

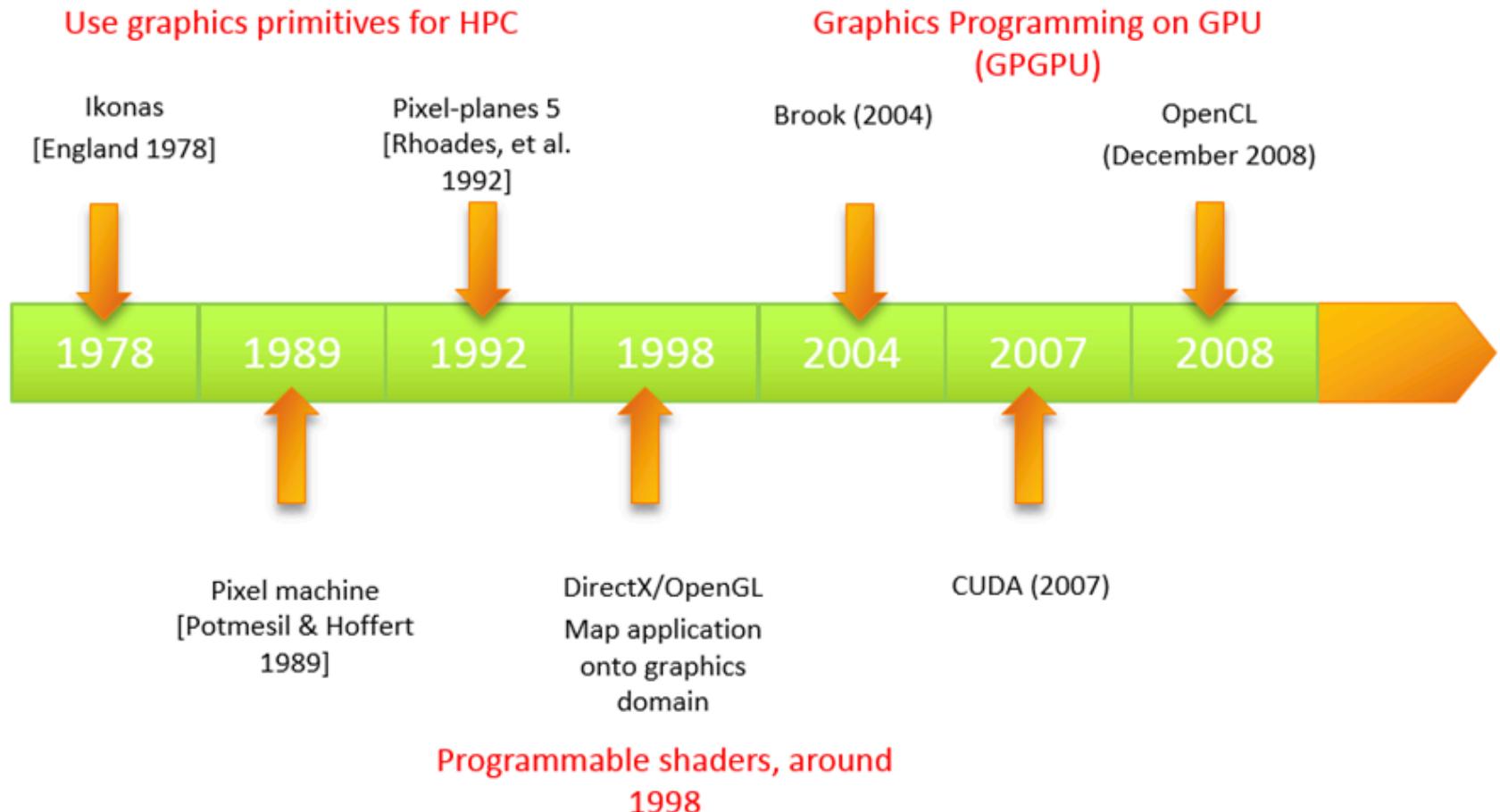
Curs 10

Introducere in CUDA

# Ce este CUDA?

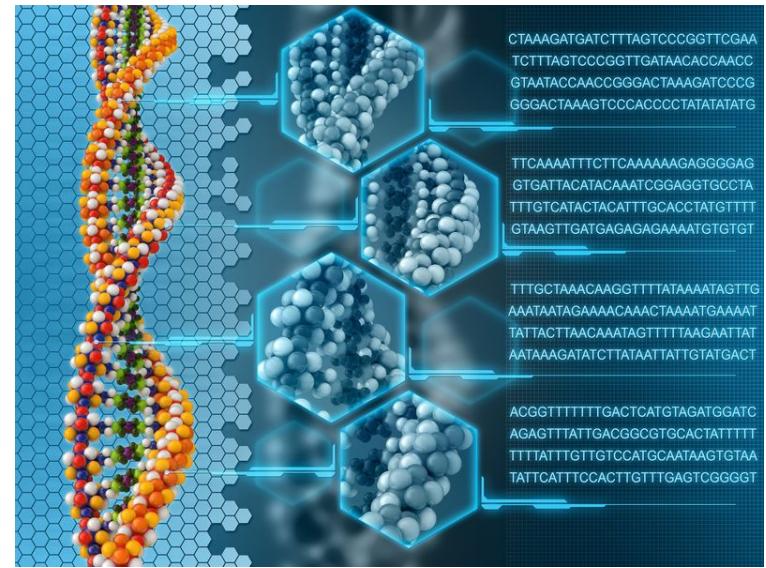
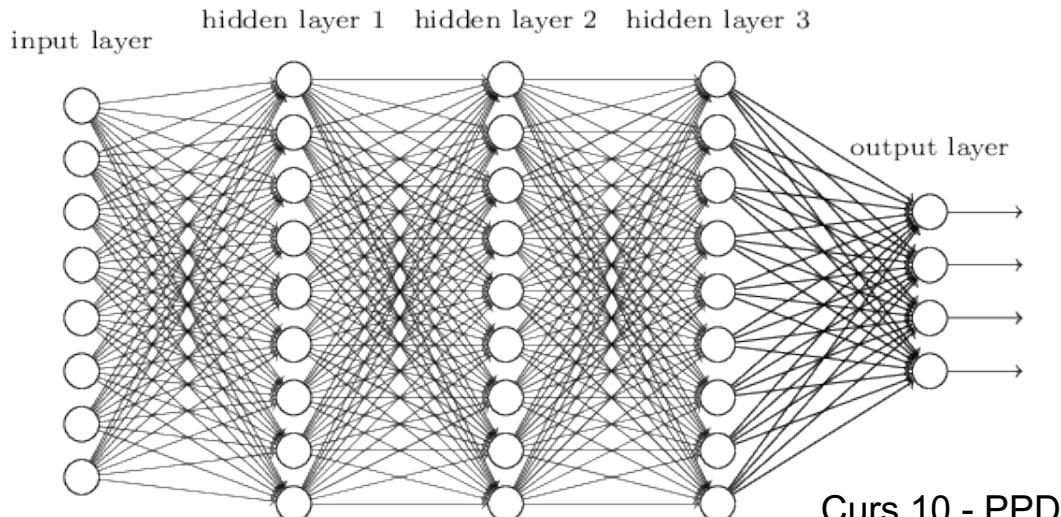
- Compute Unified Device Architecture"
- o platforma de programare paralela->
- Arhitectura care foloseste GPU pt calcul general
  - permite cresterea performantei
- Released by NVIDIA in 2007
- Model de programare
  - Bazat pe extensii C / C++ - pt a permite ‘heterogeneous programming’
  - API pt gestionarea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

