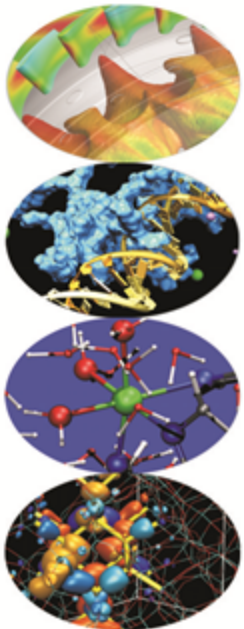
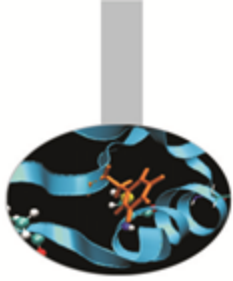


# Introduction to GPGPUs and to CUDA programming model

Gabriele Fatigati

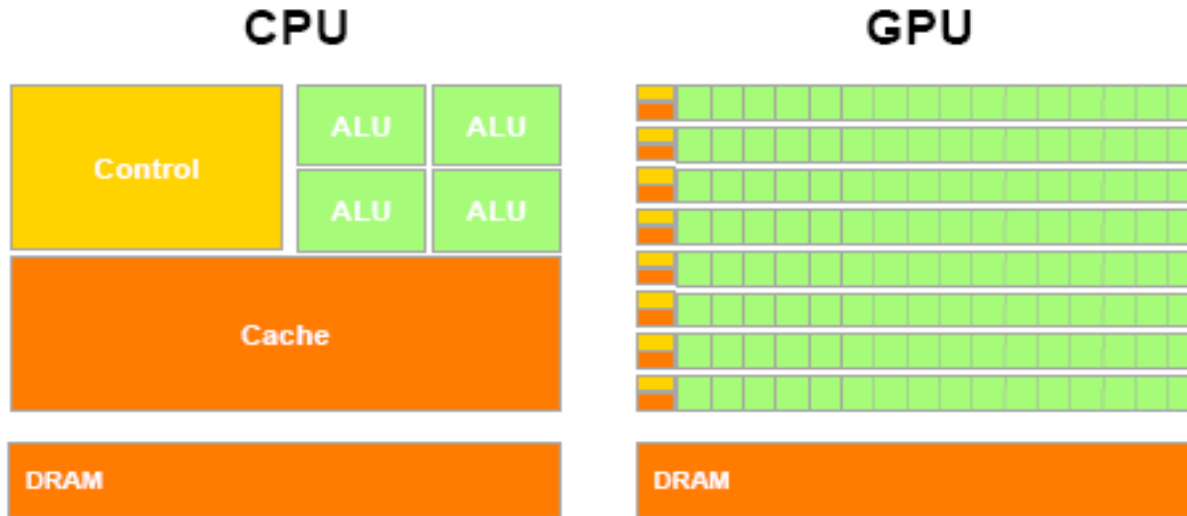
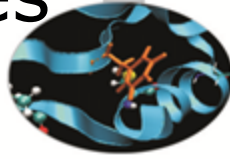


# Agenda



- 🔑 GPGPU architecture
- 🔑 CUDA programming model
- 🔑 CUDA efficient programming

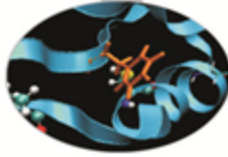
# GPU vs CPU: different philosophies



- Design of CPUs optimized for sequential code performance:
- multi-core
- sophisticated control logic unit
- large cache memories to reduce access latencies

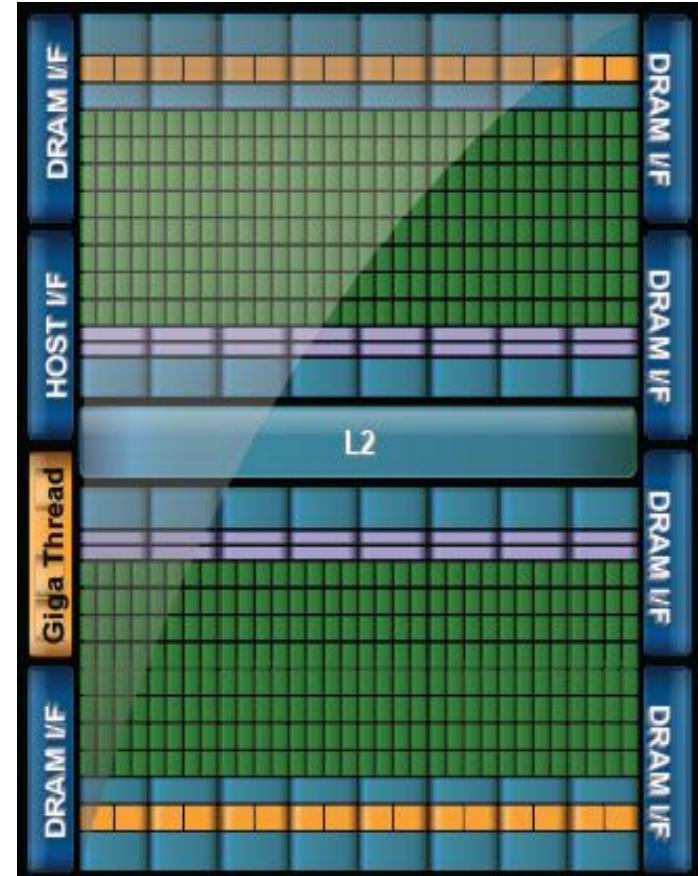
Design of GPUs optimized for the execution of large number of threads dedicated to floating-points calculations:

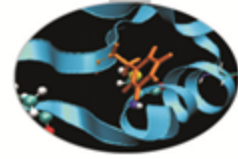
- many-cores (several hundreds)
- minimized the control logic in order to manage lightweight threads and maximize execution throughput
- taking advantage of large number of threads to overcome long-latency memory accesses






# Fermi architecture

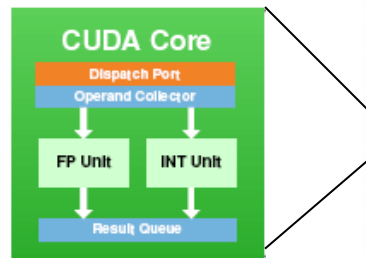
- 512 cores  
(16 SM (Streaming Multiprocessor x 32 SP (Streaming processor))
- first GPU architecture to support a true cache hierarchy:
  - L1 cache per SM
  - unified L2 caches (768 KB)
- Memory Bandwidth (GDDR5)  
148 GB/s (ECC off)
- 6 GB of global memory
- 48KB of shared memory
- Concurrent Kernels execution
- support C++



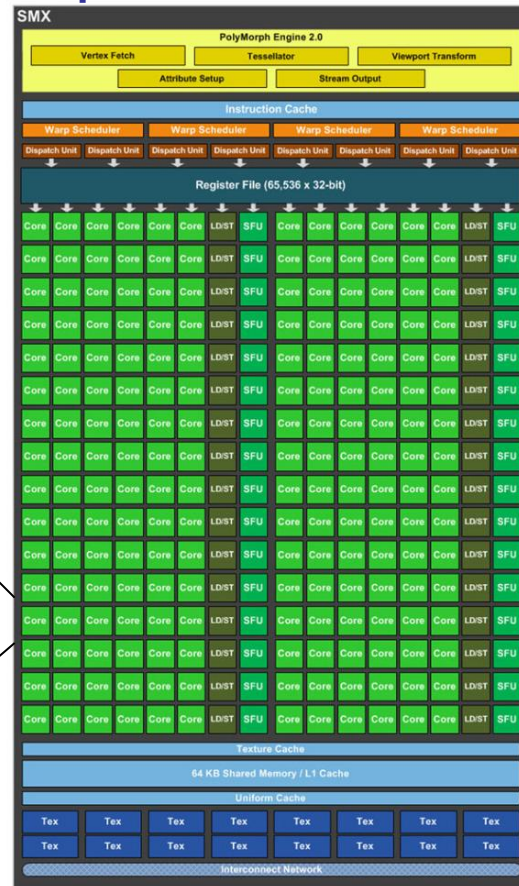


# CUDA core architecture

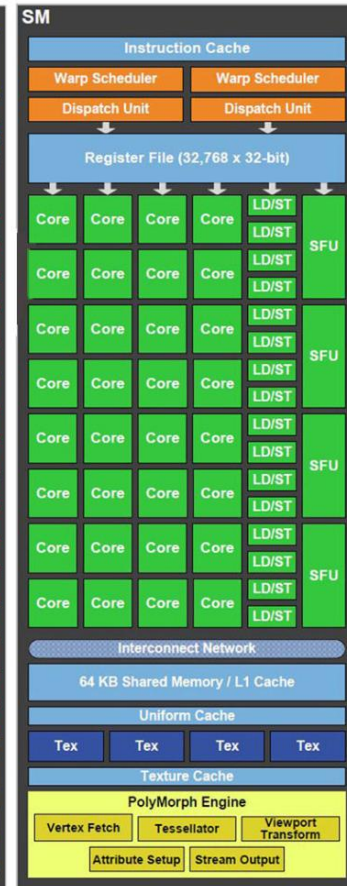
- 
**New IEEE 754-2008 floating point standard**
  
- 
**Fused multiply-add (FMA) instruction for both single and double precision**
  
- 
**Newly designed integer ALU optimized for 64-bit and extended precision operations**



