

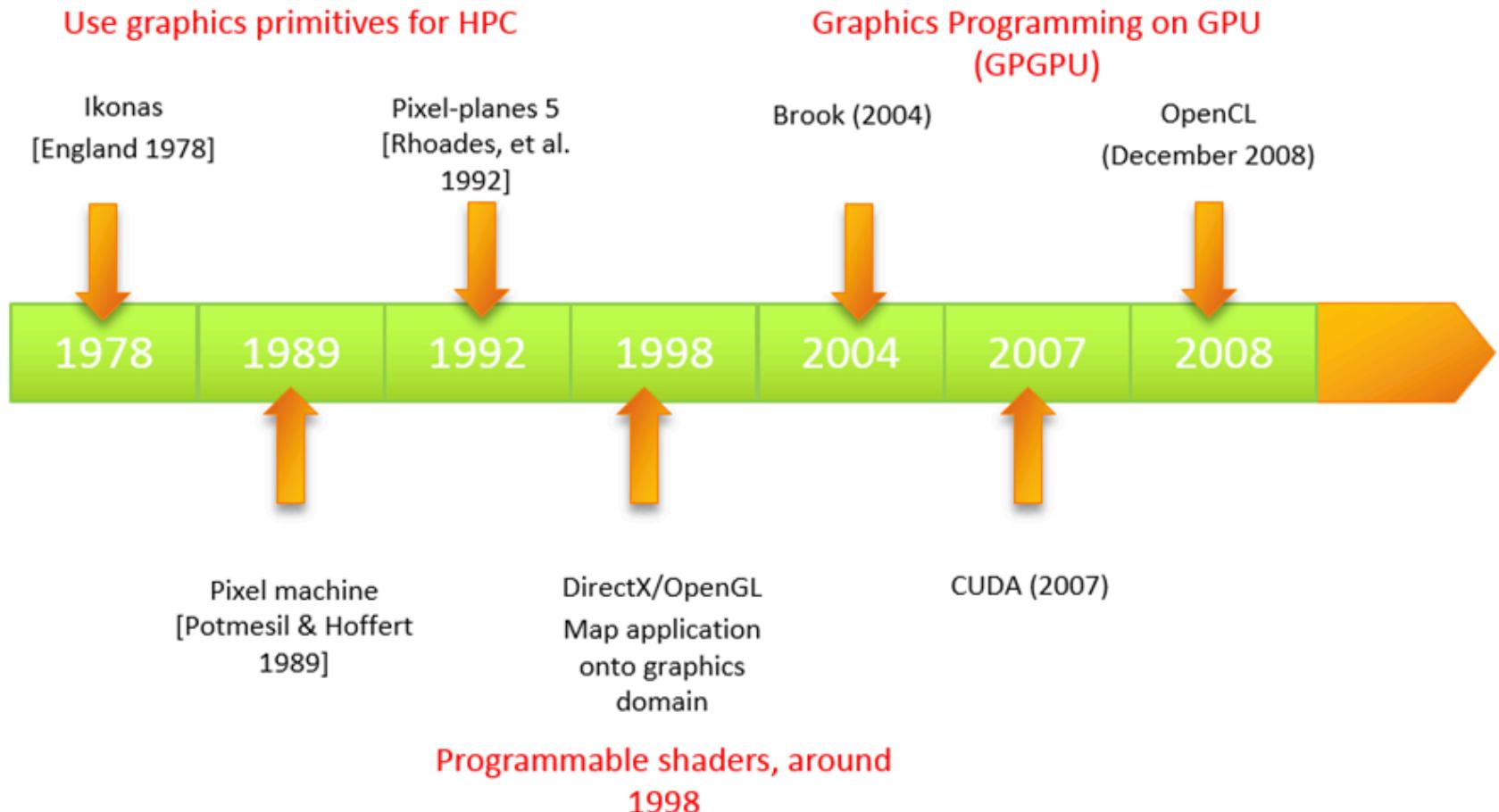
Curs 9

Introducere in CUDA

Ce este CUDA?

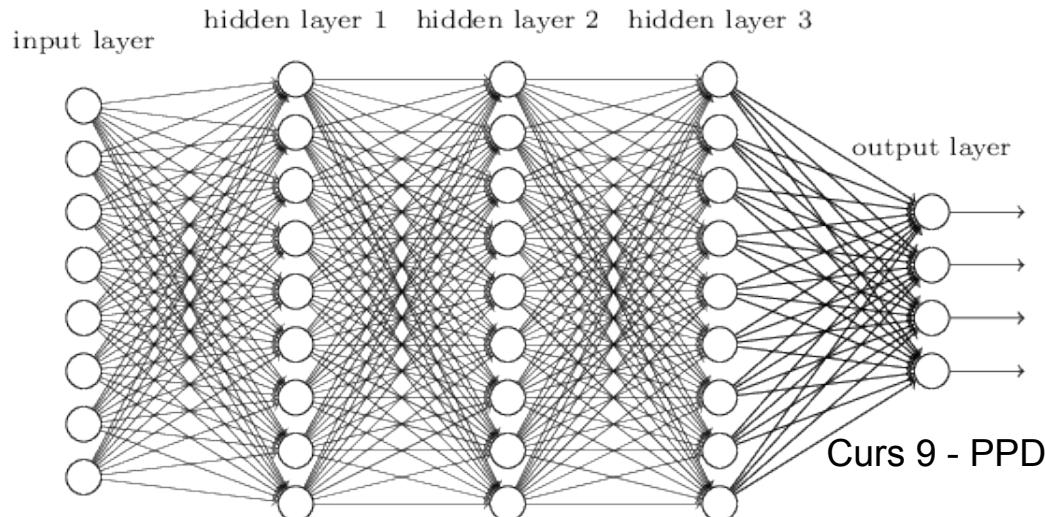
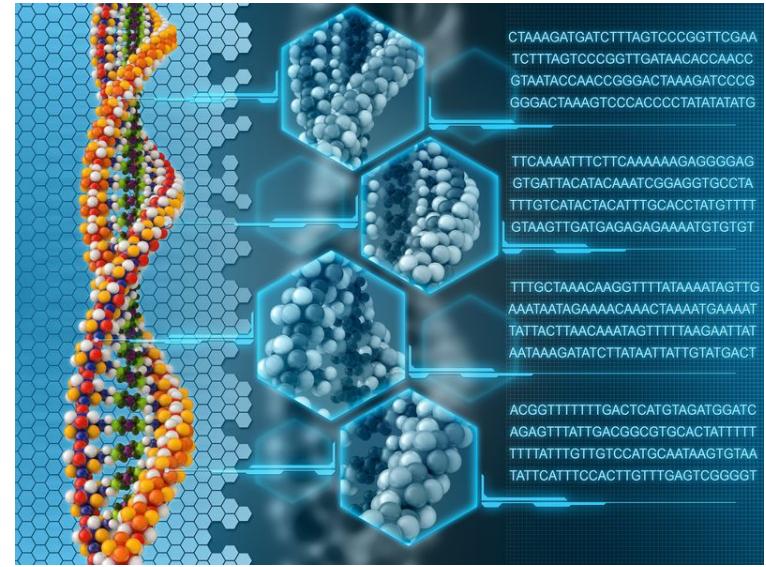
- Compute Unified Device Architecture”
- o platforma de programare paralela->
- Arhitectura care foloseste GPU pt calcul general
 - permite cresterea performantei
- Released by NVIDIA in 2007
- Model de programare
 - Bazat pe extensii C / C++ - pt a permite ‘heterogeneous programming’
 - API pt gestionarea device-urilor, a memoriei etc.

Historic



Aplicatii

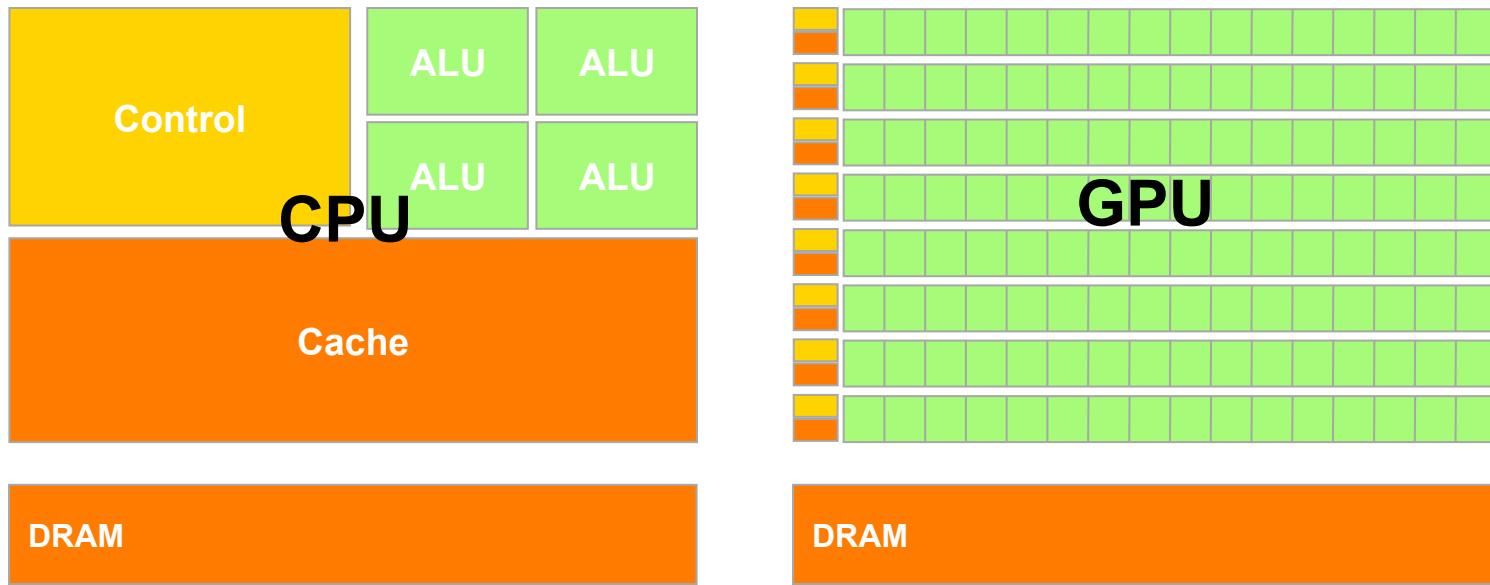
- Bioinformatica
 - Calcul financiar
 - Deep learning
 - Molecular dynamics simulation
 - Video and audio coding and manipulation
 - 3D imaging and visualization
 - Consumer game physics
 - virtual reality products
 - ...