# Historic



# Aplicatii

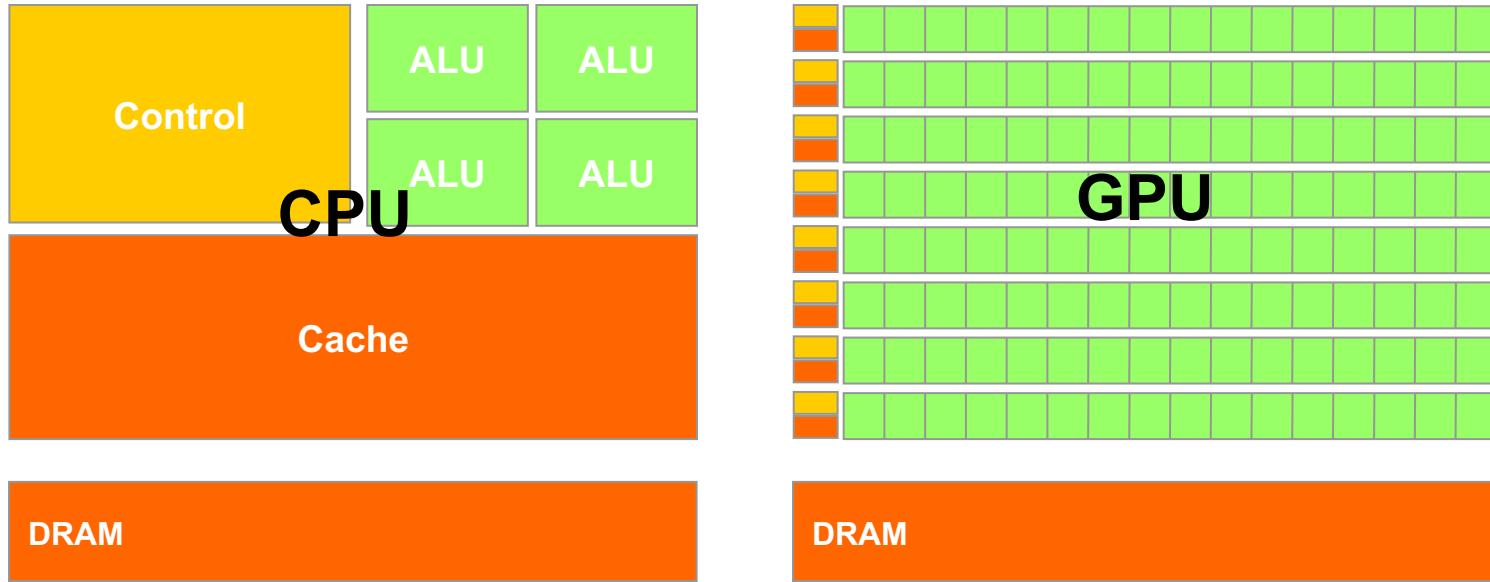
- Bioinformatica
- Calcul financiar
- Deep learning
- Molecular dynamics simulation
- Video and audio coding and manipulation
- 3D imaging and visualization
- Consumer game physics
- virtual reality products
- ...