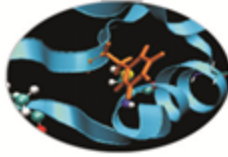
**Kepler SMX**



**Fermi SM**



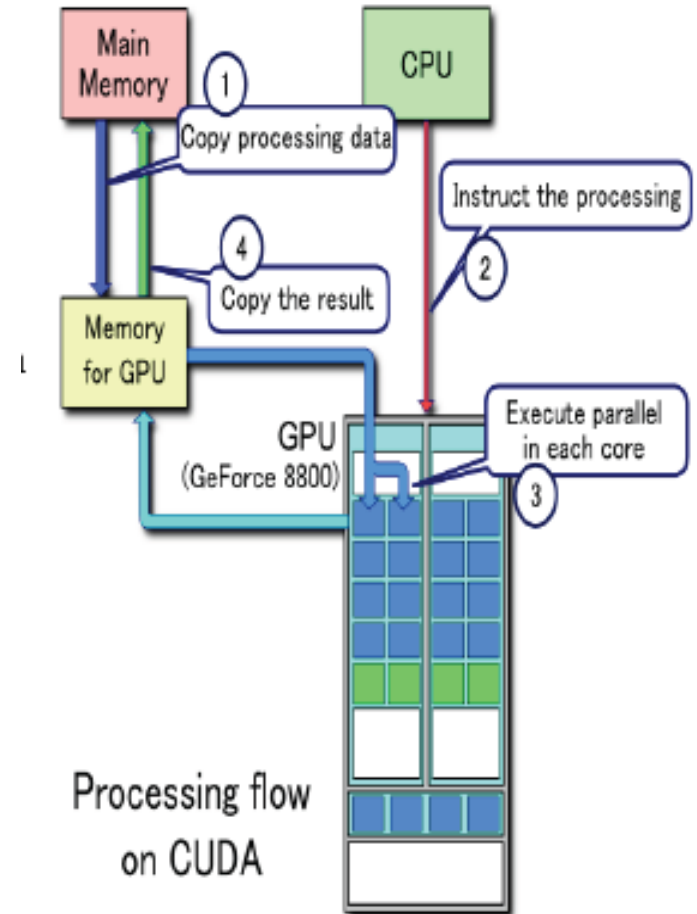
# There cannot be a GPU without a CPU



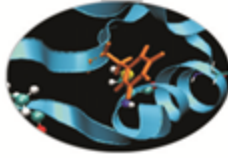
GPUs are designed as numeric computing engines, therefore they will not perform well on other tasks.

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.







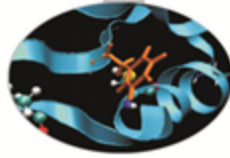
## Compute **U**nified **D**evice **A**rchitecture:

- ⌚ 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 SIMD mode as **CUDA kernels**.
- ⌚ host and device have separate memory spaces: programmers need to transfer data between CPU and GPU.

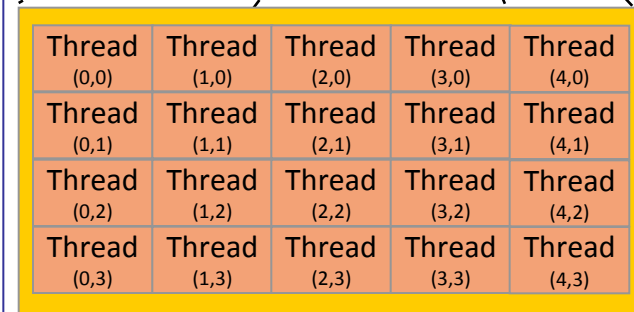
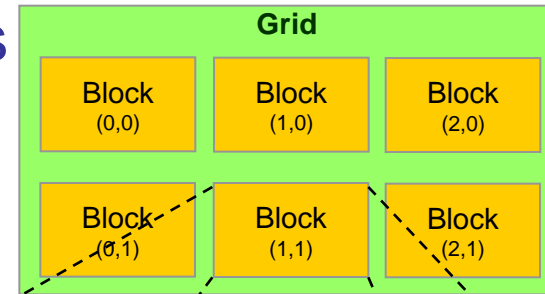
# CUDA threads organization



A kernel is executed as a **grid** of many parallel threads.

They are organized into a two-level hierarchy:

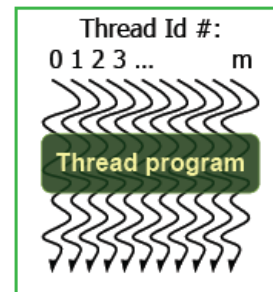
- 🔑 a grid is organized as up to 3-dim *array of thread blocks*
- 🔑 each block is organized into up to 3-dim *array of threads*
- 🔑 all blocks have the same number of threads
- 🔑 organized in the same manner.



## Block of threads:

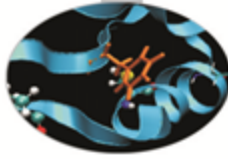
set of concurrently executing threads that can *cooperate* among themselves through

- 🔑 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`)

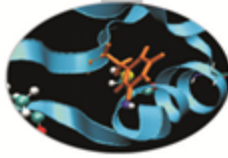
The exact organization of a grid is determined by the execution configuration provided at kernel launch.

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

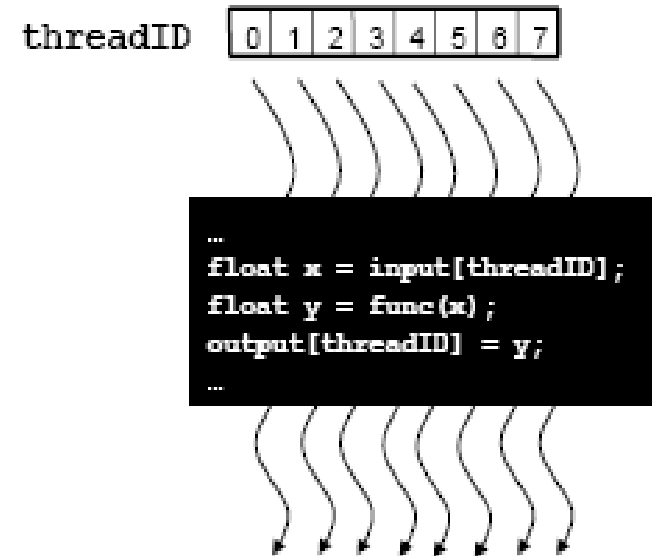
- `gridDim` → dimensions of the grid in terms of number of blocks

- `blockDim` → dimensions of the block in terms of number of threads

# Thread ID computation



The built-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;
```

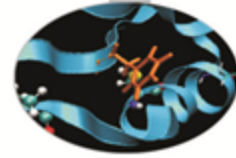
## 2D:

```
int iy = blockDim.y * blockIdx.y + threadIdx.y;
```

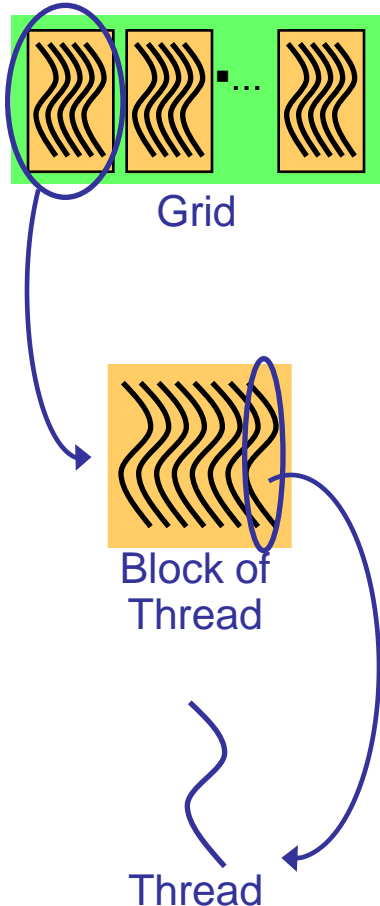
```
int ix = blockDim.x * blockIdx.x + threadIdx.x;
```