GPGPU

- GPU au devenit mai puternice
 - Mai multă putere de calcul
 - Memory bandwidth (on chip) ridicată
- General Purpose GPU (GPGPU)
- Sute de mii de core-uri în GPU care rulează threaduri în paralel.
- core-uri mai slabe dar ... multe...

CPU vs. GPU



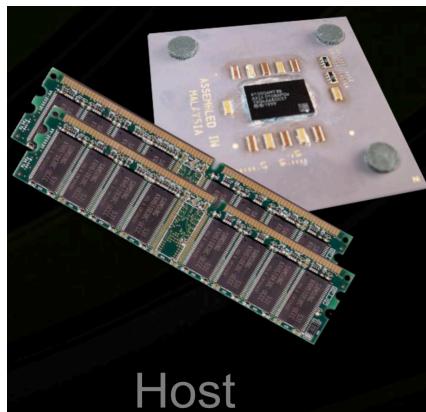
Terminologie

Host: = CPU si memoria asociata

Device: = GPU si memoria asociata

- device

- Is a coprocessor to the CPU or **host**
- Has its own DRAM (**device memory**)
- Runs many **threads in parallel**
- Is typically a **GPU** but can also be another type of parallel processing device



Host

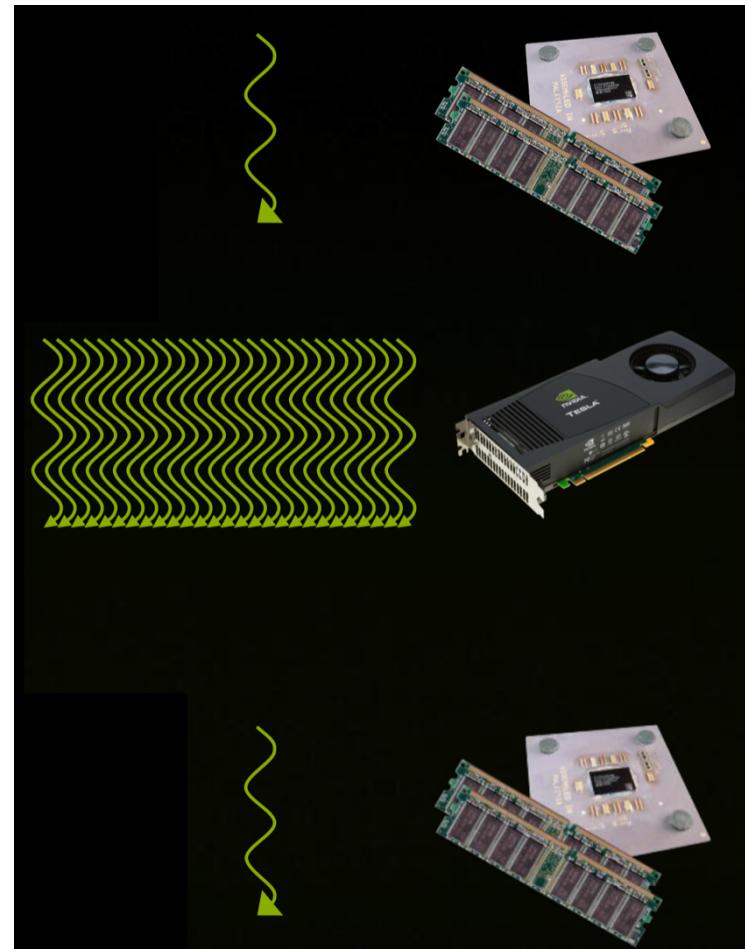
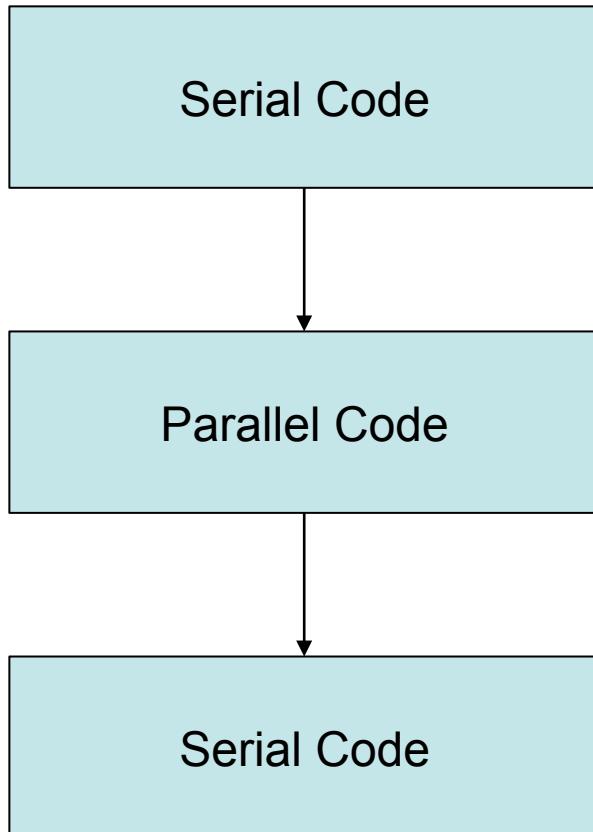
Diferente intre threadurile GPU si CPU

- GPU threads are extremely lightweight
 - Very little creation overhead
- GPU needs 1000s of threads for full efficiency
 - Multi-core CPU needs only a few