# 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



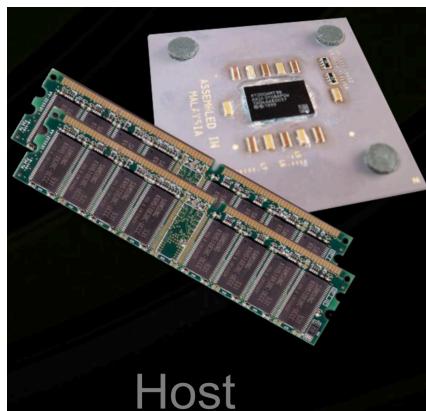
# Terminologie

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

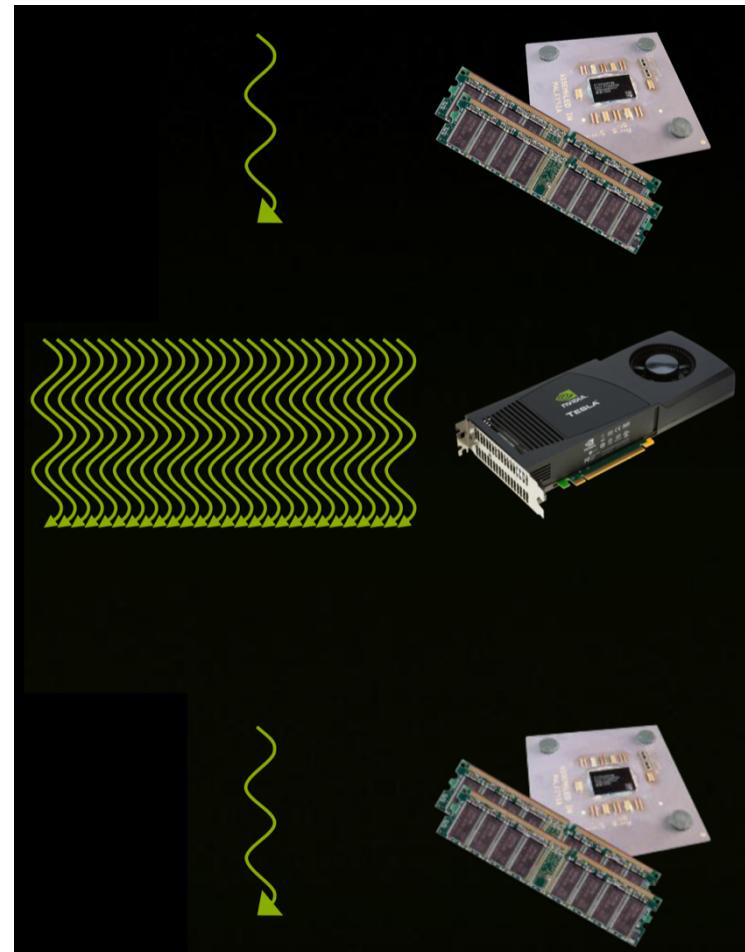
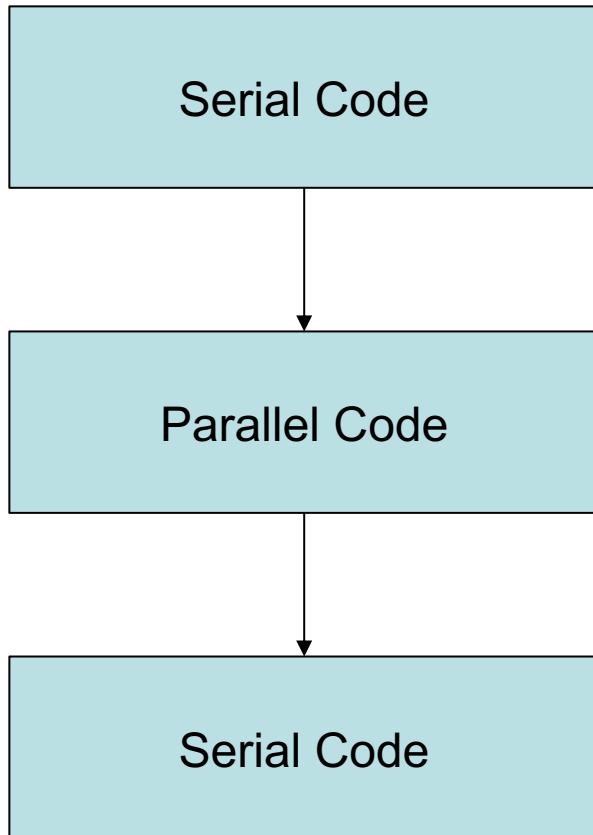


## Diferente intre threadurile GPU si CPU

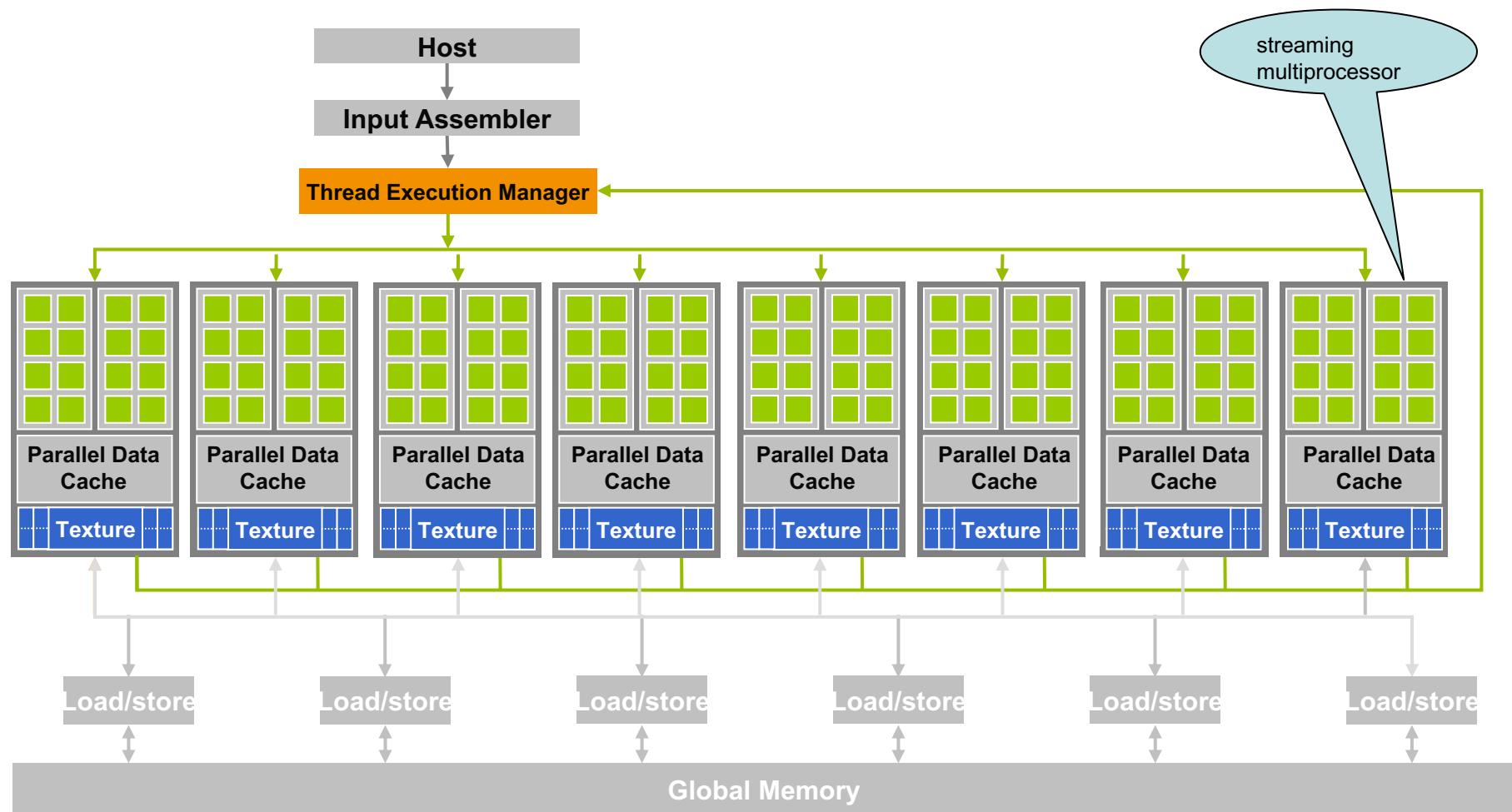
- GPU threads are extremely lightweight
  - Very little creation overhead
- GPU needs 1000s of threads for full efficiency
  - Multi-core CPU needs only a few