```
int id = iy * dimx + ix;
```

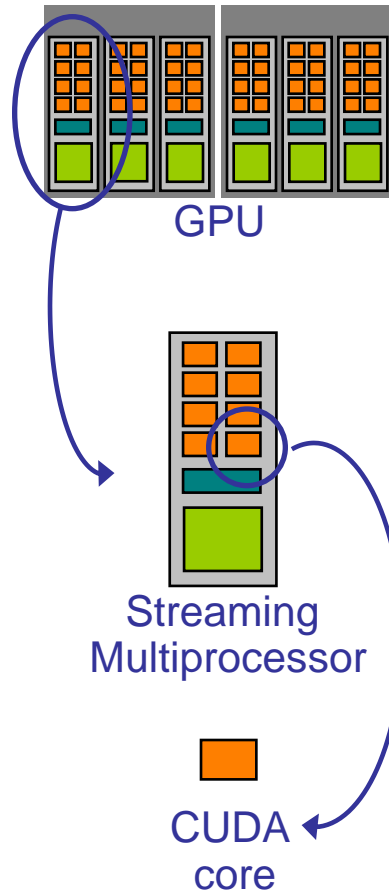
# Threads execution model



## Software



## Hardware

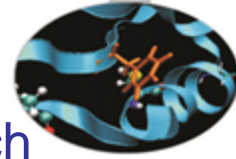


CUDA's hierarchy of threads/memories maps to the hierarchy of processors on the GPU:

- 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.

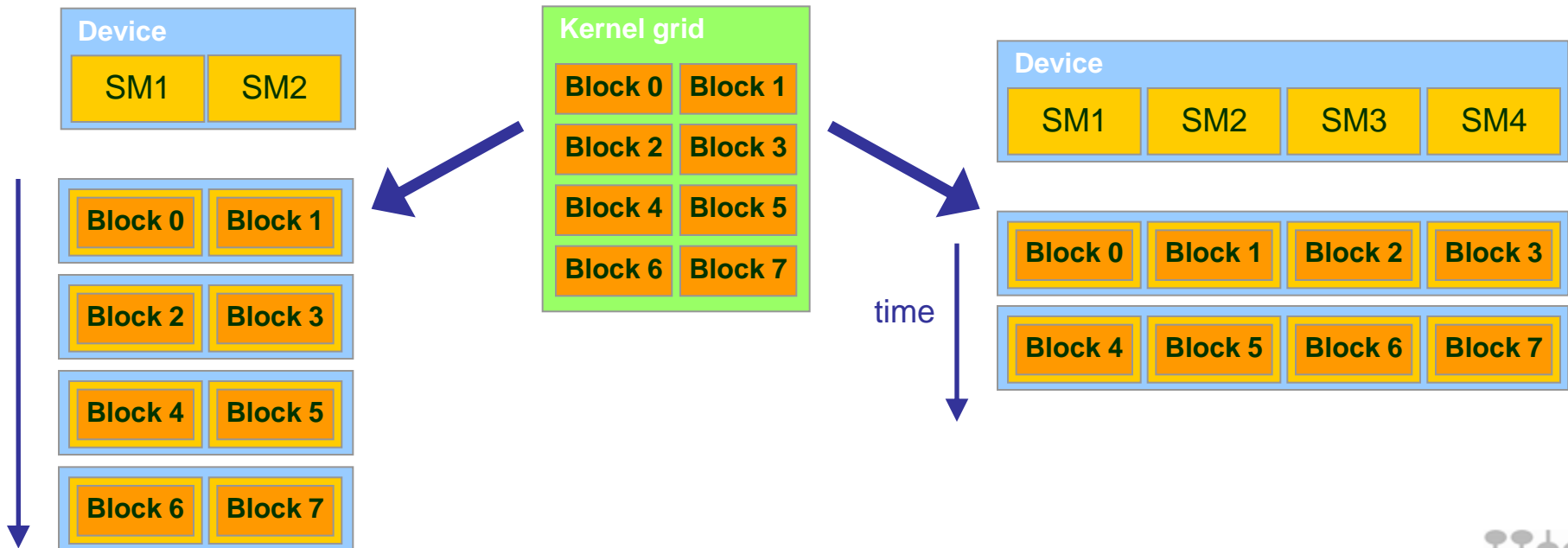
A maximum number of blocks can be assigned to each SM (8 for Fermi, 16 for Kepler)  
The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete the execution of blocks previously assigned to them.

# Transparent scalability

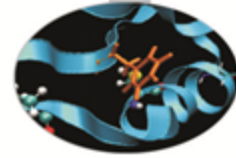


By not allowing threads in different blocks to synchronize with each other, CUDA runtime system can execute blocks in any order relative to each other.

This flexibility enables to execute the same application code on hardware with different numbers of SM (*transparent scalability*).



# Launching a kernel



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

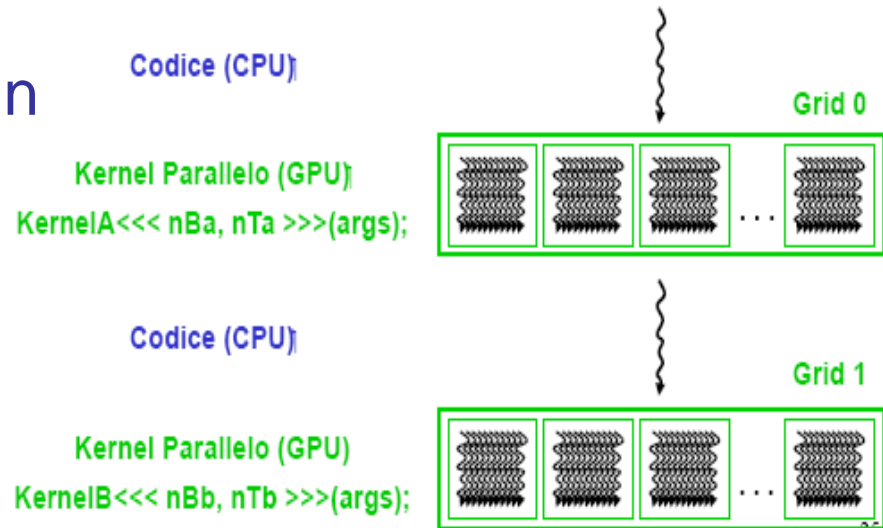
```

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

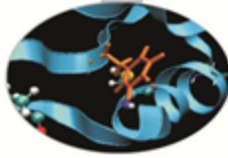
//call the kernel
KernelFunc<<< gridDim, blockDim >>>(<arguments>);
  
```

Typical CUDA grids contain thousands to millions of threads.

All kernel calls are asynchronous!



# Kernel example



## CPU code:

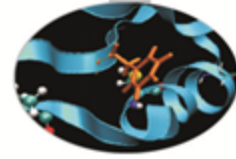
```
void increment_cpu(float* a, float b, int n){
    for (idx=0; idx<n; ++idx)
        a[idx]+=b;
}
int main(void){
    //...
    increment_cpu(h_a,h_b,16);
}
```

## GPU code:

```
__global__ increment_gpu(float* a, float b, int n){
    int idx = threadIdx.x + blockIdx.x*blockDim.x;
    if (idx < n)
        a[idx]+=b;
}
int main(void){
    //...
    increment_gpu<<<blocks,threads>>>(d_a,d_b,16);
}
```



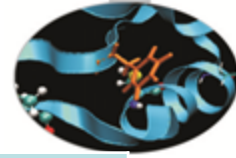
# CUDA Function modifiers



CUDA extends C function declarations with three qualifier keywords.

Function declaration	Executed on the	Only callable from the
<code>__device__</code> ( <i>device functions</i> )	device	device
<code>__global__</code> ( <i>kernel function</i> )	device	host
<code>__host__</code> ( <i>host functions</i> )	host	host

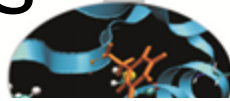
# CUDA variable qualifiers



Variable declaration	memory	lifetime	scope
Automatic scalar variables	register	kernel	thread
<code>__shared__</code>	shared	kernel	block
<code>__device__</code>	global	application	grid
<code>__constant__</code>	constant	application	grid

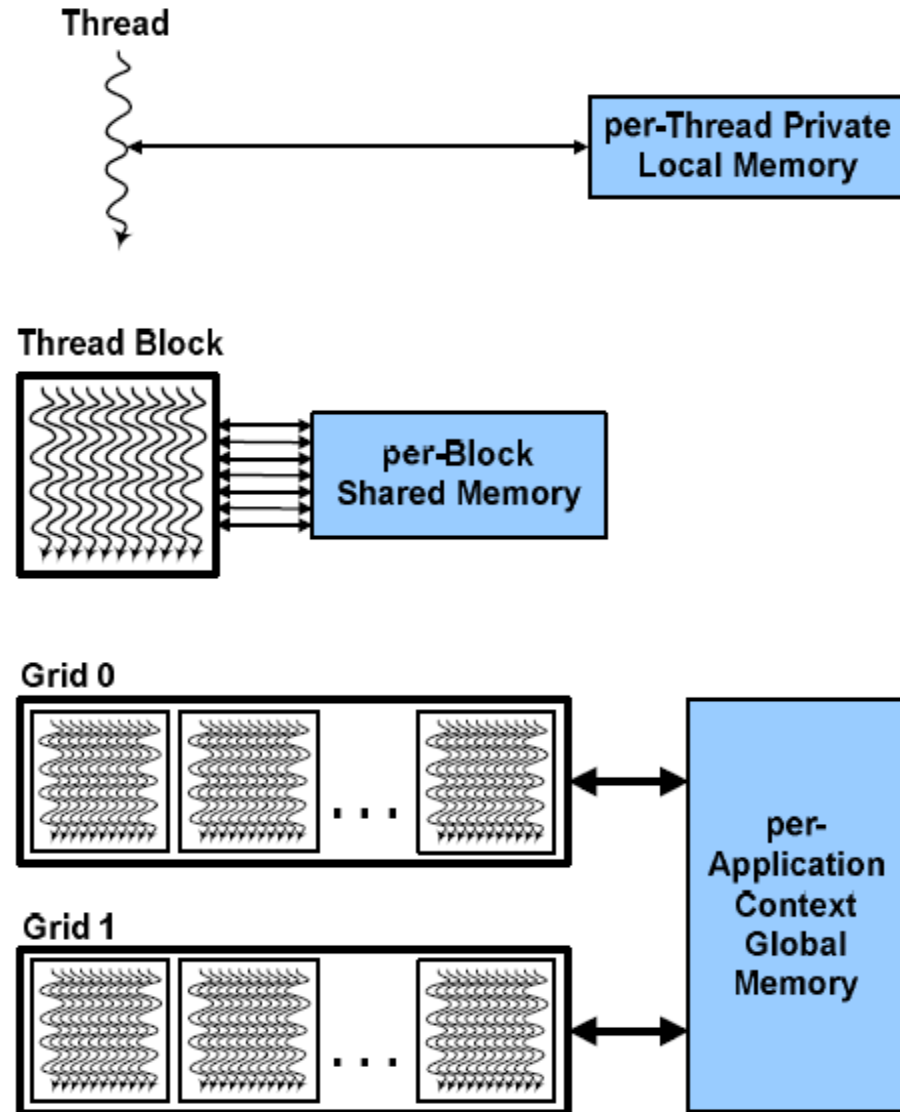
- † Local memory is used in case of register spilling (`--pextras-options=-v` flag)
- † Global variables are often used to pass information from one kernel to another.
- † Constant variables are often used for providing input values to kernel functions.

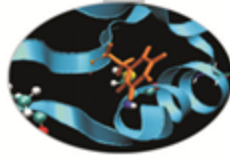
# Hierarchy of device memories



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**



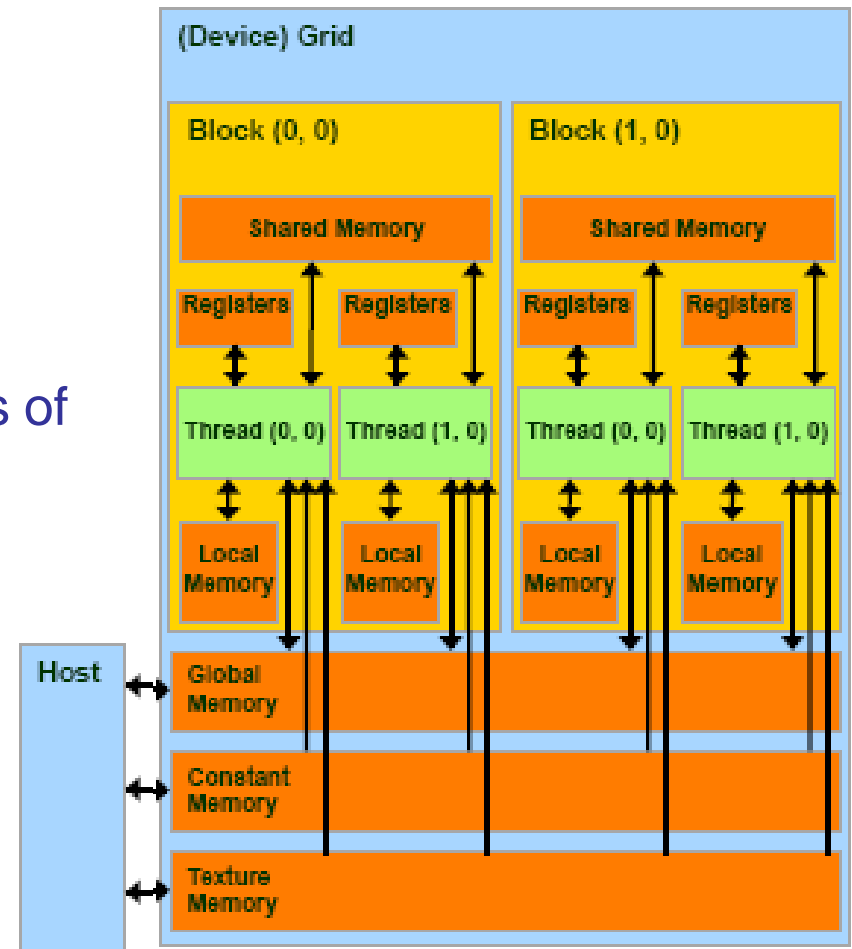


## on-chip memories:

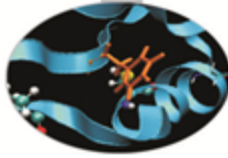
- registers (~8KB) → SP
- shared memory (~16KB) → SM
- they can be accessed at very high speed in a highly parallel manner.

## per-grid memories:

- global memory (~4GB)
  - long access latencies (hundreds of clock cycles)
  - finite access bandwidth
- constant memory (~64KB)
  - read only
  - short-latency (cached) and high bandwidth 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.



## Static modality

inside the kernel:

```
__shared__ float f[100];
```

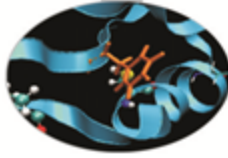
## Dynamic modality

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[ ];
```



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

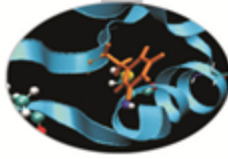
**cudaMalloc**(void\*\* bufferPtr, size\_t n)

- ‡ 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 should be cast to (void\*\*)!
- ‡ The second parameter is the size of the buffer to be allocated, in terms of bytes

**cudaFree**(void\* bufferPtr)

- ‡ It frees the storage space of the object





```
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 of the *int value* converted to unsigned char.

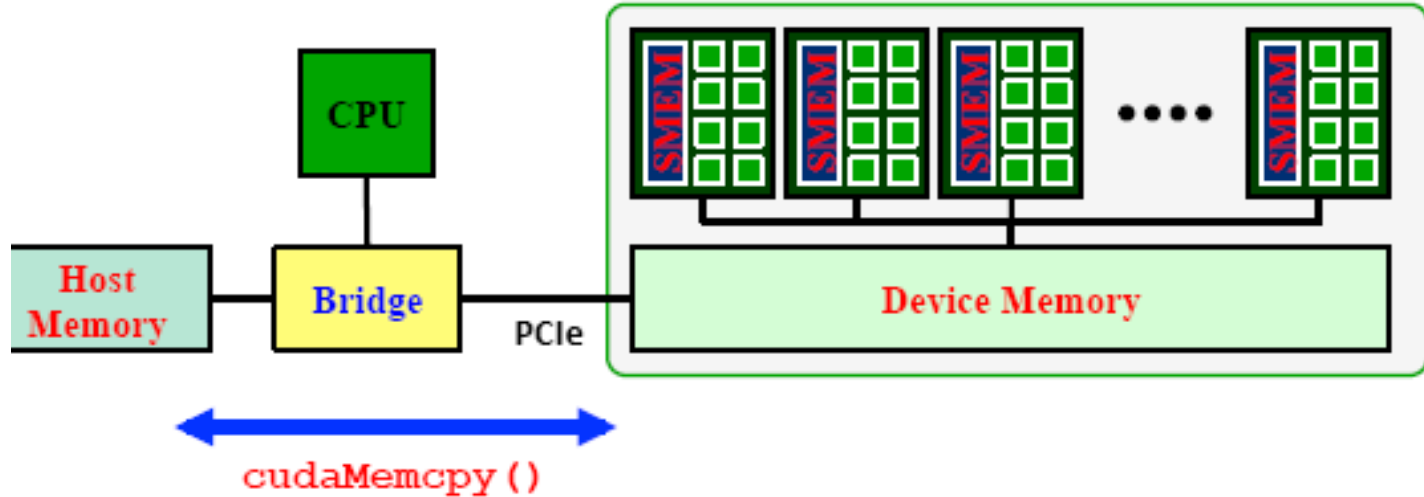
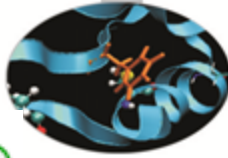
CUDA version of the C `memset()` function.

*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



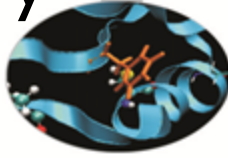
API **blocking** functions for data transfer between memories:

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

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

↙
↑
↖
↖

Destination    source data    number of bytes    symbolic constant indicating the direction



```
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.

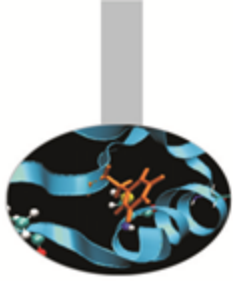
*src* - source memory address

*count* - size in bytes to copy

*offset* - offset from start of symbol in bytes

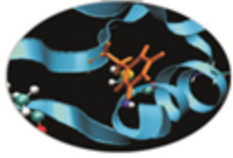
*kind* - type of transfer, it can be either

cudaMemcpyHostToDevice or  
cudaMemcpyDeviceToDevice



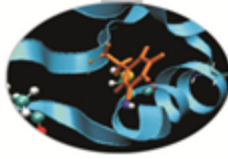
# Device management

- 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);
```



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()`

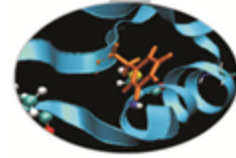
Then the following function returns the code of the last error:

```
cudaError_t cudaGetLastError()
```

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

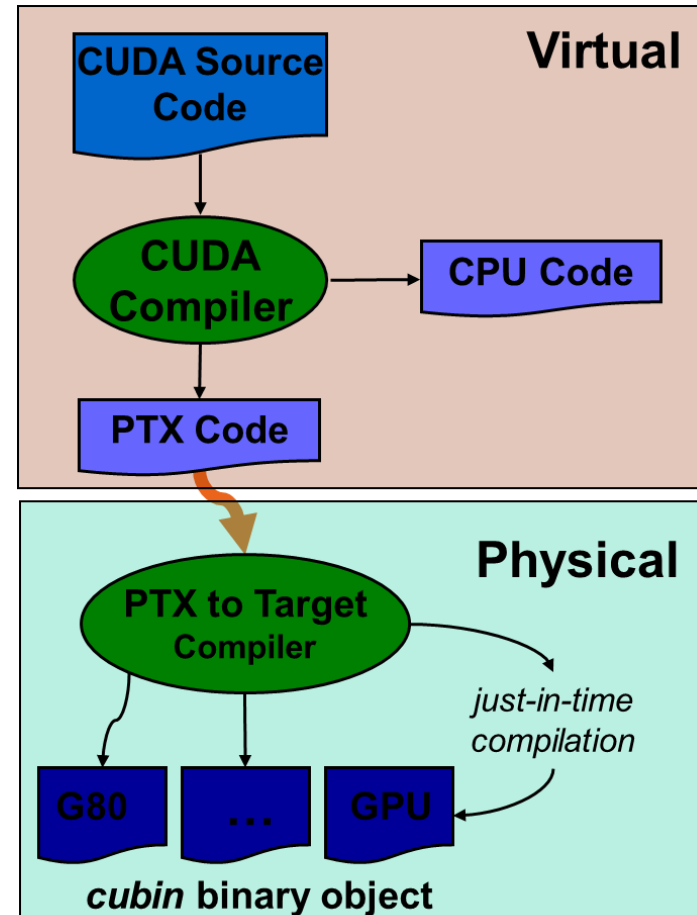


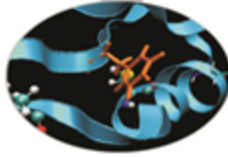
# NVIDIA C compiler



**nvcc** front-end for compilation:

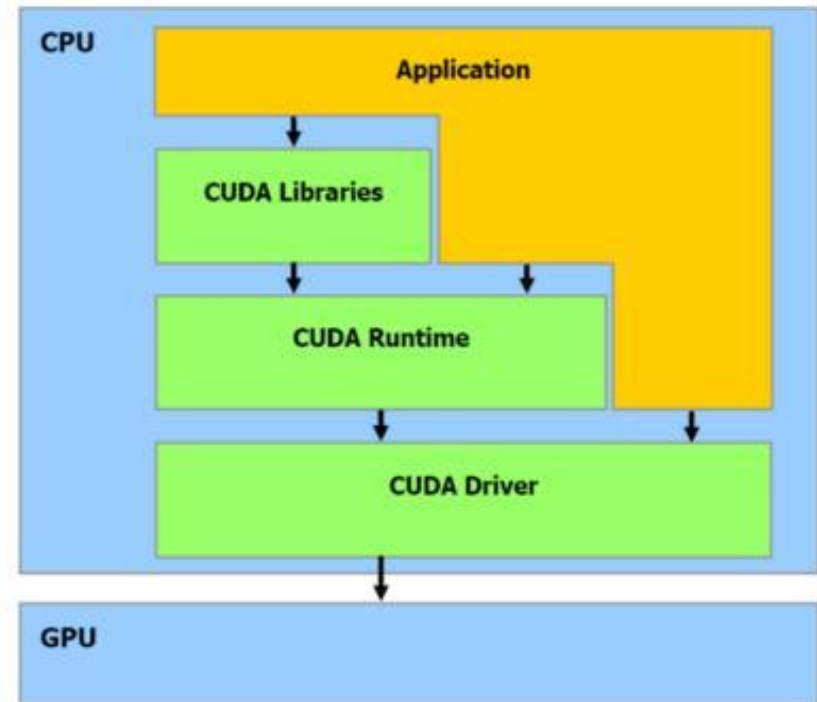
- ✦ separates GPU code from CPU code
- ✦ CPU code -> C/C++ compiler (Microsoft Visual C/C++, GCC, ecc.)
- ✦ GPU code is converted in an intermediate assembly language: PTX, then in binary form (the *cubin* object)
- ✦ link all executables

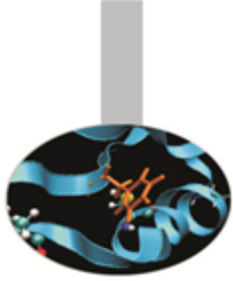




# CUDA Driver Vs Runtime API

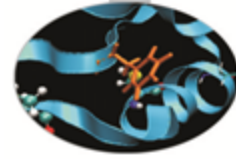
- † CUDA is composed of two APIs:
  - ‡ the CUDA runtime API
  - ‡ the CUDA driver API
- † They are mutually exclusive
- † Runtime API:
  - ‡ easier to program
  - ‡ it eases device code management: it's where the C-for-CUDA language lives
- † Driver API:
  - ‡ requires more code: no syntax sugar for the kernel launch, for example
  - ‡ finer control over the device especially 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**.
- † 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.
- † Kernels are launched using API entry points.



# Vector add: driver Vs runtime API

## // driver API

### // initialize CUDA

```
err = cuInit(0);  
err = cuDeviceGet(&device, 0);  
err = cuCtxCreate(&context, 0, device);
```

### // setup device memory

```
err = cuMemAlloc(&d_a, sizeof(int) * N);  
err = cuMemAlloc(&d_b, sizeof(int) * N);  
err = cuMemAlloc(&d_c, sizeof(int) * N);
```

### // copy arrays to device

```
err = cuMemcpyHtoD(d_a, a, sizeof(int) * N);  
err = cuMemcpyHtoD(d_b, b, sizeof(int) * N);
```

### // prepare kernel launch

```
kernelArgs[0] = &d_a;  
kernelArgs[1] = &d_b;  
kernelArgs[2] = &d_c;
```

### // load device code

```
err = cuModuleLoad(&module, module_file);  
err = cuModuleGetFunction(&function, module, kernel_name);
```

### // execute the kernel over the <N,1> grid

```
err = cuLaunchKernel(function, N, 1, 1, // Nx1x1 blocks  
                    1, 1, 1, // 1x1x1 threads  
                    0, 0, kernelArgs, 0);
```

## // runtime API

### // setup device memory

```
err = cudaMalloc((void**)&d_a, sizeof(int) * N);  
err = cudaMalloc((void**)&d_b, sizeof(int) * N);  
err = cudaMalloc((void**)&d_c, sizeof(int) * N);
```

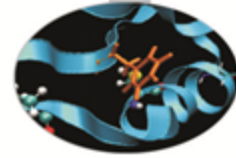
### // copy arrays to device

```
err=cudaMemcpy(d_a, a, sizeof(int) * N, cudaMemcpyHostToDevice);  
err=cudaMemcpy(d_b, b, sizeof(int) * N, cudaMemcpyHostToDevice);
```

### // launch kernel over the <N, 1> grid

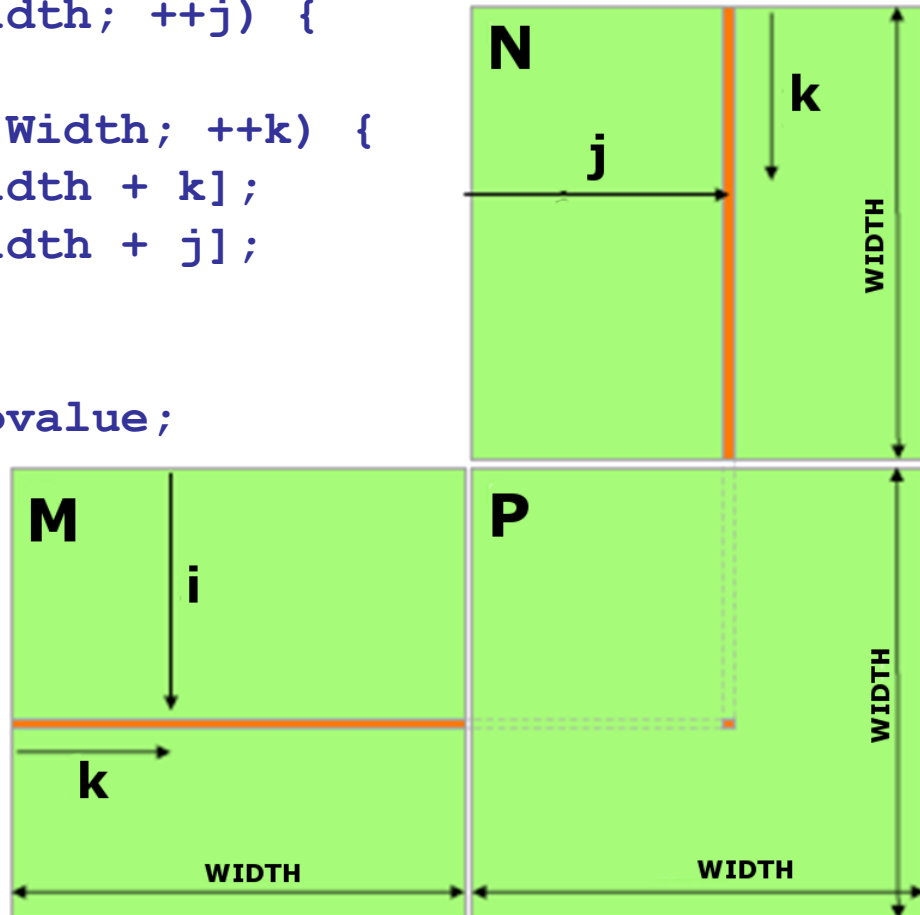
```
matSum<<<N,1>>>(d_a, d_b, d_c); // yum, syntax sugar!
```

# Matrix-Matrix multiplication example



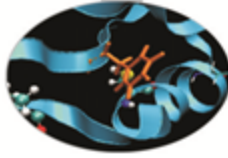
```
void MatrixMulOnHost(float* M, float* N, float* P,
                    int Width) {
    for (int i = 0; i < Width; ++i) {
        for (int j = 0; j < Width; ++j) {
            float pvalue = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * Width + k];
            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

$$P = M * N$$



CUDA parallelization: each thread computes an element of P

# Matrix-Matrix multiplication *device code*

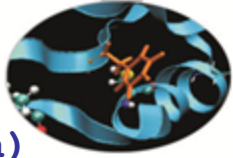


```
__global__ void MNKernel(float* Md, float *Nd, float *Pd, int
width)
{
    // 2D thread ID
    int col = threadIdx.x;
    int row = threadIdx.y;

    // 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];

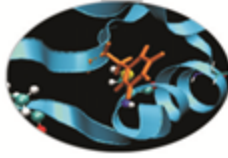
    // write the matrix to device memory
    // (each thread writes one element)
    Pd[row * width + col] = Pvalue;
}
```

# 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;
    // allocate M, N and P on the device
    cudaMalloc((void**)&Md, size);
    cudaMalloc((void**)&Nd, size);
    cudaMalloc((void**)&Pd, size);
    // transfer M and N to the device memory
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
    // 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 example