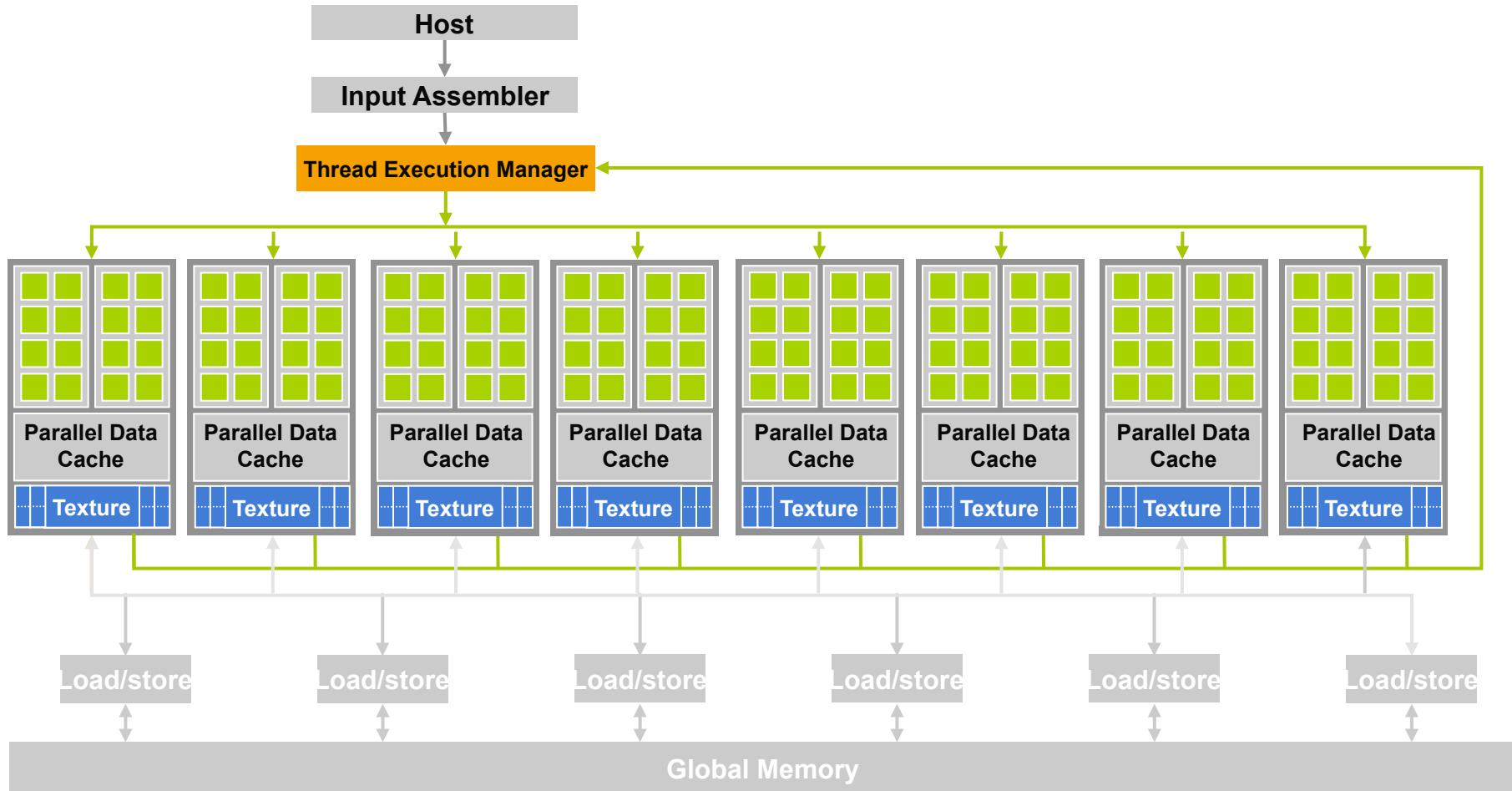


Device

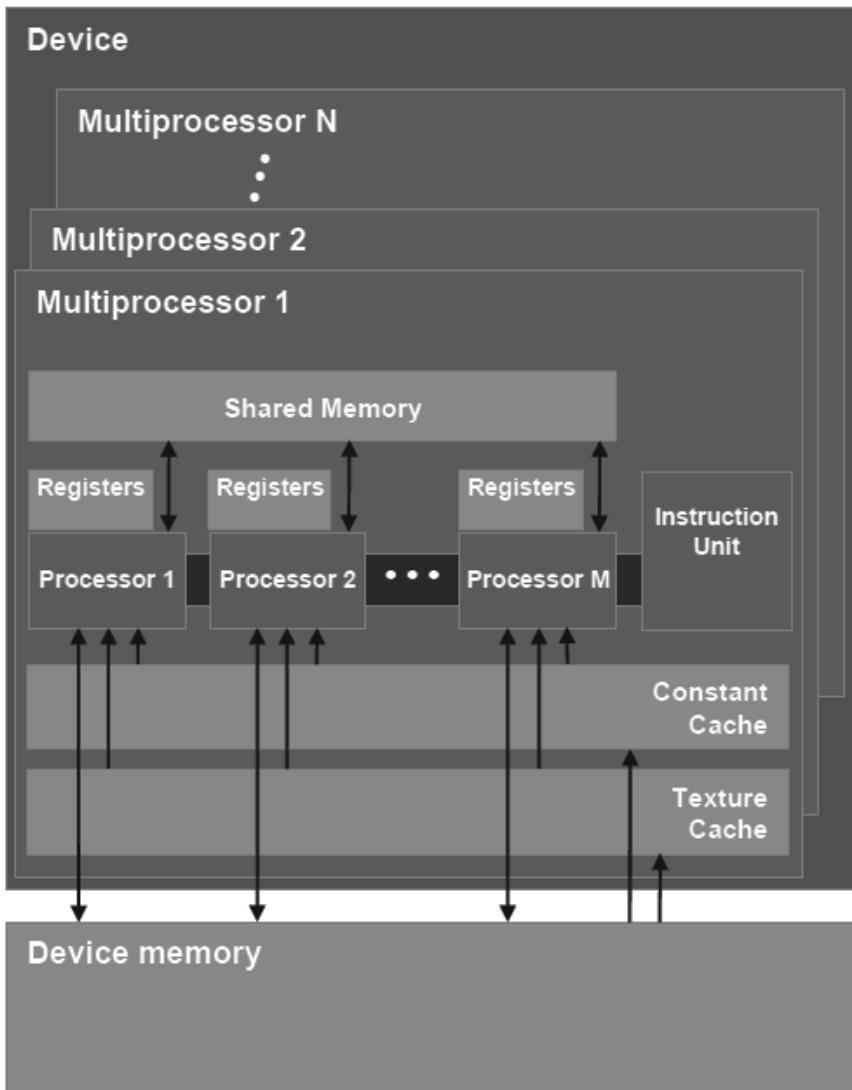
Heterogeneous Computing



Architecture of a CUDA-capable GPU



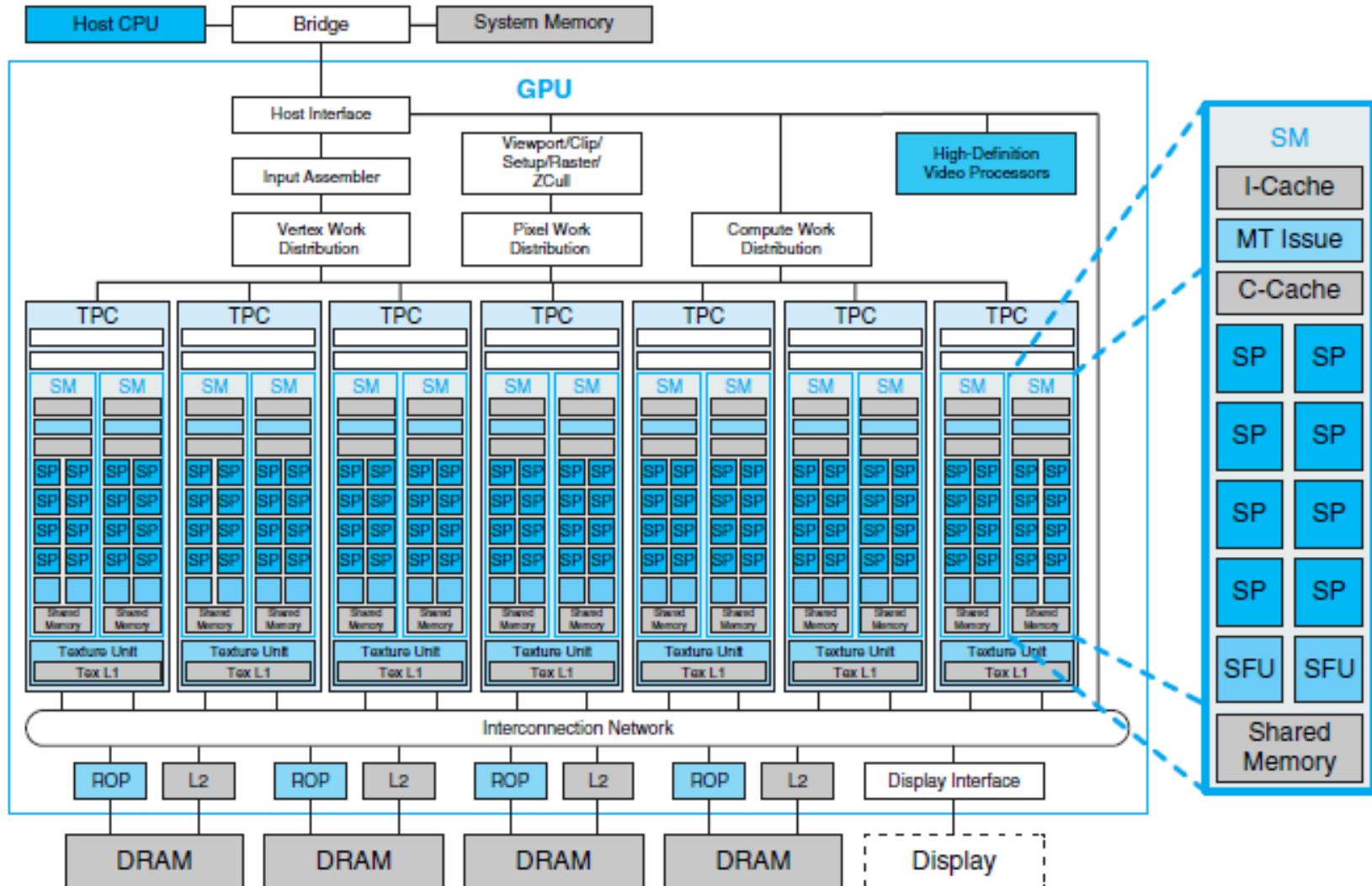
GPU device

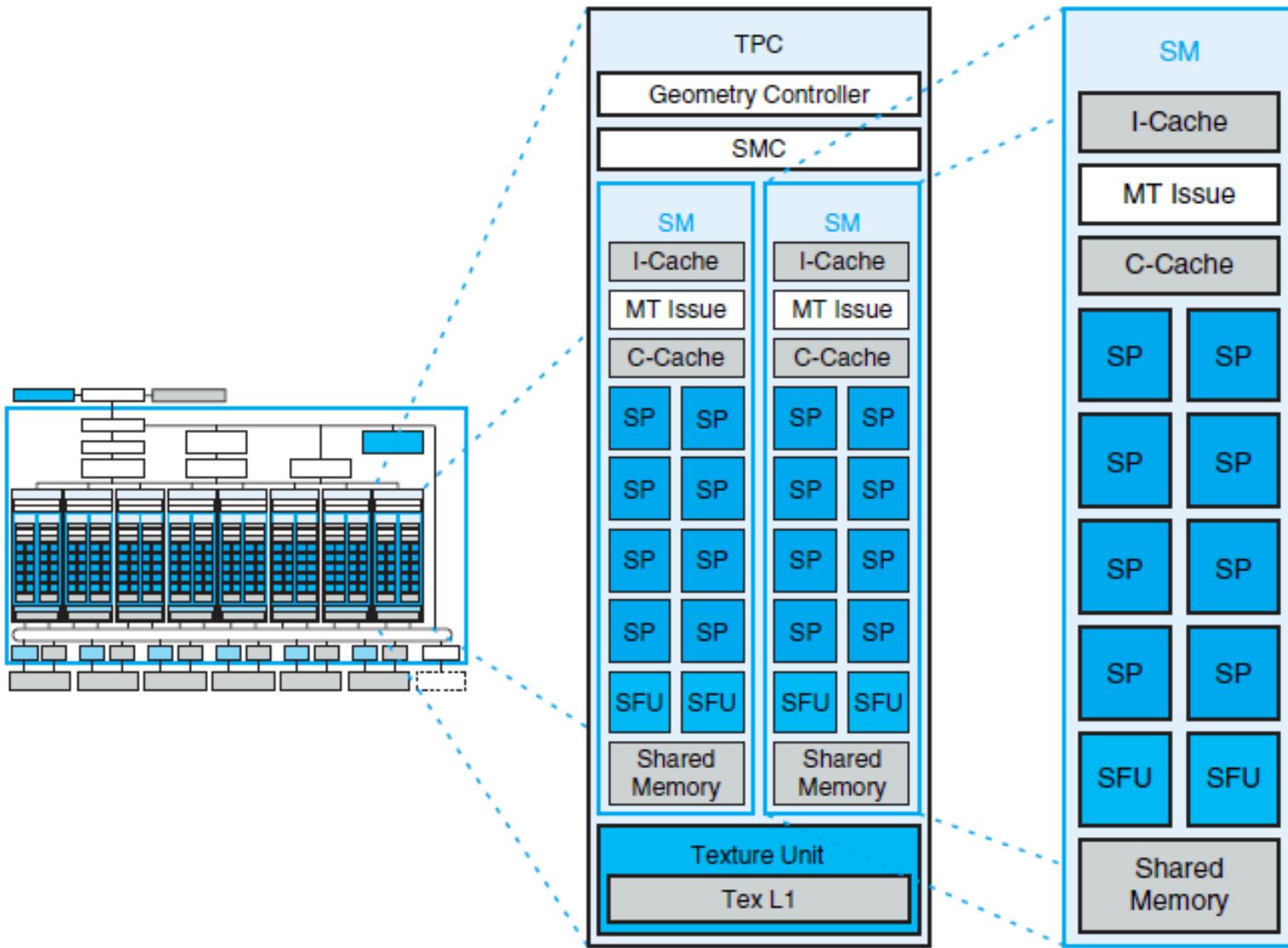


- Global memory
- Streaming Multiprocessors (SM) where each SM has:
 - Control units
 - Registers
 - Execution pipelines
 - Caches