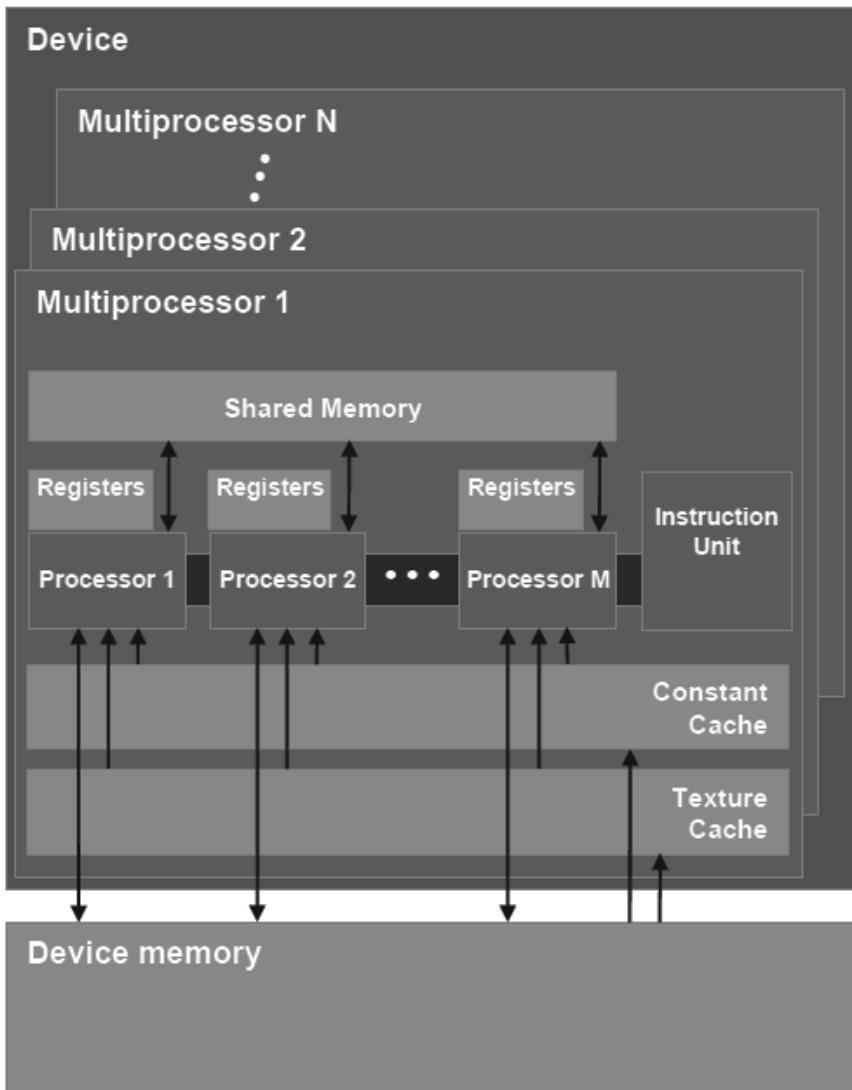
# Heterogeneous Computing



# Architecture of a CUDA-capable GPU



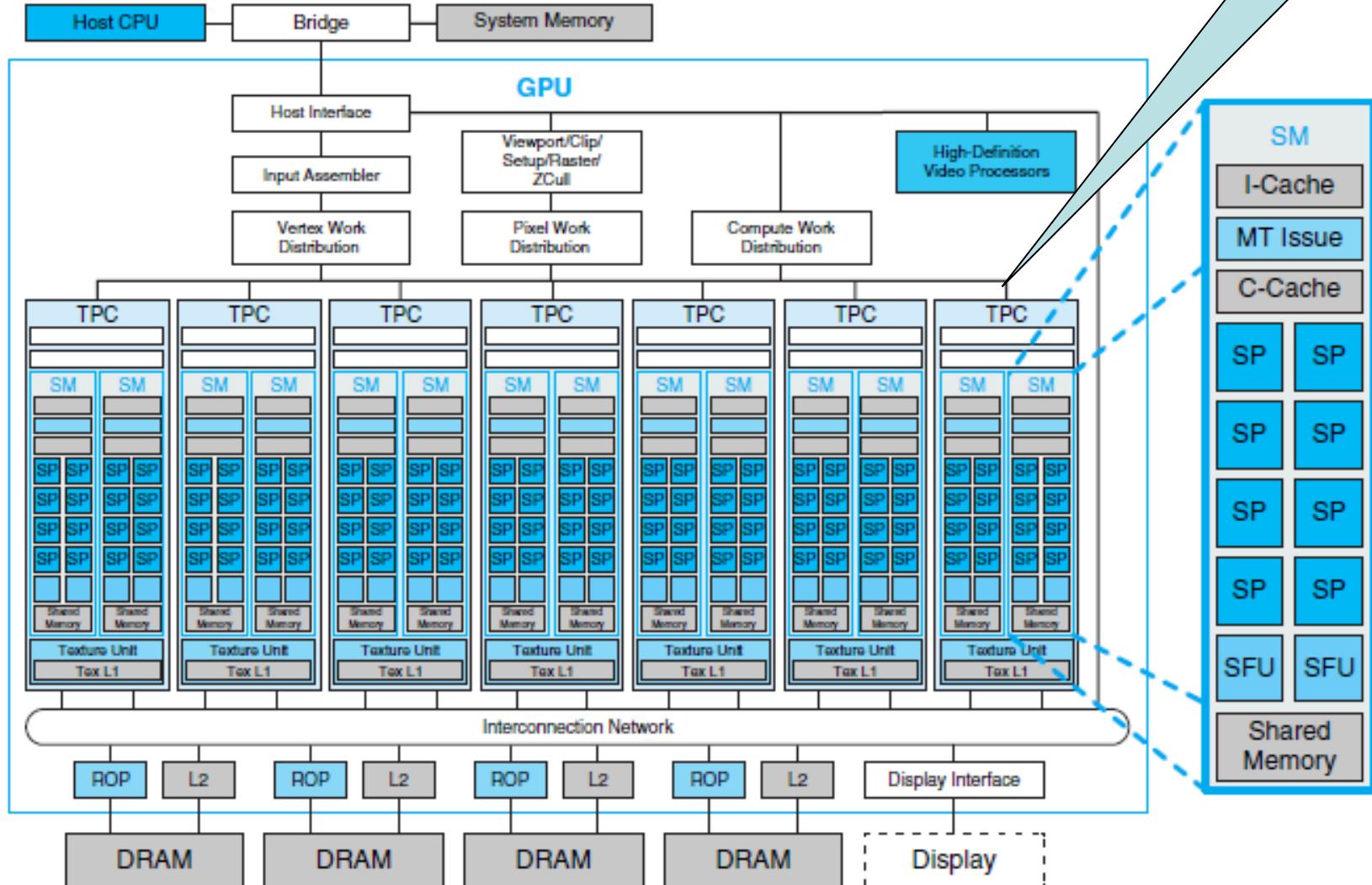
