
only 4 x 2kB = 8kB of the shared memory will be used.

Hint: Use occupancy calculator

Thread scheduling

Once a block is assigned to a SM, it is further partitioned into 32-thread units called warps.
Warps are scheduling units in SM: all threads in a same warp execute the same instruction when the warp is selected for execution (Single-Instruction, Multiple-Thread)

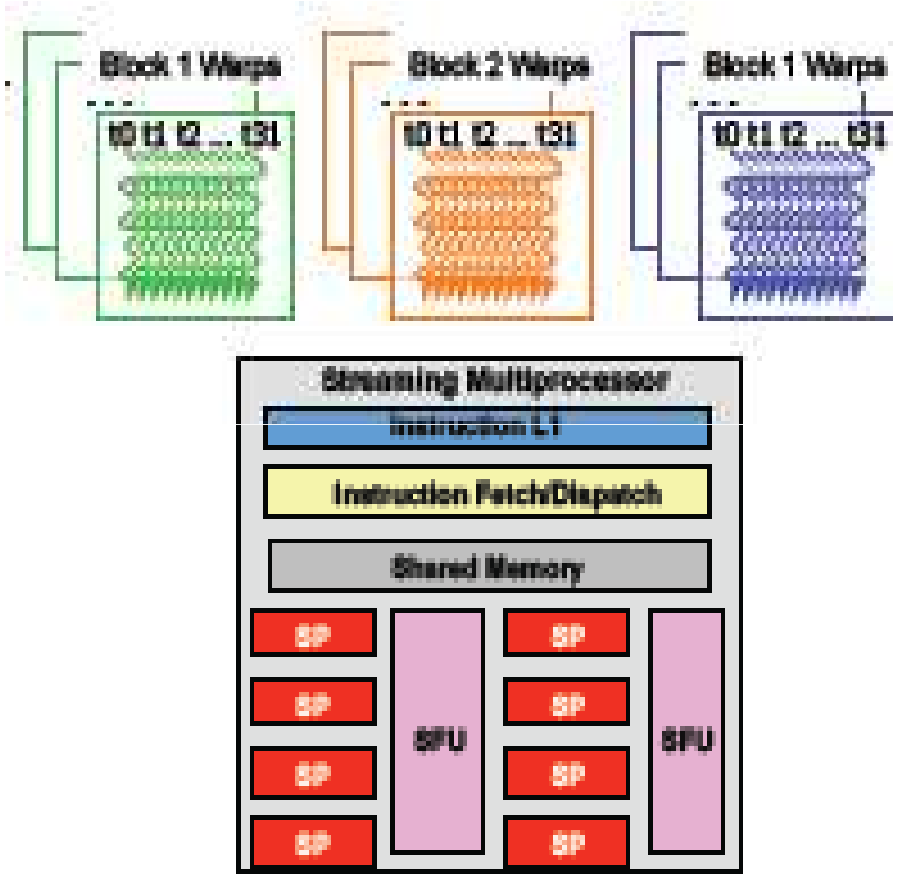


Threads often execute long-latency operations:

- *global memory access*
- *pipelined floating point arithmetics*
- *branch instructions*

It is convenient to assign a large number of warps to each SM, because the long waiting time of some warp instructions is hidden by executing instructions from other warps. Therefore the selection of ready warps for execution does not introduce any idle time into the execution timeline (zero-overhead thread scheduling).

Control flow



Try to avoid if-then-else !

The hardware executes an instruction for all threads in the same warp before moving to the next instruction (SIMT).

It works well when all threads within a warp follow the same control flow path when working their data.

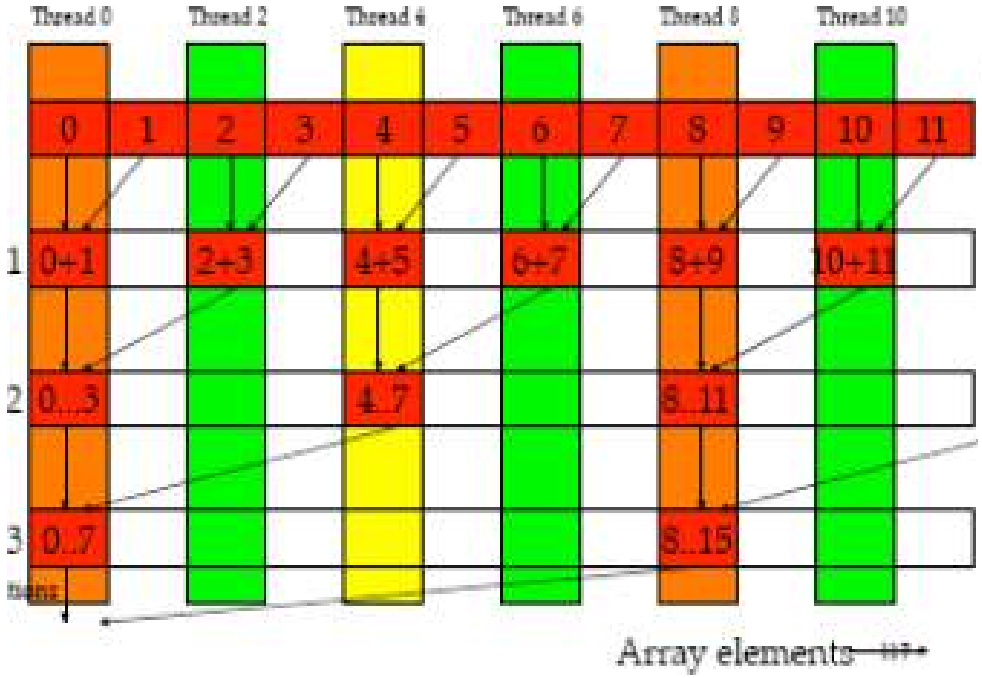
When threads in the same warp follow different paths of control flow, we say that these threads diverge in their execution.