<https://tatourian.com/2013/09/03/nvidia-gpu-architecture-cuda-programming-environment/>

GeForce 8800 Architecture





Hardware Requirement

Cerinte pt GPU

- CUDA-capable GPU
 - Lista device-urilor acceptate: <https://developer.nvidia.com/cuda-gpus>
- Instalare-> documentatie
 - <http://docs.nvidia.com/cuda/>

Fluxul de procesare

- Se copiaza datele in memoria GPU
- Se executa calcul in GPU (mii de threaduri)
- Se copiaza datele din memoria GPU in memoria host
- Kernel: codul GPU care se va executa

Extended C

- **Declspecs**

- **global, device, shared, local, constant**

```
__device__ float filter[N];  
__global__ void convolve (float *image) {  
  
    __shared__ float region[M];  
    ...  
    region[threadIdx] = image[i];  
  
    __syncthreads()  
    ...  
  
    image[j] = result;  
}  
  
// Allocate GPU memory  
void *myimage = cudaMalloc(bytes)  
  
// 100 blocks, 10 threads per block  
convolve<<<100, 10>>> (myimage);
```

- **Keywords**

- **threadIdx, blockIdx**

- **Intrinsics**

- **__syncthreads**

- **Runtime API**

- **Memory, symbol, execution management**

- **Function launch**

CUDA Function Declarations

	Executed on the:	Only callable from the:
<code><u>device</u> float DeviceFunc()</code>	device	device
<code><u>global</u> void KernelFunc()</code>	device	host
<code><u>host</u> float HostFunc()</code>	host	host

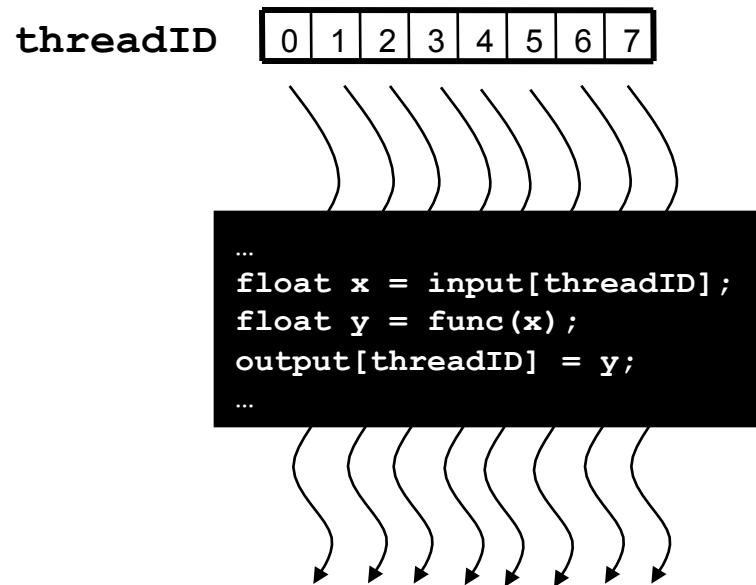
- `global` defines a kernel function
 - **Must return `void`**

CUDA Function Declarations

- **device** functions cannot have their address taken
- For functions executed on the device:
 - *No recursion*
 - *No static variable declarations inside the function*
 - *No variable number of arguments*

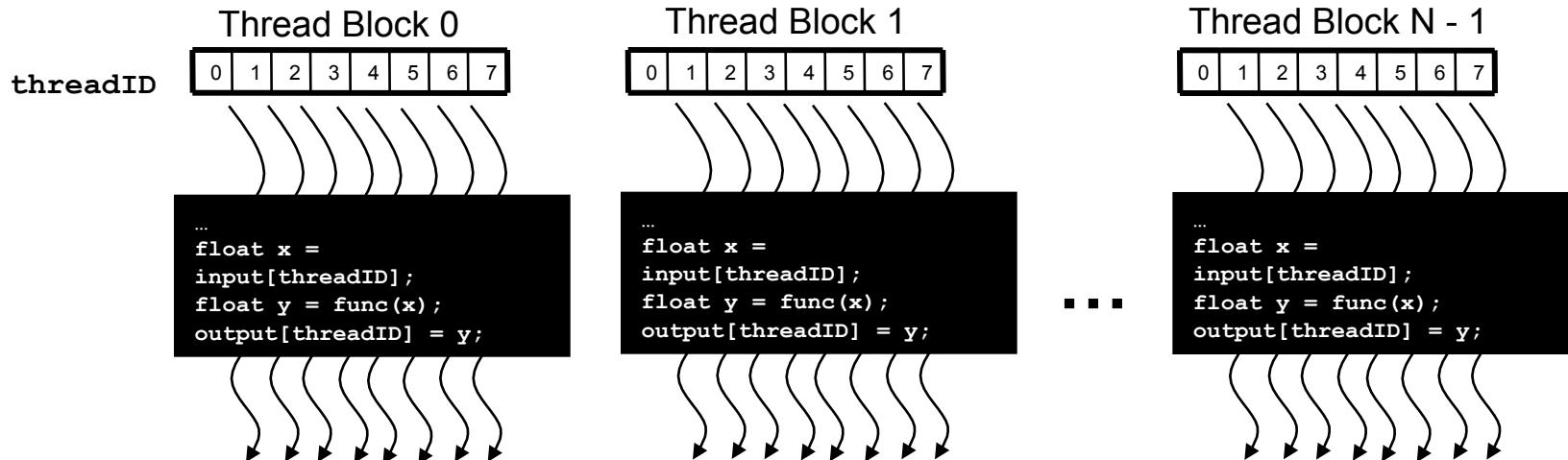
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
 - All threads run the same code (SPMD)
 - Each thread has an ID



Thread Blocks: Scalable Cooperation

- Thread-urile dintr-un bloc pot coopera via **shared memory, atomic operations, synchronization barrier**
- Thread-urile din blocuri diferite nu pot coopera!



Exemplu: Adunare vectori

Parallel code: kernel

```
__global__ void vectorAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

Cod Serial : setup

```
int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 1<<20;
    // Host input vectors
    double *h_a;  double *h_b;
    //Host output vector
    double *h_c;
    // Device input vectors
    double *d_a;  double *d_b;
    //Device output vector
    double *d_c;
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, no_bytes);
    cudaMalloc(&d_b, no_bytes);
    cudaMalloc(&d_c, no_bytes);

    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, no_bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, no_bytes, cudaMemcpyHostToDevice);
```

Cod Serial: apelare kernel, colectare rezultate

```
int blockSize, gridSize;
// Number of threads in each thread block
blockSize = 1024;
// Number of thread blocks in grid
gridSize = (int)ceil((float)n/blockSize);
// Execute the kernel
vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

// Copy array back to host
cudaMemcpy( h_c, d_c, no_bytes, cudaMemcpyDeviceToHost );