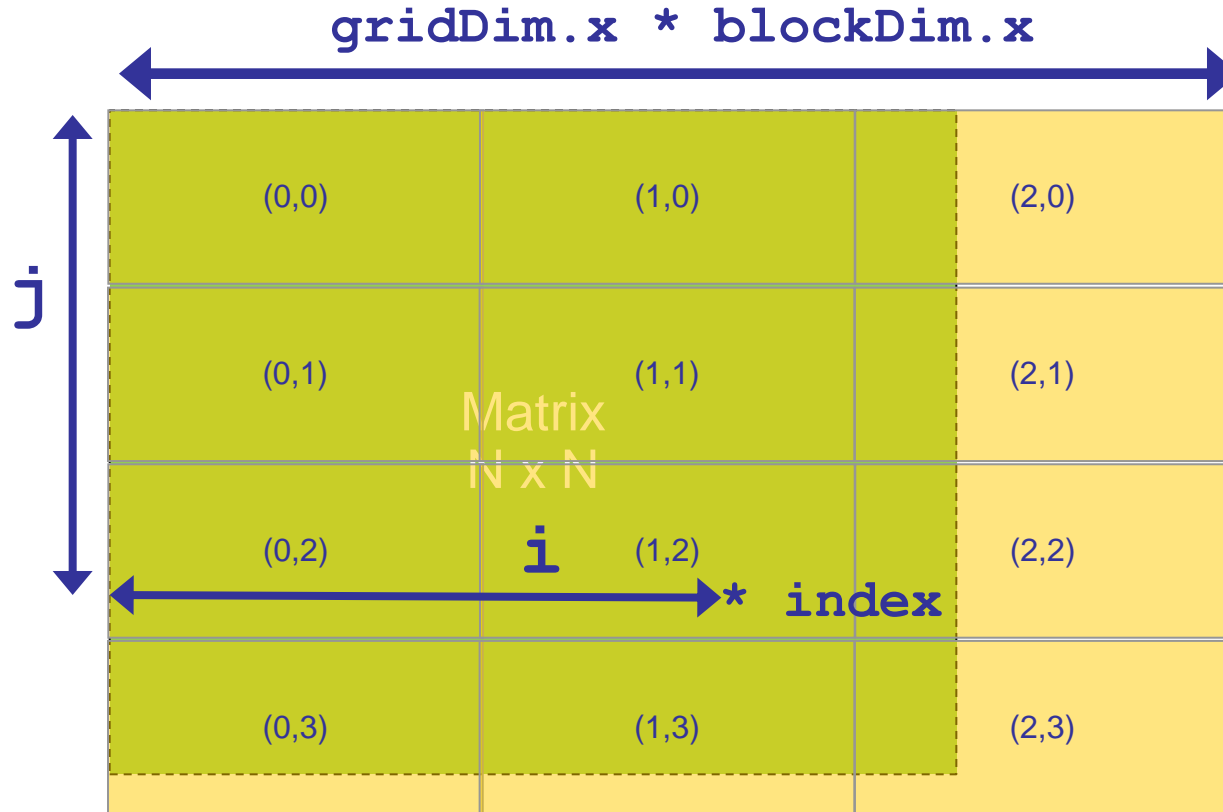
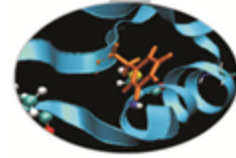
Limitation: a block can have up to 1024 threads (for Fermi and Kepler). Therefore the previous implementation can compute square matrices of order less or equal to 32.

## 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



```

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

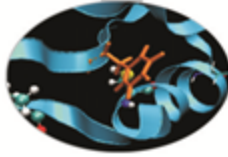
```

```

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

```

# Matrix-Matrix multiplication example



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

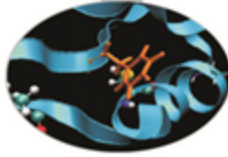
    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;
    for (int k=0; k < width; k++)
        Pvalue += Md[row * width + k] * Nd[k * width + col];

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

## Kernel invocation:

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

# Matrix-Matrix multiplication example



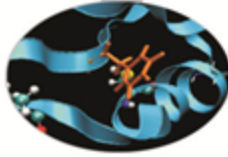
Which is the optimal dimension of the block (i.e. TILE\_WIDTH)?

Knowing that each SM of a Fermi can have up to 1536 threads, we have

- $8 \times 8 = 64$  threads  $\Rightarrow 1536/64 = 24$  blocks to fully occupy an SM; but we are limited to 8 blocks in each SM therefore we will end up with only  $64 \times 8 = 512$  threads in each SM.
- $16 \times 16 = 256$  threads  $\Rightarrow 1536/256 = 6$  blocks we will have full thread capacity in each SM.
- $32 \times 32 = 1024$  threads  $\Rightarrow 1536/1024 = 1.5 \Rightarrow 1$  block.

 **TILE\_WIDTH = 16**

# Matrix-Matrix multiplication example

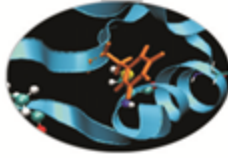


Which is the optimal dimension of the block (i.e. TILE\_WIDTH)?

Knowing that each SM of a Kepler can have up to 2048 threads, we have

- $8 \times 8 = 64$  threads  $\Rightarrow 2048/64 = 32$  blocks to fully occupy an SM; but we are limited to 16 blocks in each SM therefore we will end up with only  $64 \times 16 = 1024$  threads in each SM.
- $16 \times 16 = 256$  threads  $\Rightarrow 2048/256 = 8$  blocks we will have full thread capacity in each SM.
- $32 \times 32 = 1024$  threads  $\Rightarrow 2048/1024 = 2$  blocks.

 **TILE\_WIDTH = 16 or 32**

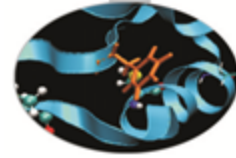


Although having many threads available for execution can theoretically tolerate long memory access latency, one can easily run into a situation where traffic congestion prevents all but few threads from making progress, thus making some SM idle!

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.*

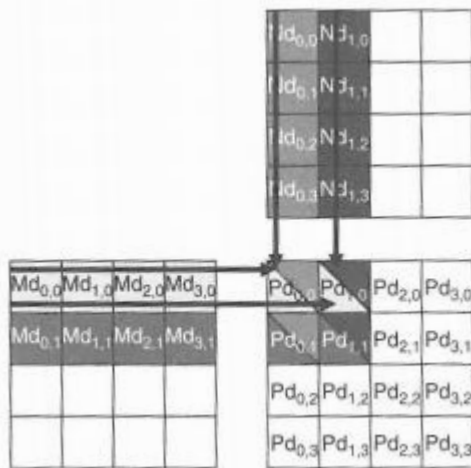
# Matrix-Matrix multiplication example



In the previous kernel:

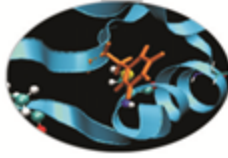
*thread(x,y) of block(0,0) access the elements of **Md** row  $x$  and **Nd** column  $y$  from the global memory.*

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



	Pd <sub>0,0</sub> Thread(0,0)	Pd <sub>1,0</sub> Thread(1,0)	Pd <sub>0,1</sub> Thread(0,1)	Pd <sub>1,1</sub> Thread(1,1)
	Md <sub>0,0</sub> * Nd <sub>0,0</sub>	Md <sub>0,0</sub> * Nd <sub>1,0</sub>	Md <sub>0,1</sub> * Nd <sub>0,0</sub>	Md <sub>0,1</sub> * Nd <sub>1,0</sub>
	Md <sub>1,0</sub> * Nd <sub>0,1</sub>	Md <sub>1,0</sub> * Nd <sub>1,1</sub>	Md <sub>1,1</sub> * Nd <sub>0,1</sub>	Md <sub>1,1</sub> * Nd <sub>1,1</sub>
	Md <sub>2,0</sub> * Nd <sub>0,2</sub>	Md <sub>2,0</sub> * Nd <sub>1,2</sub>	Md <sub>2,1</sub> * Nd <sub>0,2</sub>	Md <sub>2,1</sub> * Nd <sub>1,2</sub>
	Md <sub>3,0</sub> * Nd <sub>0,3</sub>	Md <sub>3,0</sub> * Nd <sub>1,3</sub>	Md <sub>3,1</sub> * Nd <sub>0,3</sub>	Md <sub>3,1</sub> * Nd <sub>1,3</sub>

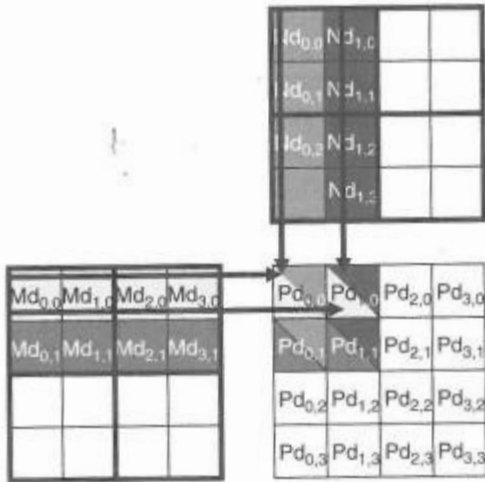
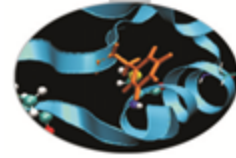
Access order ↓



*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  $N$ , using  $N \times N$  blocks!