# GPU device

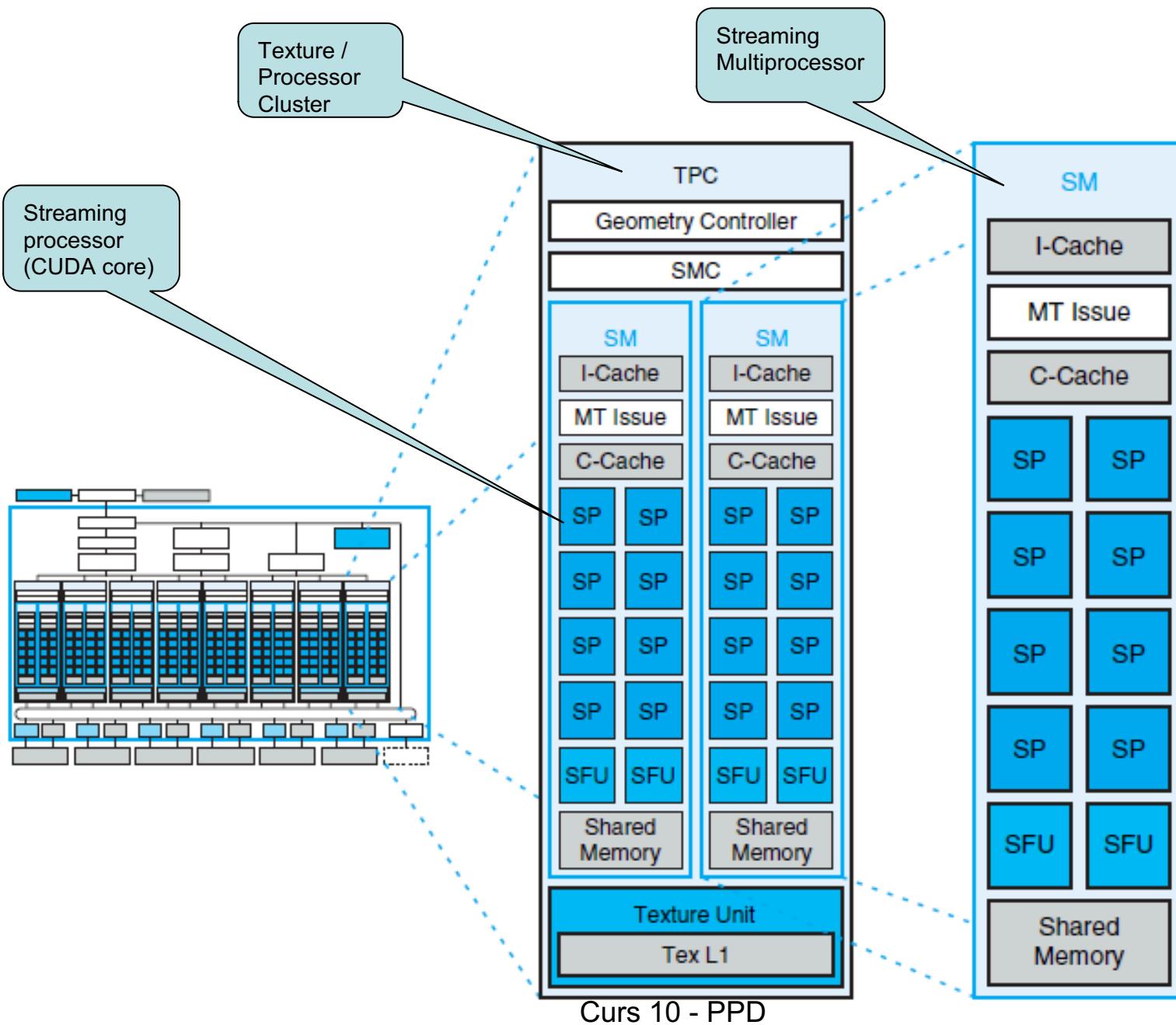


- Global memory
- Streaming Multiprocessors (SM) where each SM has:
  - Control units
  - Registers
  - Execution pipelines
  - Caches