// Release device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
// Release host memory
free(h_a);
free(h_b);
free(h_c);
return 0;
}
```

><><><><><><><><><

- Lansare kernel = executie __global__ function <<<>>>

vectorAdd<<<3, 4>>>(d_a, d_b, d_c);

Functia Kernel

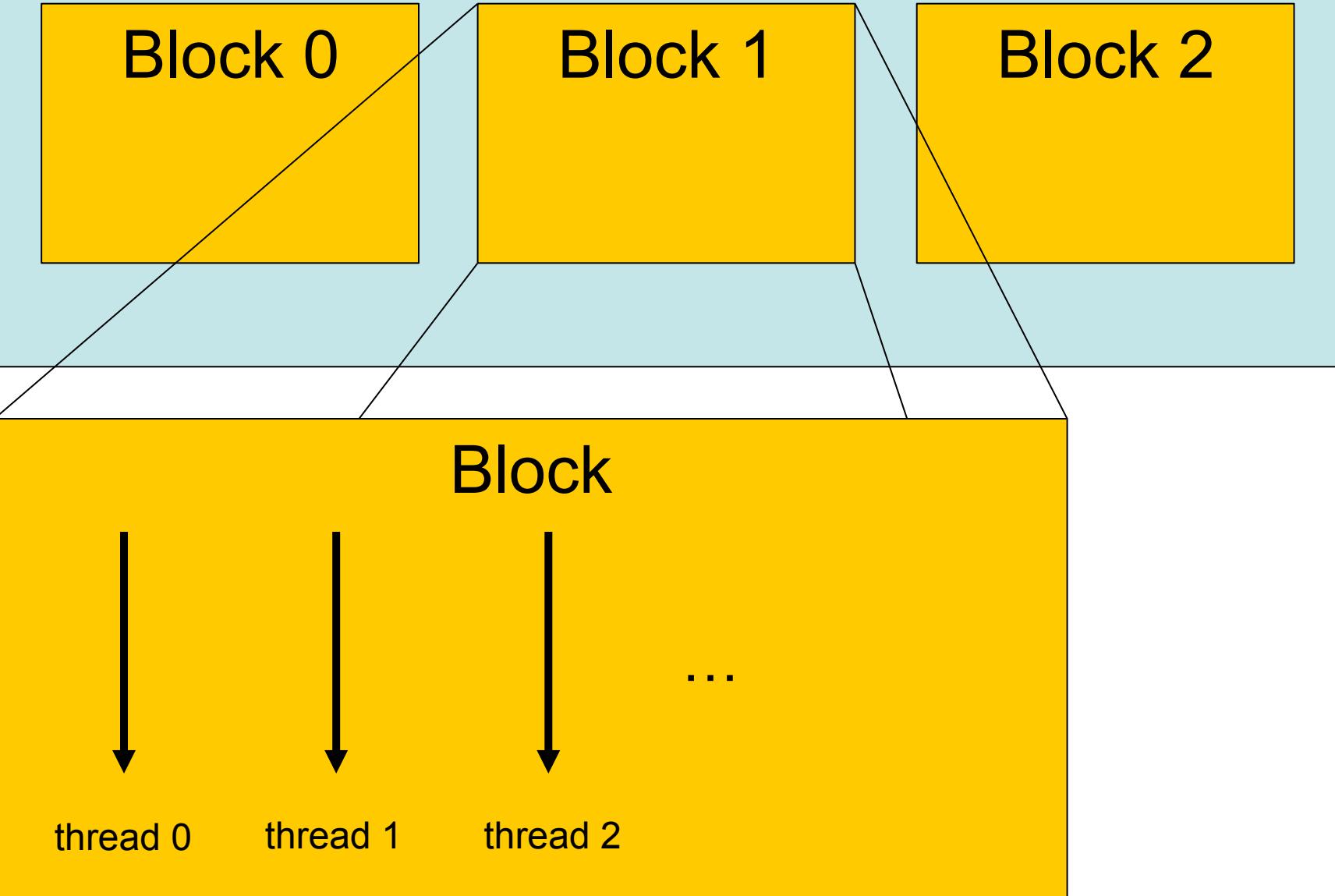
Cate blocuri si cate
threaduri per bloc

parametrii

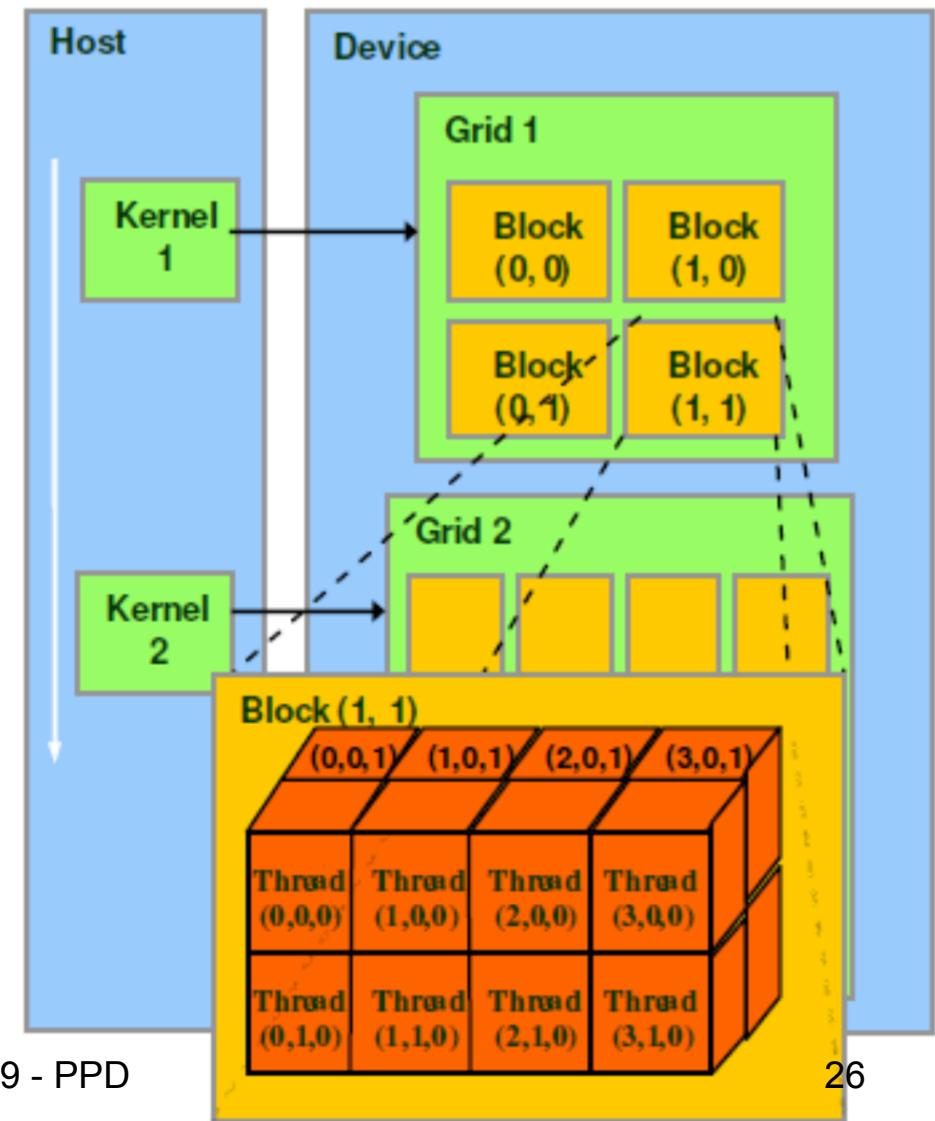
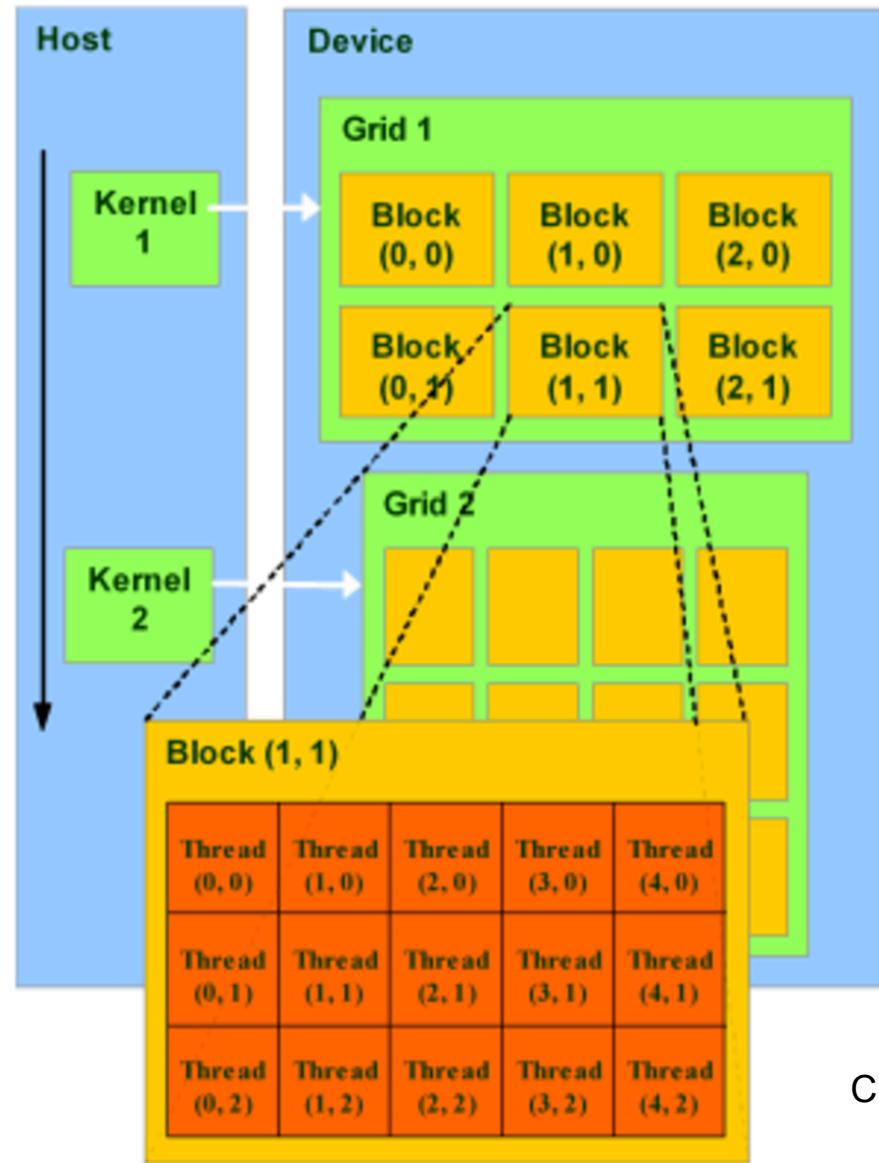
Thread, Block, Grid

- CUDA foloseste o structura ierarhica :
 - grid
 - block
 - thread

Grid



- Grid-ul poate sa fie compus din blocuri organizate 1D, 2D sau 3D
- Blocurile pot fi compuse din threaduri organizate 1D, 2D sau 3D



- <<<blocks per grid, threads per block>>>
 - <<<1, 1>>> : a grid with 1 block inside, and one block is consisted of 1 thread.
Total threads: 1
 - <<<2, 3>>>: a grid with 2 blocks inside, and one block is consisted of 3 threads.
Total threads: 6

dim3 struct

```
struct __device_builtin__ dim3
{
    unsigned int x, y, z;
#ifndef __cplusplus
    __host__ __device__ dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1) : x(vx), y(vy), z(vz) {}
    __host__ __device__ dim3(uint3 v) : x(v.x), y(v.y), z(v.z) {}
    __host__ __device__ operator uint3(void) { uint3 t; t.x = x; t.y = y; t.z = z; return t; }
#endif /* __cplusplus */
};
```

Ex.

```
dim3 grid(256);           // defines a grid of 256 x 1 x 1 blocks
dim3 block(512,512);      // defines a block of 512 x 512 x 1 threads
```