### 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.

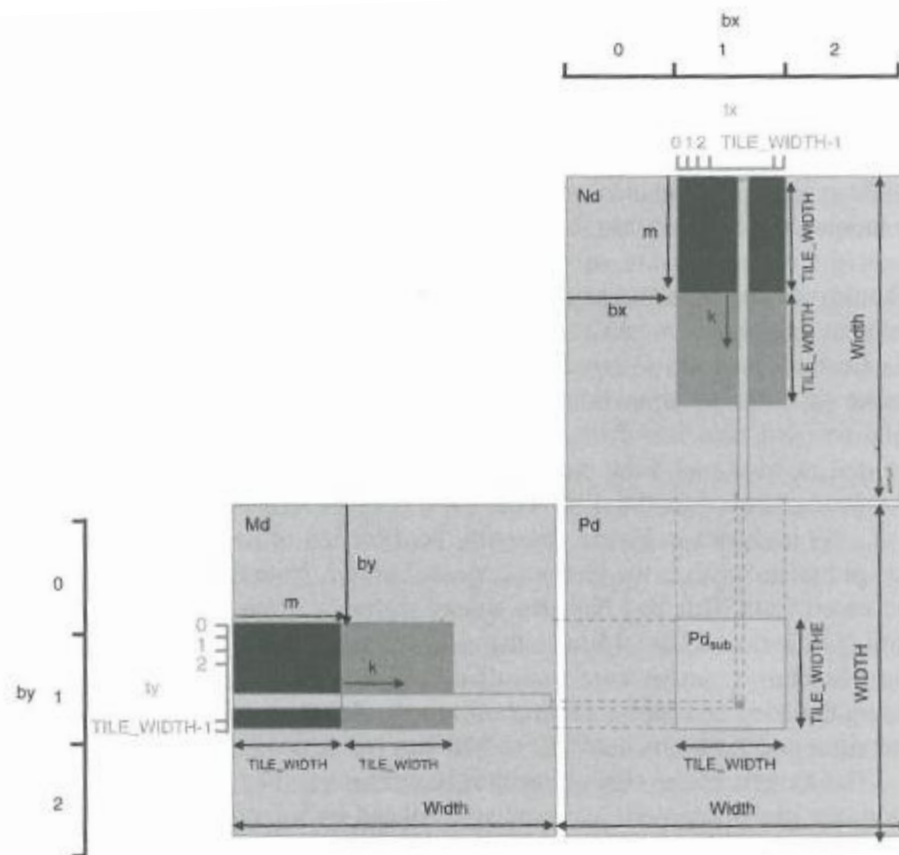
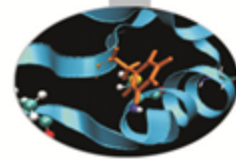


	Phase 1			Phase 2		
$T_{0,0}$	$Md_{0,0}$ ↓ $Mds_{0,0}$	$Nd_{0,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{1,0} * Nds_{0,1}$	$Md_{2,0}$ ↓ $Mds_{0,0}$	$Nd_{0,2}$ ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{1,0} * Nds_{0,1}$
$T_{1,0}$	$Md_{1,0}$ ↓ $Mds_{1,0}$	$Nd_{1,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{0,0} * Nds_{1,0} +$ $Mds_{1,0} * Nds_{1,1}$	$Md_{3,0}$ ↓ $Mds_{1,0}$	$Nd_{1,2}$ ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{0,0} * Nds_{1,0} +$ $Mds_{1,0} * Nds_{1,1}$
$T_{0,1}$	$Md_{0,1}$ ↓ $Mds_{0,1}$	$Nd_{0,1}$ ↓ $Nds_{0,1}$	$PdValue_{0,1} +=$ $Mds_{0,1} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{0,1}$	$Md_{2,1}$ ↓ $Mds_{0,1}$	$Nd_{0,3}$ ↓ $Nds_{0,1}$	$PdValue_{0,1} +=$ $Mds_{0,1} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{0,1}$
$T_{1,1}$	$Md_{1,1}$ ↓ $Mds_{1,1}$	$Nd_{1,1}$ ↓ $Nds_{1,1}$	$PdValue_{1,1} +=$ $Mds_{0,1} * Nds_{1,0} +$ $Mds_{1,1} * Nds_{1,1}$	$Md_{3,1}$ ↓ $Mds_{1,1}$	$Nd_{1,3}$ ↓ $Nds_{1,1}$	$PdValue_{1,1} +=$ $Mds_{0,1} * Nds_{1,0} +$ $Mds_{1,1} * Nds_{1,1}$

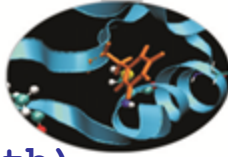
time →

- *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**.*





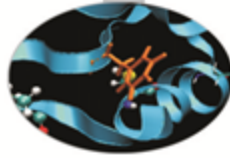
# Matrix-Matrix multiplication example



```
__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*BlockDim.x + tx;
    int row = blockIdx.y*BlockDim.y + ty;
    float Pvalue = 0;
    // Loop over the Md and Nd tiles required to compute the Pd element
    // m is the number of phases
    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:

$TILE\_WIDTH = 16 \implies$  each block requires  $16 \times 16 \times 4 = 1\text{KB}$  of storage for **Mds**  
+ 1KB for **Nds**  
 $\implies$  2KB of shared memory per block

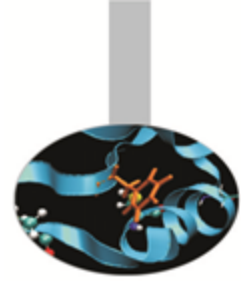
The 48KB shared memory allows 24 blocks to simultaneously reside in an SM. **OK!**

But the maximum number of threads per SM is 1536 (for Fermi)

 only  $1536/256 = 8$  blocks are allowed in each SM  
only  $8 \times 2\text{KB} = 16\text{KB}$  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 the *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)

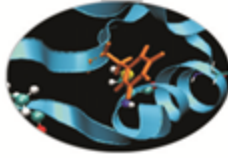


Threads often execute *long-latency operations*:

- 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**).*

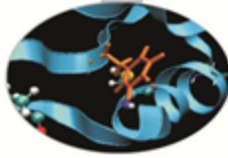
# Control flow



- 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.

**Try to avoid *warp divergence***

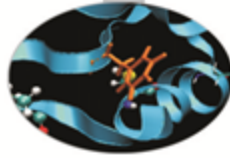
# Multi-GPUs: P2P



```
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_0, gpuid_1);  
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_1, gpuid_0);  
  
cudaSetDevice(gpuid_0);  
cudaDeviceEnablePeerAccess(gpuid_1, 0);  
cudaSetDevice(gpuid_1);  
cudaDeviceEnablePeerAccess(gpuid_0, 0);  
  
cudaMemcpy(gpu0_buf, gpu1_buf, buf_size, cudaMemcpyDefault);
```

- `cudaMemcpy()` knows that our buffers are on different devices (UVA), will do a P2P copy now
- Note that this will *transparently* fall back to a normal copy through the host if P2P is not available

# Multi-GPUs: direct access



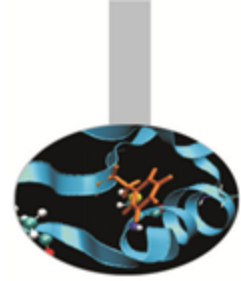
```
__global__ void SimpleKernel(float *src, float *dst)
{
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    dst[idx] = src[idx];
}
```

```
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_0, gpuid_1);
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_1, gpuid_0);

cudaSetDevice(gpuid_0);
cudaDeviceEnablePeerAccess(gpuid_1, 0);
cudaSetDevice(gpuid_1);
cudaDeviceEnablePeerAccess(gpuid_0, 0);

cudaSetDevice(gpuid_0);
SimpleKernel<<<blocks, threads>>> (gpu0_buf, gpu1_buf);
SimpleKernel<<<blocks, threads>>> (gpu1_buf, gpu0_buf);
cudaSetDevice(gpuid_1);
SimpleKernel<<<blocks, threads>>> (gpu0_buf, gpu1_buf);
SimpleKernel<<<blocks, threads>>> (gpu1_buf, gpu0_buf);
```

- After P2P initialization, this kernel can now read and write data in the memory of multiple GPUs (just *deferencing pointers!*)

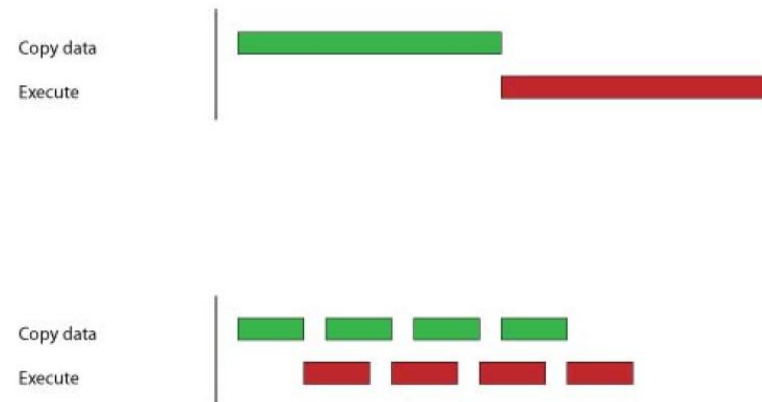


# Asynchronous operations

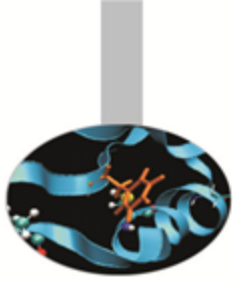
- Kernel calls are asynchronous by default
- Memory transfers and copybacks *are blocking*
- The `cudaMemcpy` has an asynchronous version (`cudaMemcpyAsync`)
- Boards  $\leq 1.3$  can overlap *copy-copy (opposite directions)* and *copy-kernel*
- Boards  $\geq 2.0$  (Fermi and Kepler) can overlap *kernel-kernel* execution.