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

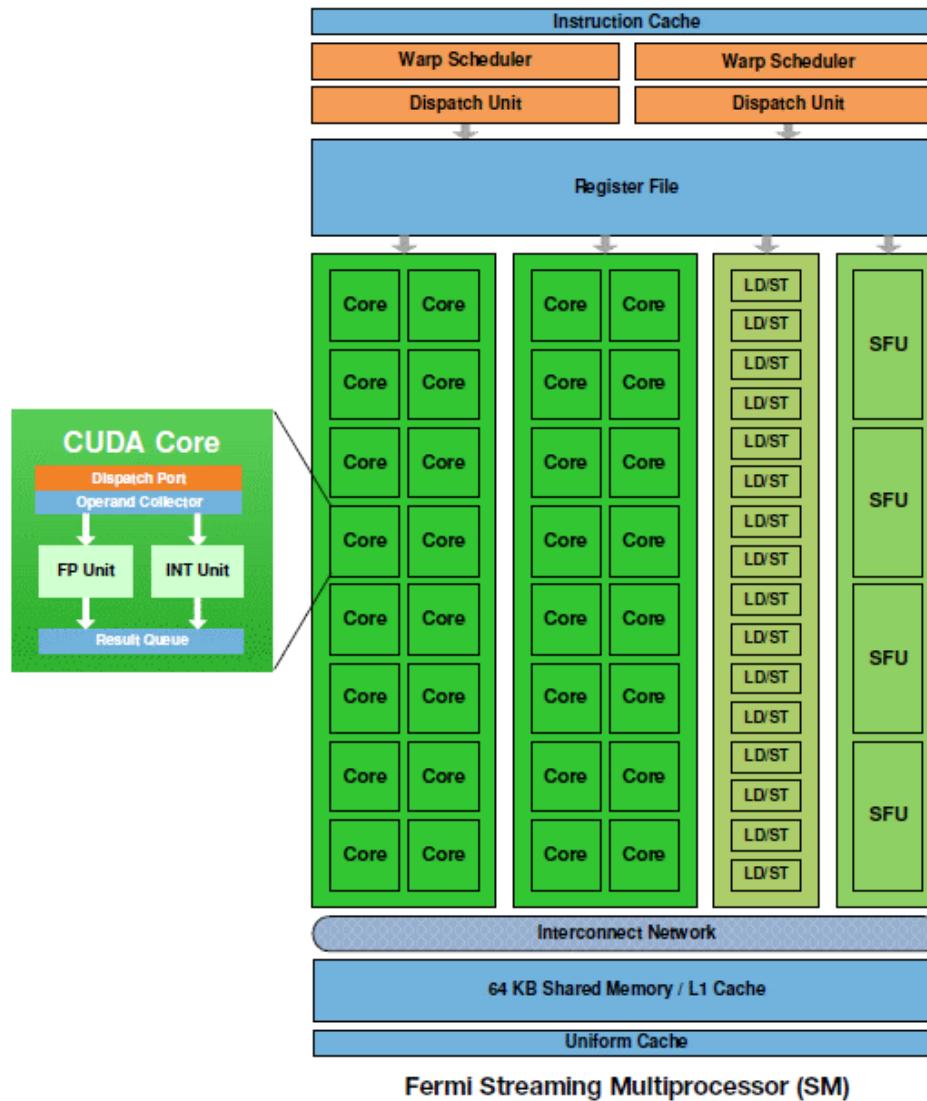
# GeForce 8800 Architecture





In Fermi architecture,  
a SM is made up of two SIMD  
16-way units.