```
foo<<<grid,block>>>(...);
```

CUDA Built-In Variables

- `blockIdx.x`, `blockIdx.y`, `blockIdx.z` are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.
- `threadIdx.x`, `threadIdx.y`, `threadIdx.z` are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this particular block.
- `blockDim.x`, `blockDim.y`, `blockDim.z` are built-in variables that return the “block dimension” (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis).
- The full global thread ID in x dimension can be computed by:
`x = blockIdx.x * blockDim.x + threadIdx.x;`

Exemplul 1

- `BlockIdx.x` is the x number of the block
- `BlockDim.x` is the total threads in x dimension (width)
- If we launch `vector_add<<<2, 4>>>`
 - Primul thread (block(0), thread(0)):
 $\text{idx} = 0 + 0 * 4 = 0$
 - Threadul al 5-lea (block(1), thread(0)):
 $\text{idx} = 0 + 1 * 4 = 4$



Exemplul 2

Global Thread ID

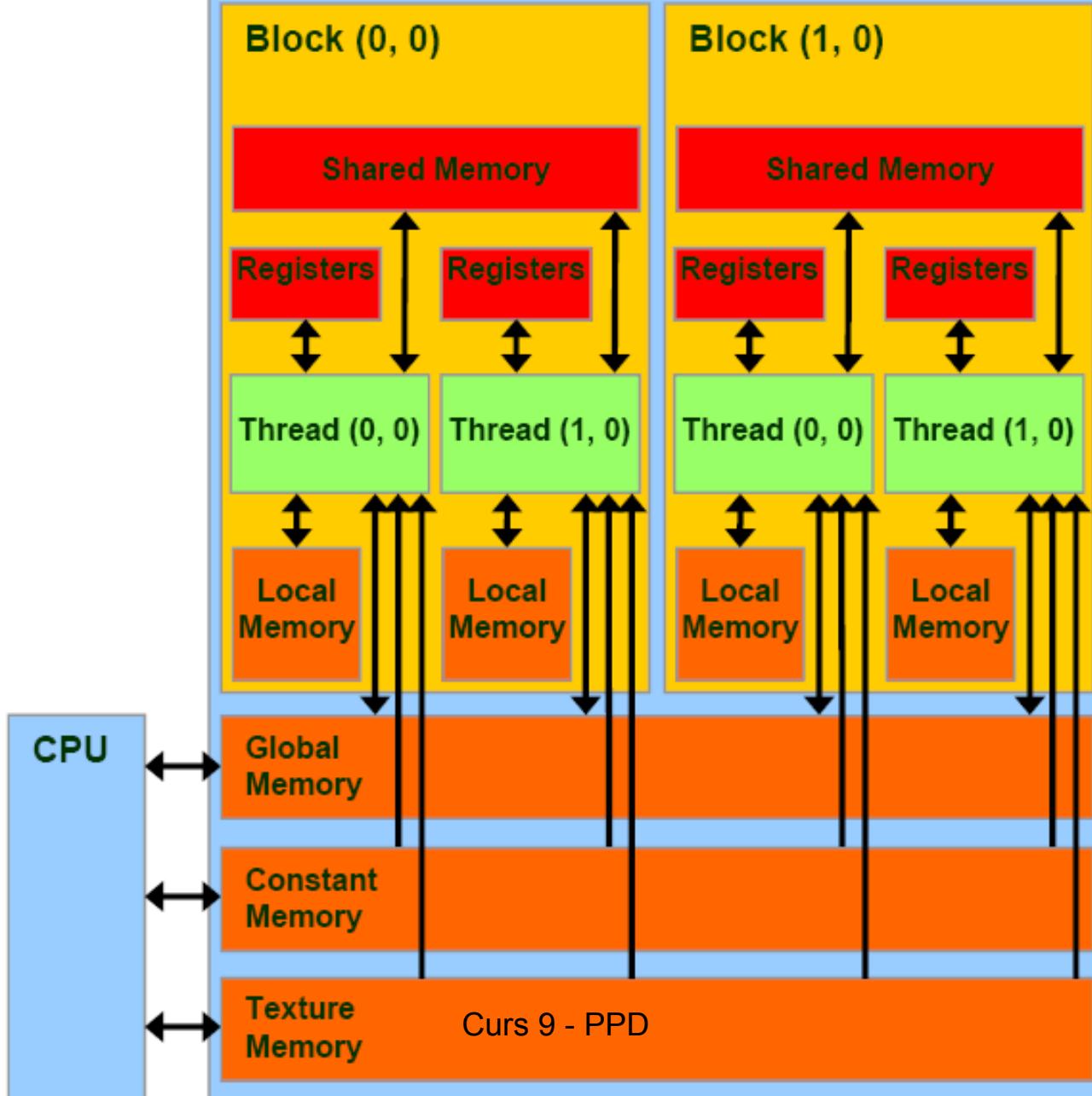
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
threadIdx.x								threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2								blockIdx.x = 3							

- Assume a hypothetical ID grid and ID block architecture: 4 blocks, each with 8 threads.
- For Global Thread ID 26:
 - $\text{gridDim.x} = 4 \times 1$
 - $\text{blockDim.x} = 8 \times 1$
 - $\text{Global Thread ID} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$
 - $= 3 \times 8 + 2 = 26$

Memory Types

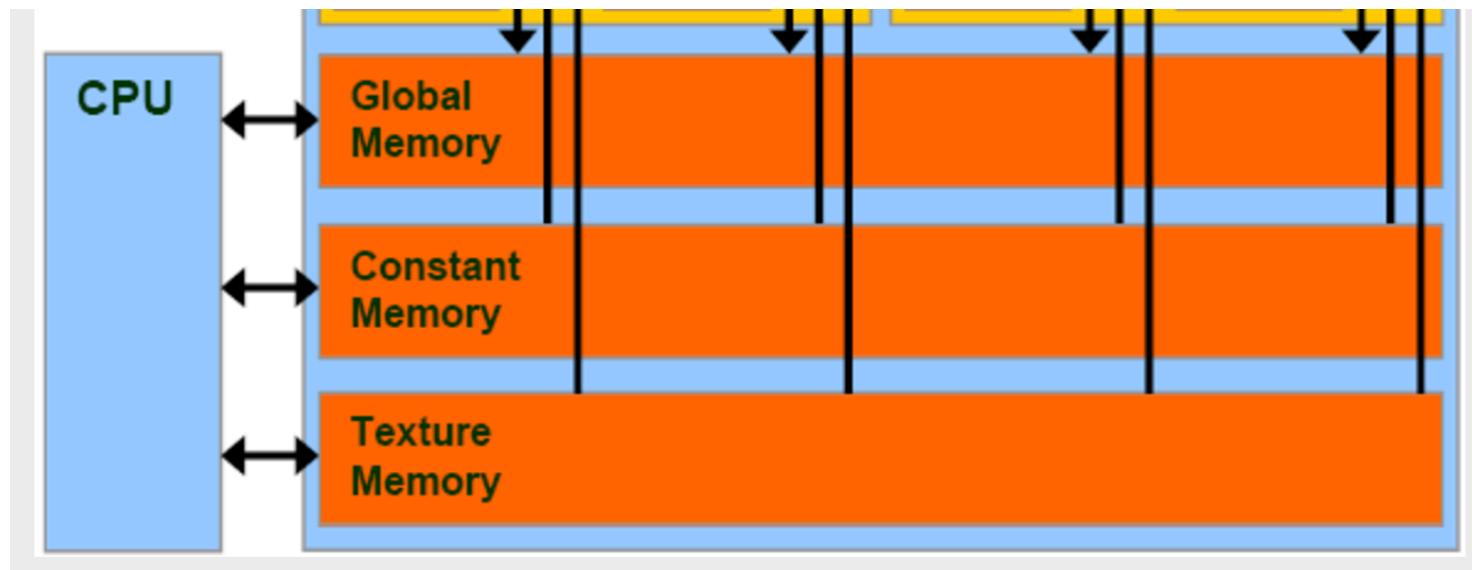
- CUDA foloseste 5 tipuri de memorie fiecare cu proprietati diferite
- Proprietati:
 - Size
 - Access speed
 - Read/write, read only

GPU Grid



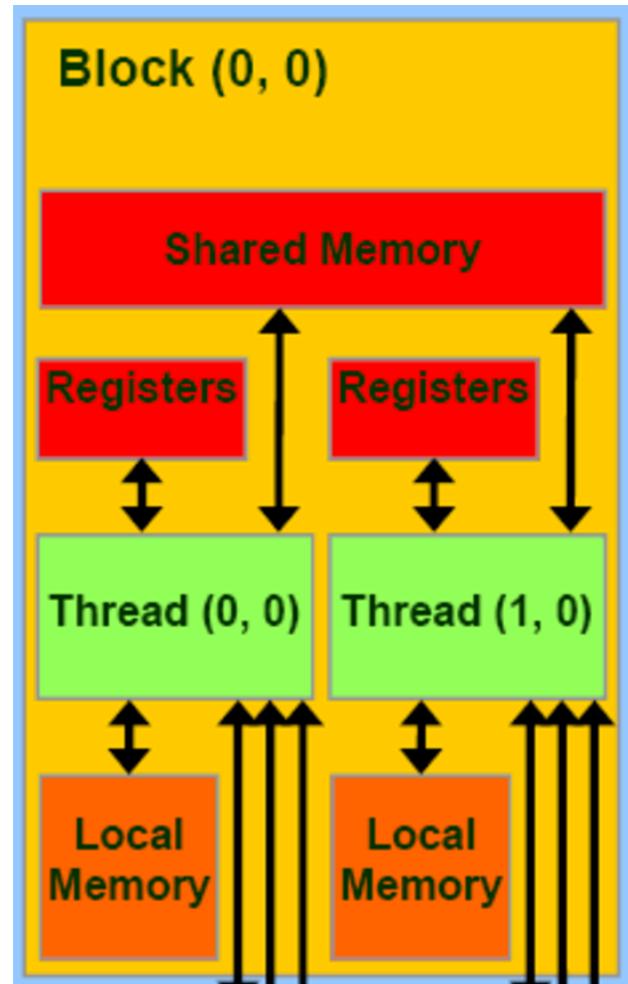
Memory Types

- Global memory: cudaMalloc memory, the size is large, but slow (has cache)
- Texture memory: read only, cache optimized for 2D access pattern
- Constant memory: slow but with cache (8KB)



Memory Types

- Local memory:
local to thread, but it is as slow as global memory
- Shared memory:
100x fast to global
memory, it is accessible
to all threads in one
block



Memory Types

- Shared memory is very fast, but usually only has 49KB (can be configured to 64KB).
- Actually, shared memory is the same as “L1 cache” of CPU, but controllable by user.
- One block has one shared memory, that’s one reason why we manage the threads in grid and block way!

Shared Memory Example

- 1D stencil:



- If we program this way

```
1 __global__ stencil_kernel(int *stencil, int *result) {
2     int idx = threadIdx.x + blockIdx.x * blockDim.x;
3     int sum = 0;
4     if(idx > RADIUS && idx < OUTPUT_BIN + RADIUS) {
5         for(int offset = 0; offset <= RADIUS; ++offset) {
6             sum += stencil[idx + offset];
7         }
8     }
9     result[idx] = sum;
10 }
```

- With shared memory:

```
1 __global__ stencil_kernel(int *stencil, int *result) {
2     __shared__ tmp[OUTPUT_BIN + 2 * RADIUS];
3     int idx = threadIdx.x + blockIdx.x * blockDim.x;
4     int sum = 0;
5
6     // each thread copy one stencil value to tmp
7     tmp[idx] = stencil[idx];
8     if(idx > RADIUS && idx < OUTPUT_BIN + RADIUS) {
9         for(int offset = 0; offset <= RADIUS; ++offset) {
10             sum += tmp[idx + offset]
11         }
12     }
13     result[idx] = sum;
14 }
```

- But actually, the result is wrong...
- If thread 5 done copy, and starts calculate ($\text{tmp}[3] + \text{tmp}[4] + \text{tmp}[5]$), but thread 6 did not finish copy $\text{tmp}[5]$?
- `__syncthreads()` makes all threads in a block synchronize!
- `__syncthreads()` is a barrier, will block all threads wait until all threads reach the line

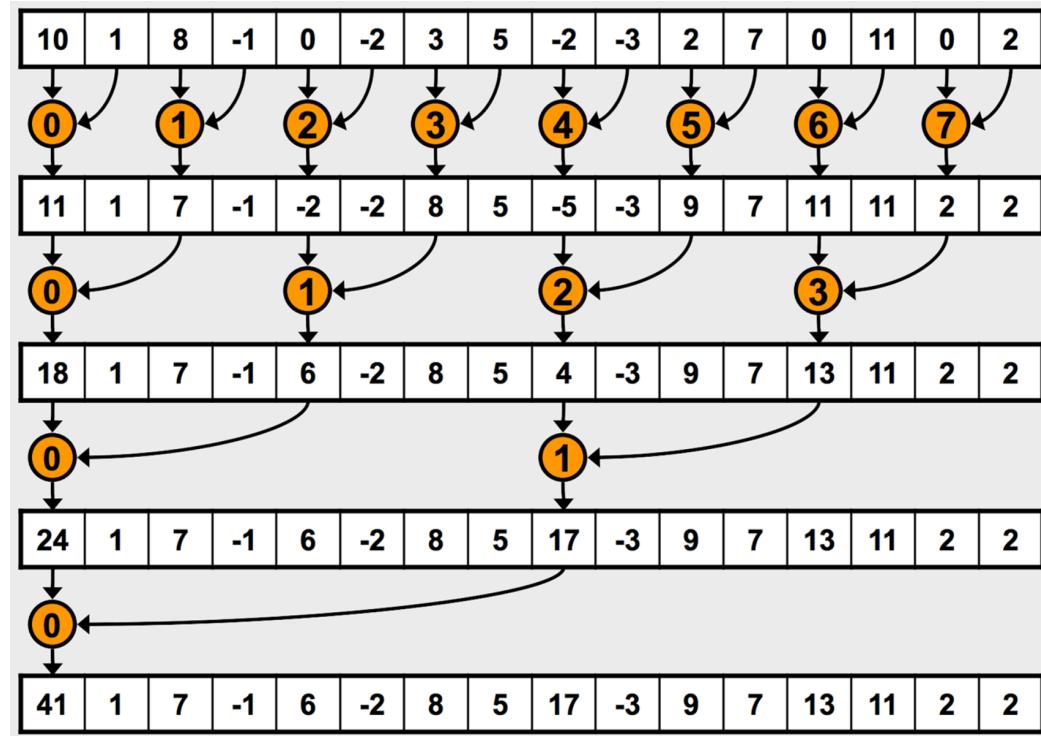
- Then we have the correct 1D stencil