For an if-then-else construct the execution of the warp will require multiple passes through the divergent paths.

Vector reduction example (within a thread block)

An if-then-else construct can result in thread divergence when its decision condition is based on **`threadIdx`** *values.*

A sum reduction algorithm extracts a single value from an array of values in order to sum them. Within a block exploit the shared memory!

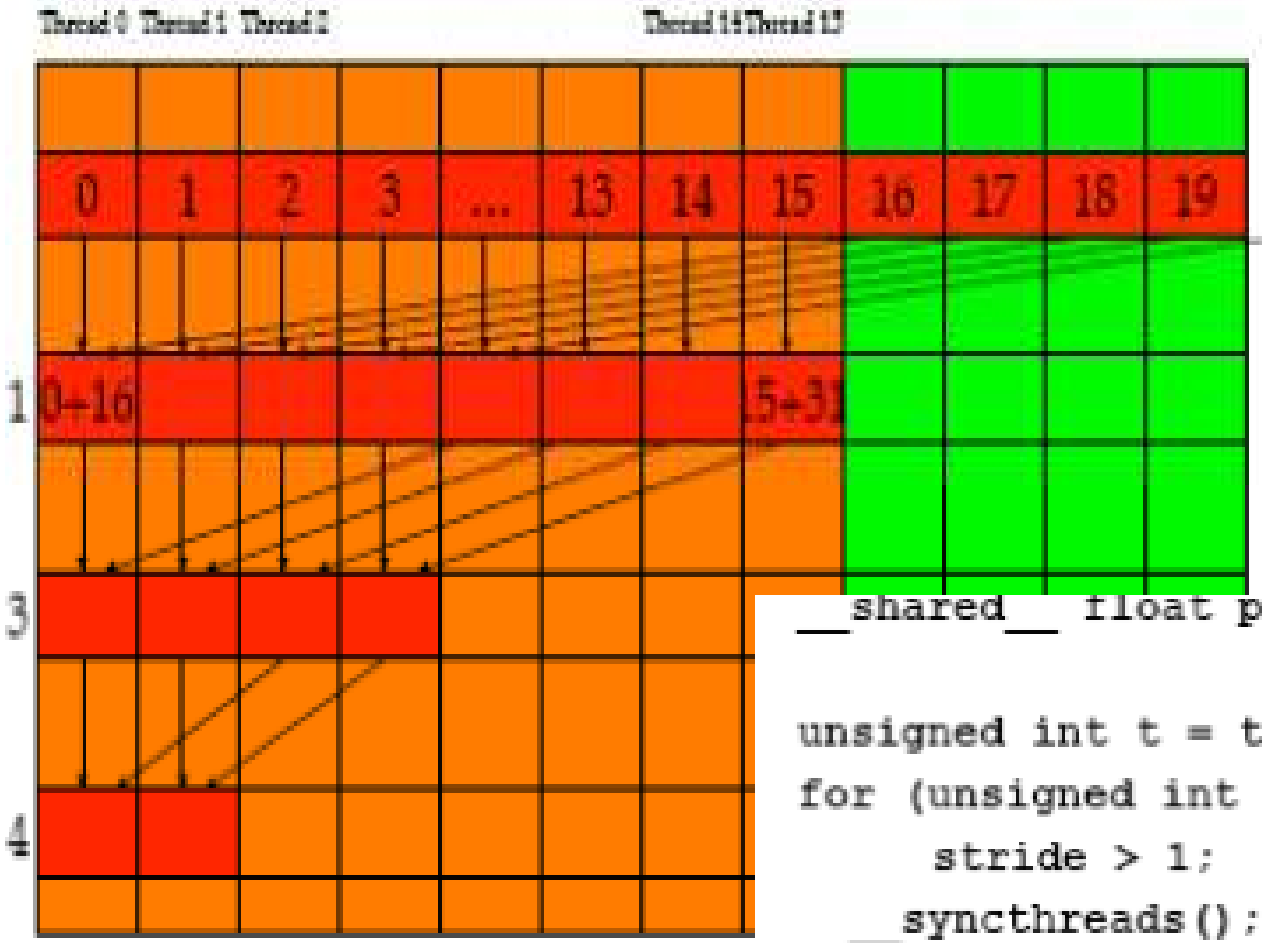



```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2) {
   __syncthreads();
  if (t % (2*stride) == 0)
     partialSum[t] += partialSum[t+stride];
}
```

There is thread divergence!

Vector reduction example



Instead of adding neighbor elements in the first round, add elements that are half a section away from each other and so on.

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x;
      stride > 1;  stride >> 1) {
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
}
```

No divergence until partial sums involve less than 32 elements (because of the warp size)

Error checking

All runtime functions return an error code of type cudaError_t.
No error is indicated as cudaSuccess.

char cudaGetErrorString(cudaError_t code)*
returns a string describing the error:

For asynchronous functions (i.e. kernels, asynchronous copies) the only way to check for errors just after the call is to synchronize: cudaDeviceSynchronize()

Then the following function returns the code of the last error:
cudaError_t cudaGetLastError()

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Reference

http://developer.nvidia.com/cuda

- *CUDA Programming Guide*
- *CUDA Zone – tools, training, webinars and more*

NVIDIA Books:

- *"Programming Massively Parallel Processors", D.Kirk - W.W. Hwu*
- *"CUDA by example", J.Sanders - E. Kandrot*