each SIMD 16-way has 16 SPs  
=> a SM in Fermi has 32 SPs or  
32 CUDA cores



# 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];
// a global variable in the GPU, not the CPU.

__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);
```

- **Runtime API**

- **Memory, symbol, execution management**

- **Function launch**

# CUDA Function Declarations

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

- `__global__` defines a kernel function

**!!! Must return `void`**

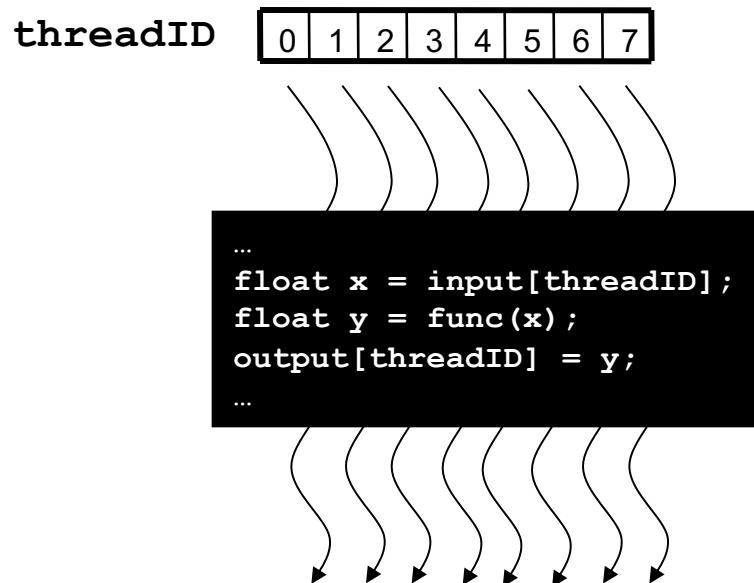
`__device__` functions cannot have their address taken

# CUDA Function Declarations

- For functions executed on the device:
  - *No recursion*
  - *No static variable declarations inside the function*
  - *No variable number of arguments*

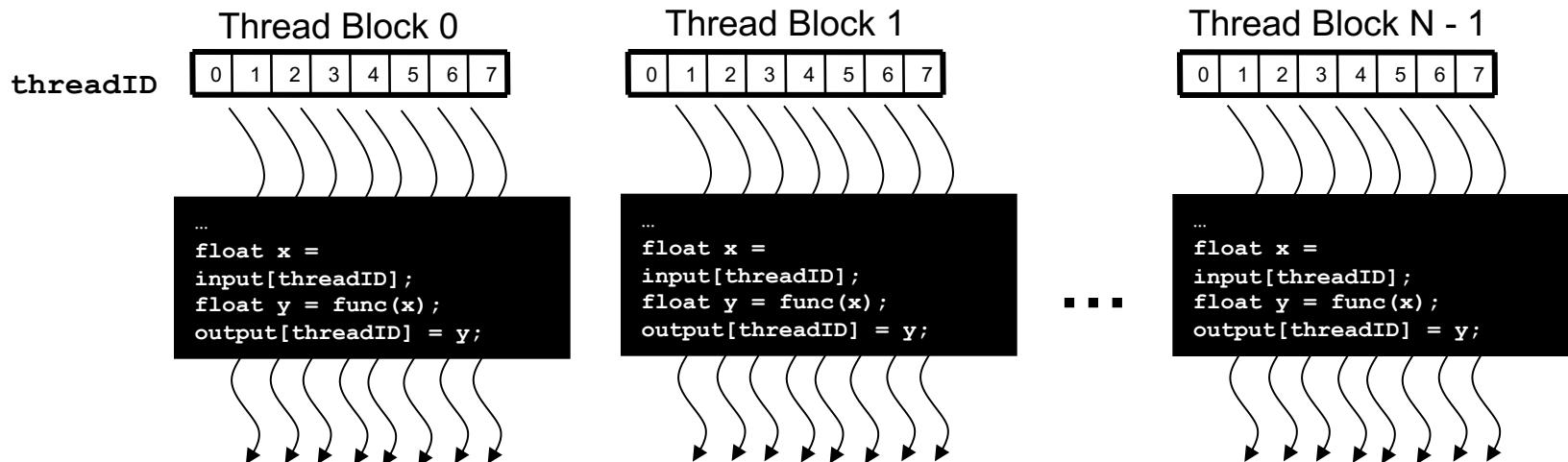
# Arrays of Parallel Threads

- 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**