```

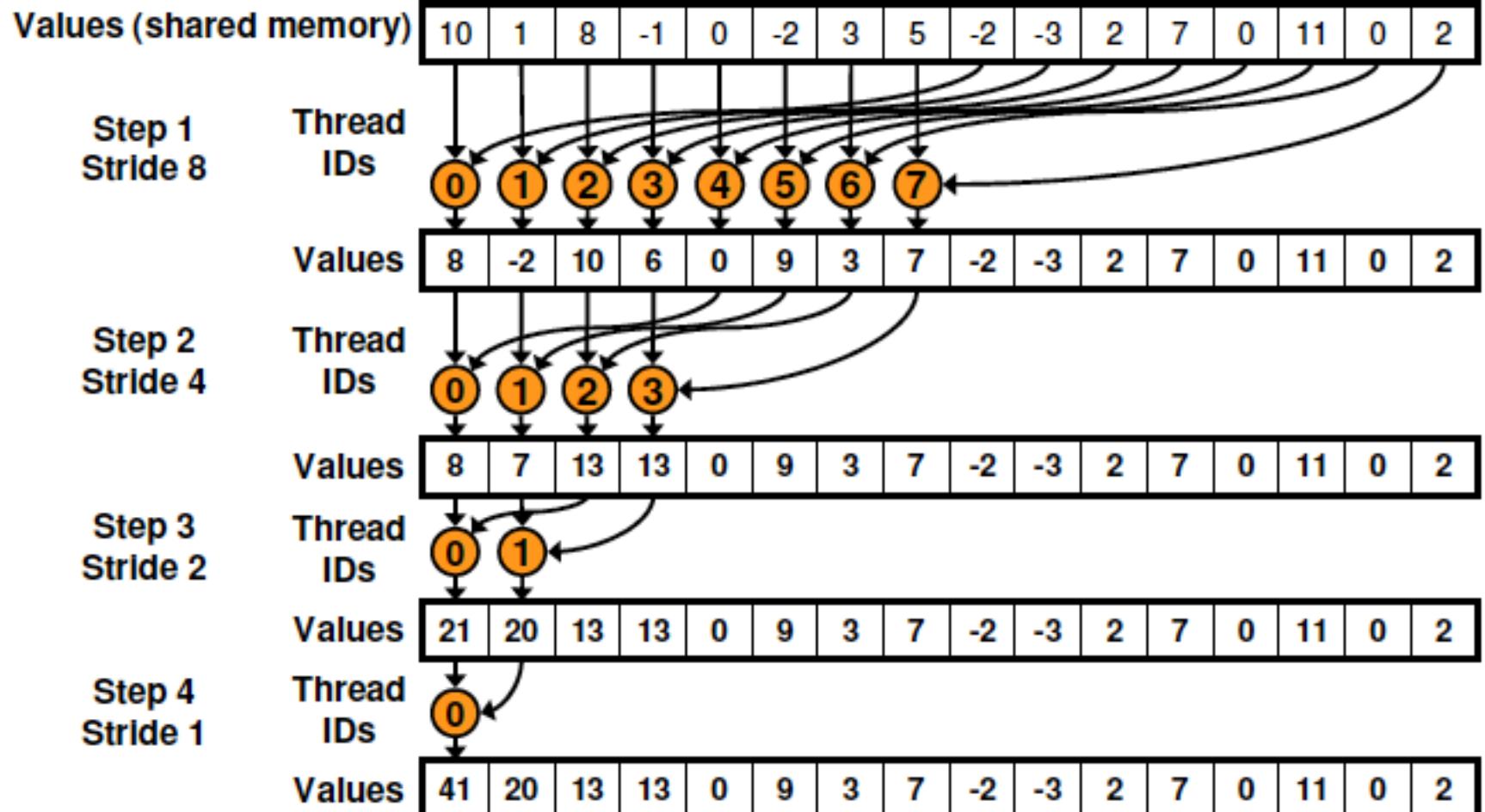
1 __global__ stencil_kernel(int *stencil, int *result) {
2     __shared__ tmp[OUTPUT_BIN + 2 * RADIUS];
3     int idx = threadIdx.x + blockIdx.x * blockDim.x;
4     int sum = 0;
5
6     // each thread copy one stencil value to tmp
7     tmp[idx] = stencil[idx];
8     // wait all thread done copy
9     __syncthreads();
10    if(idx > RADIUS && idx < OUTPUT_BIN + RADIUS) {
11        for(int offset = 0; offset <= RADIUS; ++offset) {
12            sum += tmp[idx + offset]
13        }
14    }
15    result[idx] = sum;
16 }
```

Reduction

- CUDA is running multi-threads, you can do the reduce to do:
 - Summation
 - Search
 - Etc
- !!!
with
`__syncthreads()`



O alta varianta... dar similară



Coordinare -> Host & Device

- Kernel-urile sunt pornite asincron
- Controlul este returnat catre CPU imediat
- CPU necesita sincronizare inainte sa foloseasca rezultatele obtinute pe device
- `cudaMemcpy()` blocheaza CPU pana cand copierea se finalizeaza.
 - Copierea incepe atunci cand toate apelurile CUDA anterioare s-au terminat;
- `cudaMemcpyAsync()` Asynchronous ->nu blocheaza CPU
- `cudaDeviceSynchronize()` blocheaza CPU pana toate apelurile CUDA se finalizeaza.

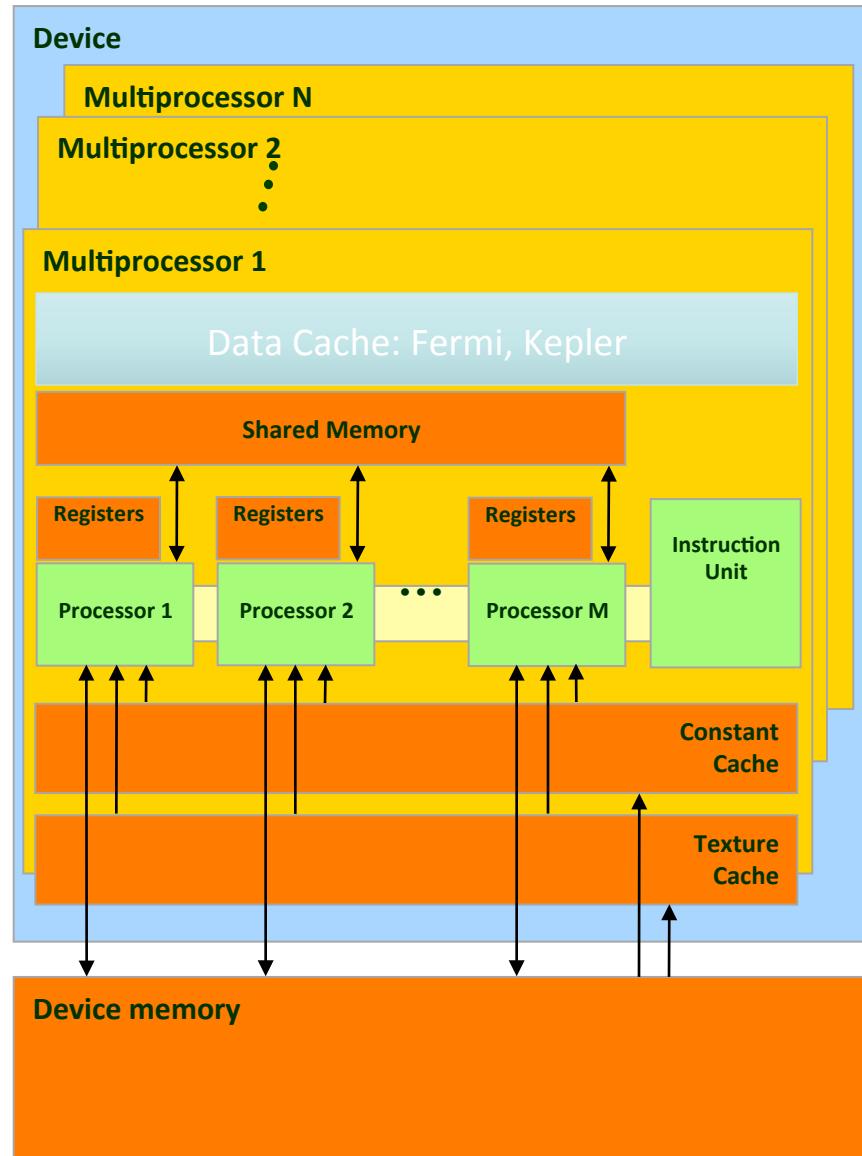
NVIDIA GPU Execution Model

I. SIMD Execution of
warpsize=M threads
(from single block)

II. Multithreaded Execution across different
instruction streams within block

- Also possibly across different
blocks if there are more blocks than
SMs

III. Each block mapped to single SM
– No direct interaction across SMs



SIMT = Single-Instruction Multiple Threads

- Introdus de catre Nvidia
- Combina executia de tip SIMD din interiorul unui Block (pe un SM) cu executia SPMD intre Blocks (distribuita pe /across SMs)

- În GPU, unitatea de procesare este SP (streaming processor);
 - Mai multe SP și alte componente formează un SM (streaming multiprocessor);
 - Mai multe SM formează un TPC (texture processing cluster)
- În CUDA, putem spune că
 - un grid este procesat de catre întreg device-ul GPU,
 - un block este procesat de catre un SM, and
 - un thread este procesat de catre un SP.

WRAP

- ❖ fiecare SM are 8 SP si pot fi 4 instructiuni in executie – pipelined - => 32
 - Cate 32 threads compun un wrap.
 - Daca se alege un numar care nu se divide cu 32 atunci restul va forma un wrap (-> ineficient)
 - De fiecare data un SM proceseaza doar un wrap si astfel ca daca sunt mai putin de 32 threads intr-un wrap atunci unele SP nu sunt folosite.
 - ***The warp size is the number of threads running concurrently on an MP.***
 - De fapt threadurile ruleaza si in paralel dar si pipelined
 - fiecare MP contine cate 8 SP
 - cea mai rapida instructiune necesita 4 cicluri (cycles).
 - => fiecare SP poate avea 4 instructiuni in propriul pipeline, deci avem un total de $8 \times 4 = 32$ instructiuni care se executa concurrent.
 - In interiorul unui warp, threadurile au indecsi secventiali:
 - Primul 0..31, urmatorul 32..63 s.a.md. Pana la numarul total de threaduri dintr-un block.[\[http://cuda-programming.blogspot.ro/2013/01/what-is-warp-in-cuda.html\]](http://cuda-programming.blogspot.ro/2013/01/what-is-warp-in-cuda.html)

Efect-> wrap

- Omogenitatea threadurilor dintr-un wrap are un efect important asupra performantei calculului (computational throughput).
 - Daca toate threadurile executa aceeasi instructiune atunci toate SP dintr-un MP pot executa aceeasi instructiune in paralel.
 - Daca un thread dintr-un wrap executa o instructiune diferita de celelalte atunci un wrap trebuie sa fie partitionat in grupuri de threaduri bazat pe instructiunile care urmeaza sa fie executate; apoi grupurile se executa unul dupa altul.
 - Aceasta serializare reduce ‘throughput-ul’ ape masura ce threadurile devin tot mai heterogene si se impart in grupuri tot mai mici.
- Rezulta ca este important sa se pastreze omogenitatatea pe cat posibil!

Threads in Blocks

- Atunci cand un thread asteapta date, unitatea SM va alege un alt thread pentru a fi executat – astfel se ascunde latenta de acces la memorie.
- Astfel mai multe threaduri dintr-un block pot ascunde mai mult latenta;
 - Dar mai multe thread-uri intr-un block inseamna ca memoria partajata per threaduri este mai mica.
- Recomandarea NVIDIA: un block necesita cel putin 196 threaduri pentru a ascunde latenta corespunzatoare accesului la memorie.

Optimizare

- Evitarea copiilor/transferurilor dintre memoriile CPU si GPU
- folosire cache (shared memory)
- Alegerea potrivita a numarului de blocuri
- Array alignment (alignment at 64 byte boundary)
- Continuous memory access
- Folosirea functiilor din CUDA API

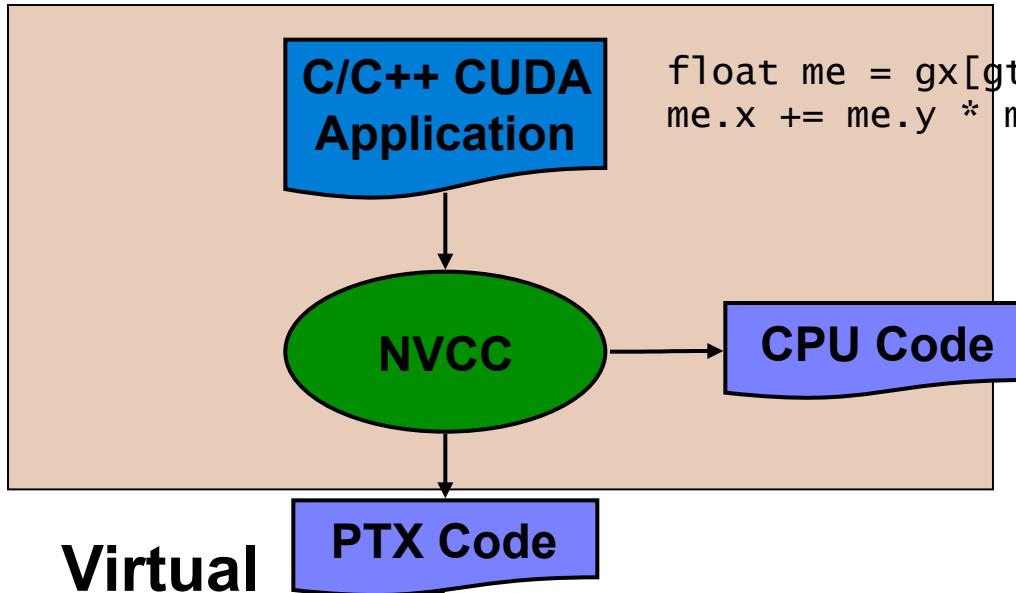
Floating Point Operations

- Results of floating-point computations will slightly differ because of:
 - Different compiler outputs, instruction sets
 - Use of extended precision for intermediate results
 - There are various options to force strict single precision on the host

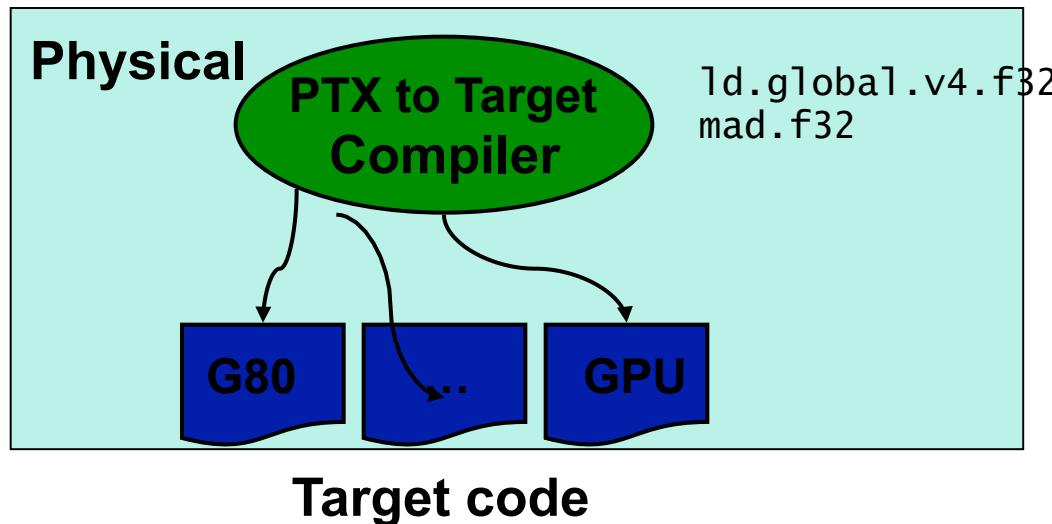
COMPILEARE

EXECUTIE

Compiling a CUDA Program



- Parallel Thread eXecution (PTX)
 - Virtual Machine and ISA
 - Programming model
 - Execution resources and state



Curs 9 - PPD

Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver
 - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC outputs:
 - C code (host CPU Code)
 - Must then be compiled with the rest of the application using another tool
 - PTX
 - Object code directly
 - Or, PTX source, interpreted at runtime

Referinte prezentare

Prezentarea este bazata pe slide-uri din urmatoarele referinte:

- David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010. ECE 498AL, University of Illinois, Urbana-Champaign
- <http://cuda-programming.blogspot.ro/2013/01/what-is-constant-memory-in-cuda.html>
- Li Sung-Chi. Taiwan Evolutionary Intelligence Laboratory. 2016/12/14 Group Meeting Presentation
- Cyril Zeller. CUDA C/C++ Basics. Supercomputing 2011 Tutorial, NVIDIA Corporation
<http://www.nvidia.com/docs/io/116711/sc11-cuda-c-basics.pdf>