```

// First transfer
cudaMemcpyAsync(inputDevPtr, hostPtr, size, cudaMemcpyHostToDevice, 0);
// First invocation
MyKernel<<<100, 512, 0, 0>>> (outputDevPtr, inputDevPtr, size);
// Second transfer
cudaMemcpyAsync(inputDevPtr2, hostPtr2, size, cudaMemcpyHostToDevice, 0);
// Second invocation
MyKernel2<<<100, 512, 0, 0>>> (outputDevPtr2, inputDevPtr2, size);
// Wrapup
cudaMemcpyAsync(hostPtr, outputDevPtr, size, cudaMemcpyDeviceToHost, 0);
cudaMemcpyAsync(hostPtr2, outputDevPtr2, size, cudaMemcpyDeviceToHost, 0);
cudaThreadSynchronize();
  
```

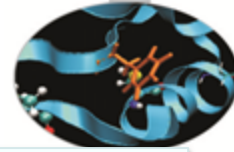






# Streams

- ⌚ A *stream* is a FIFO command queue;
- ⌚ a stream is independent to every other active stream;
- ⌚ CUDA streams are the main way to exploit concurrent execution and I/O operations.



# CUDA Streams

```

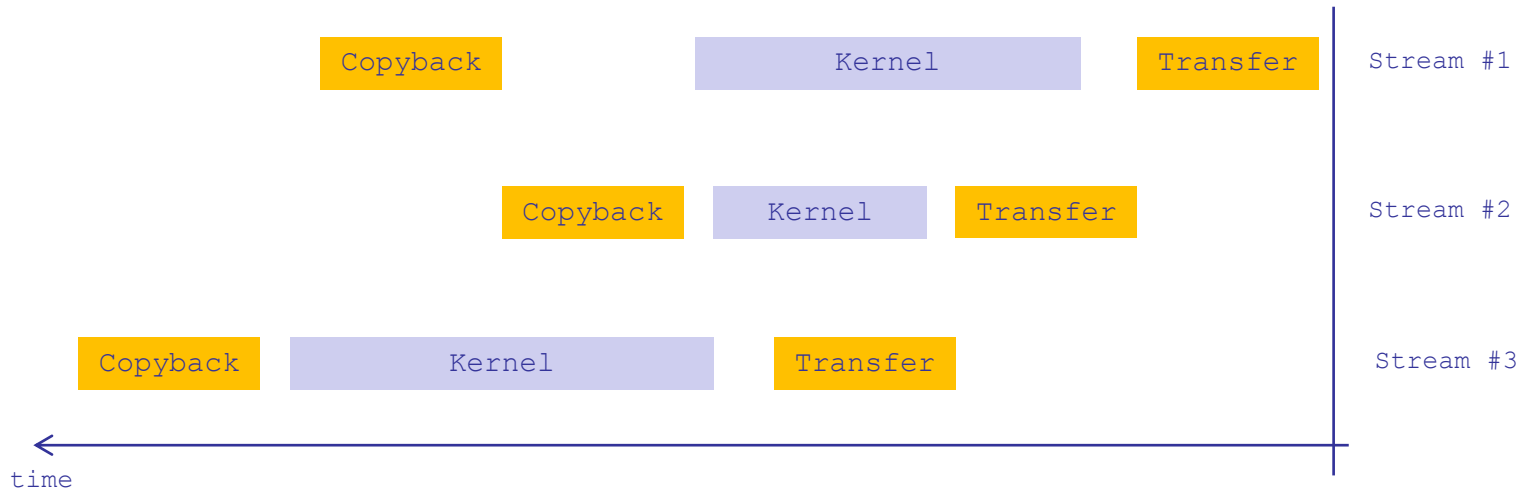
cudaStream_t stream[3];
for (int i=0; i<3; ++i) cudaStreamCreate(&stream[i]);

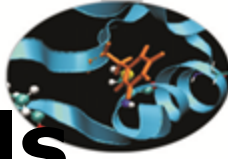
float* hostPtr;
cudaMallocHost((void**)&hostPtr, 3 * size);

for (int i=0; i<3; ++i) cudaMemcpyAsync(inputDevPtr+i*size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
for (int i=0; i<3; ++i) myComputeKernel<<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i=0; i<3; ++i) cudaMemcpyAsync(hostPtr + i * size, outputDevPtr+i*size, size, cudaMemcpyDeviceToHost, stream[i]);

cudaThreadSynchronize();

for (int i=0; i<3; ++i) cudaStreamDestroy(&stream[i]);
  
```

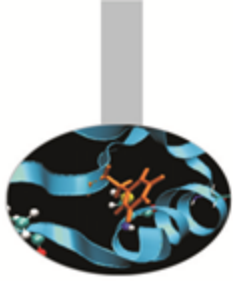




# Streams: how to overlap kernels

Starting from capability 2.0 the board has the ability to overlap computations from multiple kernels where:

- no synchronization happens between command stages;
- no operations occur on the default stream;
- the active streams are less than 16;



# Timing

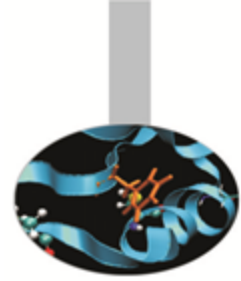
- It's allowed to use std timing facilities (host side).
- Beware of asynchronous calls!

```
start = clock();  
my_kernel<<< g, b, s >>>();  
cudaThreadSynchronize();  
end = clock();
```

- CUDA provides the *Events* facility.
- Needed to time single streams without loosing concurrency.
- Needed to create events by using `cudaEventCreate`

```
cudaEventRecord(start, 0);  
for (int i = 0; i < 2; ++i) {  
    cudaMemcpyAsync(inputDev + i * size, inputHost + i * size,  
                    size, cudaMemcpyHostToDevice, stream[i]);  
    MyKernel<<<100, 512, 0, stream[i]>>>  
        (outputDev + i * size, inputDev + i * size, size);  
    cudaMemcpyAsync(outputHost + i * size, outputDev + i * size,  
                    size, cudaMemcpyDeviceToHost, stream[i]);  
}  
cudaEventRecord(stop, 0);  
cudaEventSynchronize(stop);  
float elapsedTime;  
cudaEventElapsedTime(&elapsedTime, start, stop);
```



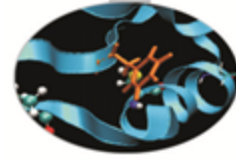


# Page-locked memory

- Pinned (or page-locked memory) is a main memory area that is not pageable by the operating system;
- ensures faster transfers (the DMA engine can work without raising interrupts);
- the only way to get closer to PCI peak bandwidth;
- allows CUDA asynchronous operations to work correctly.

```
// allocate page-locked memory
cudaMallocHost(&area, sizeof(double) * N);
// free page-locked memory
cudaFreeHost(area);
```

```
// allocate regular memory
area = (double*) malloc( sizeof(double) * N );
// lock area pages (CUDA >= 4.0)
cudaHostRegister( area, sizeof(double) * N, cudaHostRegisterPortable );
// unlock area pages (CUDA >= 4.0)
cudaHostUnregister(area);
// free regular memory
cudaFreeHost(area);
```



# Kepler: dynamic parallelism

- One of the biggest CUDA limitations is the need to fit a single grid configuration for the whole kernel.

If you need to reshape the grid, you have to resync back to host and split your code.

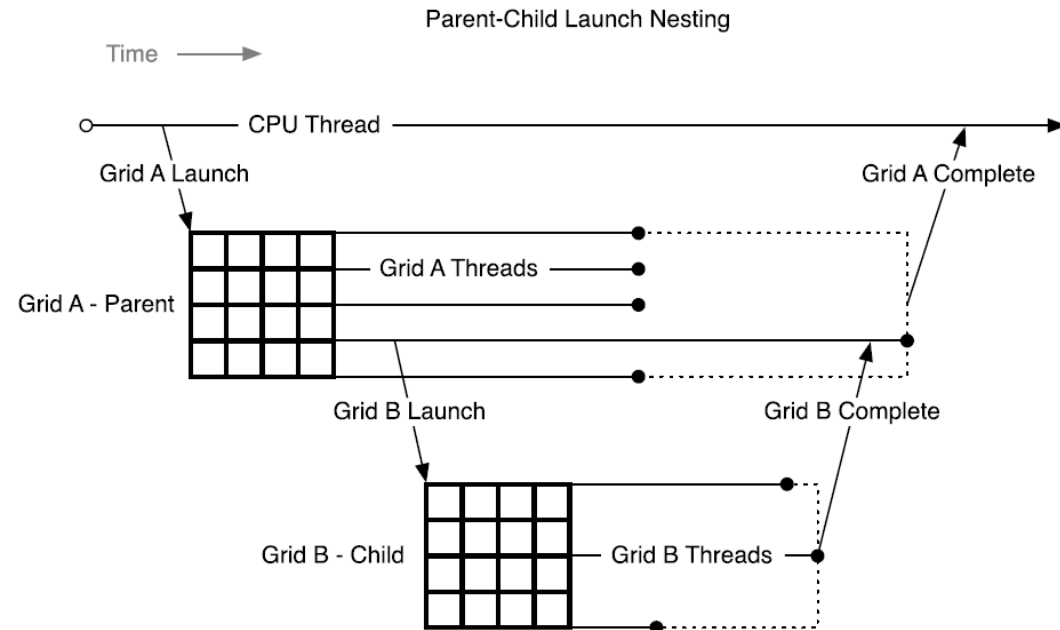
- Kepler (in addition to CUDA 5.x) introduced *Dynamic Parallelism*
- It enables a global kernel to be called from within another kernel
- The child grid can be *dynamically sized and optionally synchronized*

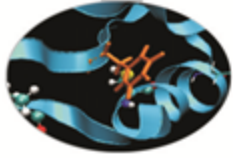
```

__global__ ChildKernel(void* data){
    //Operate on data
}

__global__ ParentKernel(void *data){
    ChildKernel<<<16, 1>>>(data);
}

// In Host Code:
ParentKernel<<<256, 64>>>(data);
    
```





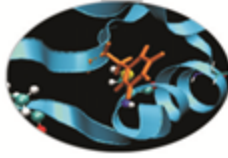
# Exploiting Multi-GPUs with OpenMP

CUDA  $\geq$  4.0 introduced the N-to-N bound feature:

1. Every thread can be bound to any board
2. Every board can be bound to an arbitrary number of threads

Multi-GPU can be exploited through your favourite multi-threading paradigm (OpenMP, pthreads, etc...)

```
#pragma omp parallel
#pragma omp sections
{
  #pragma omp section
  {
    cudaSetDevice(0);
    cudaMemcpy(device_data_1, host_data_1, size, cudaMemcpyHostToDevice);
    my_kernel<<< grid, block >>>(device_data_1);
    // ...
  }
  #pragma omp section
  {
    cudaSetDevice(1);
    cudaMemcpy(device_data_2, host_data_2, size, cudaMemcpyHostToDevice);
    my_kernel<<< grid, block >>>(device_data_2);
    // ...
  }
}
```



# 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