# 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 not to 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;
    // init h_a and 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);
//use h_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

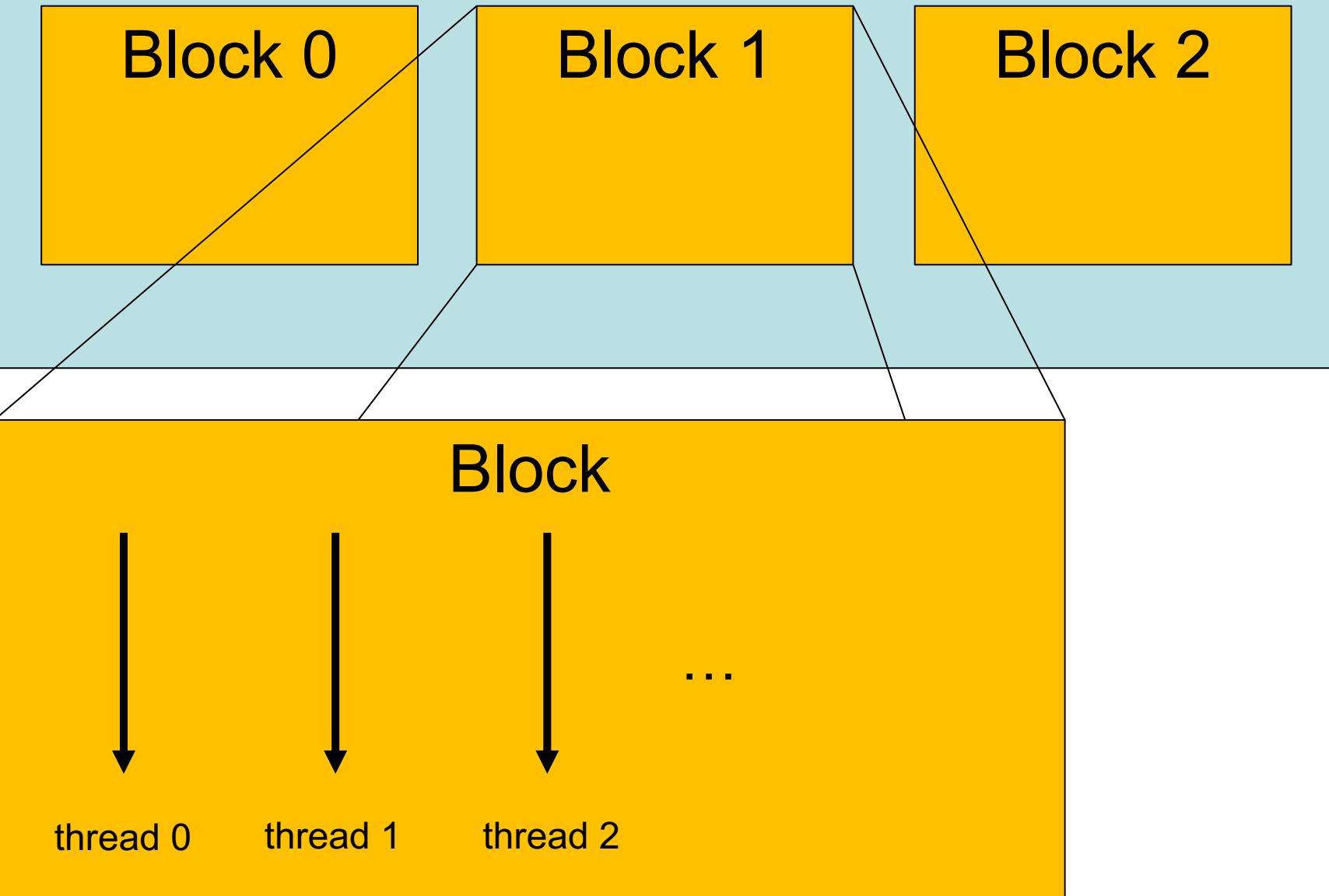
↑  
Cate blocuri si cate  
threaduri per bloc

↑  
parametrii

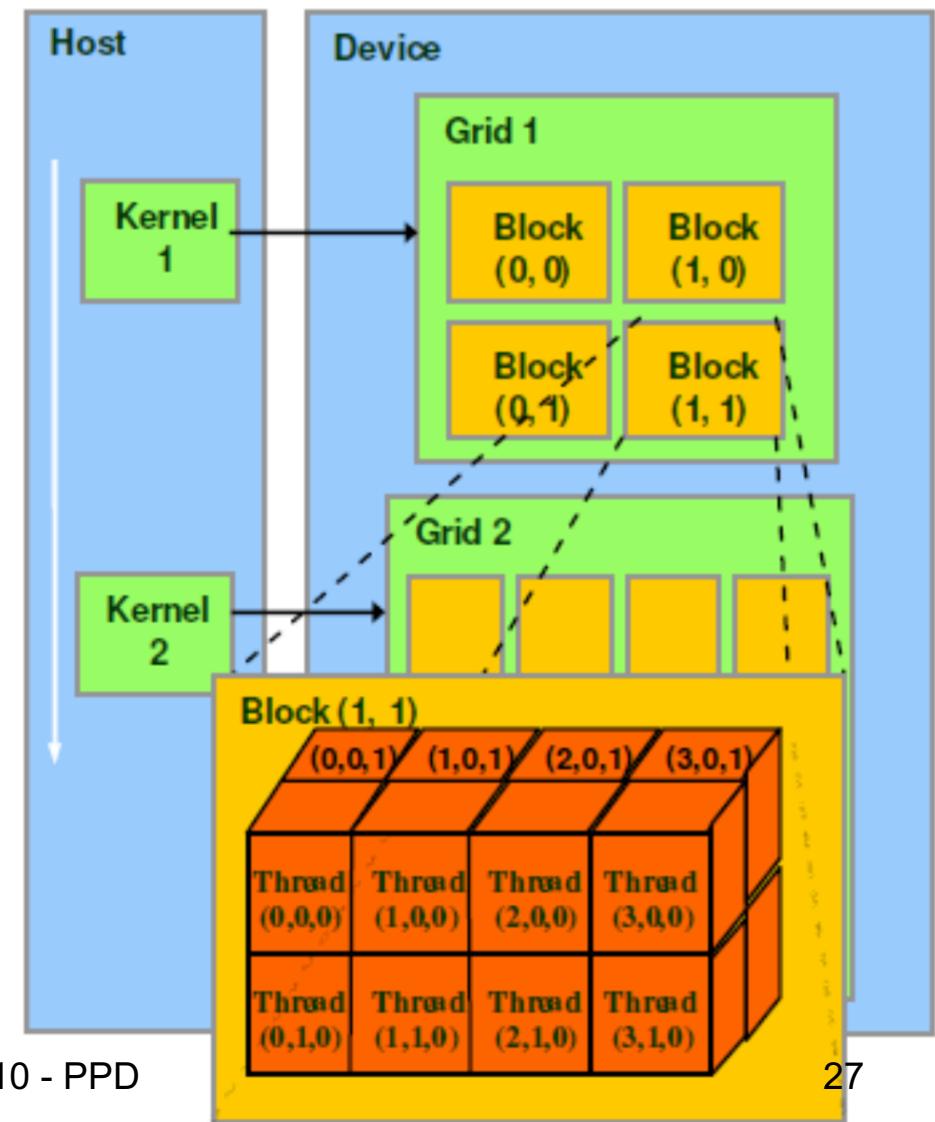
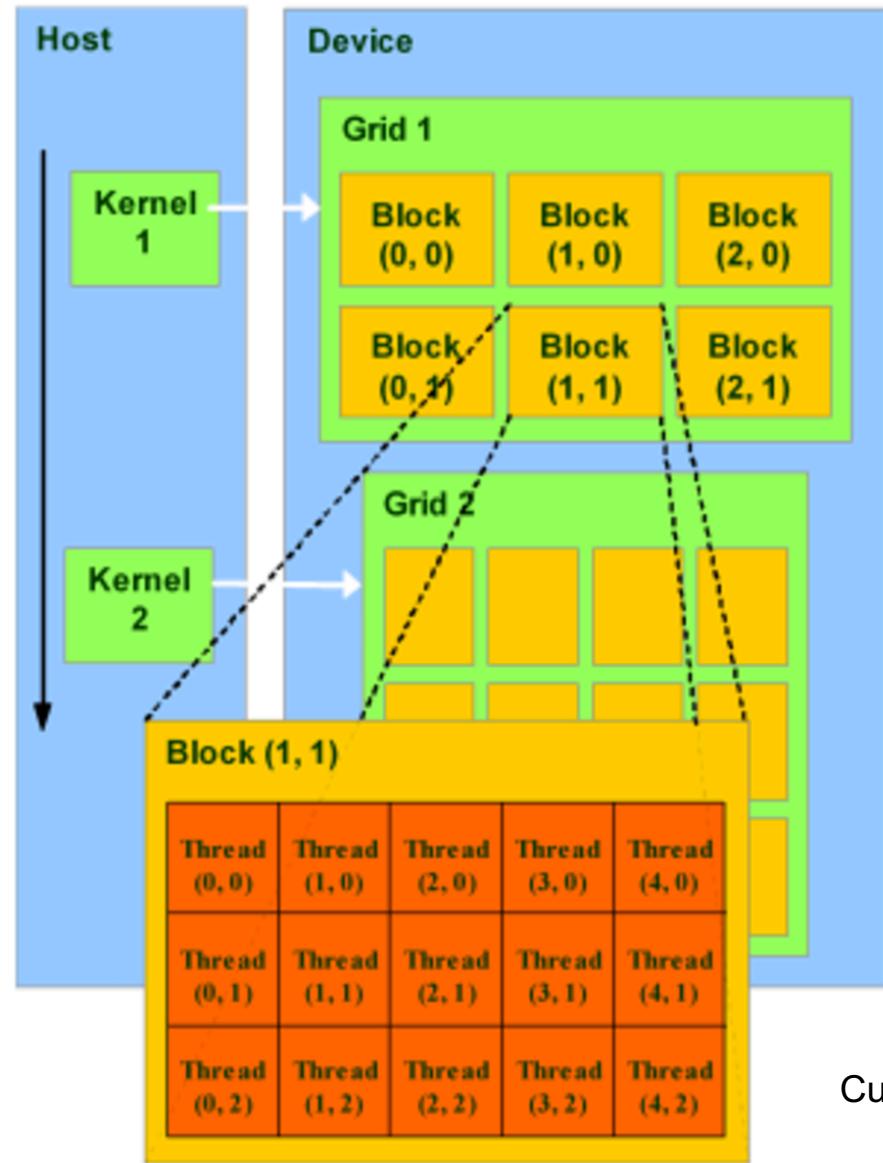
# Thread, Block, Grid

- 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

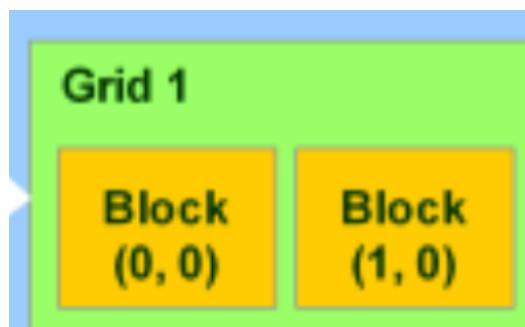
- `blockIdx.x`, `blockIdx.y`, `blockIdx.z` are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.
- `threadIdx.x`, `threadIdx.y`, `threadIdx.z` are built-in variables that return the thread ID in the x-axis, y-axis, and 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)):  
 $\text{idx} = 0 + 1 * 4 = 4$



Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)

# Exemplul 2

Global Thread ID

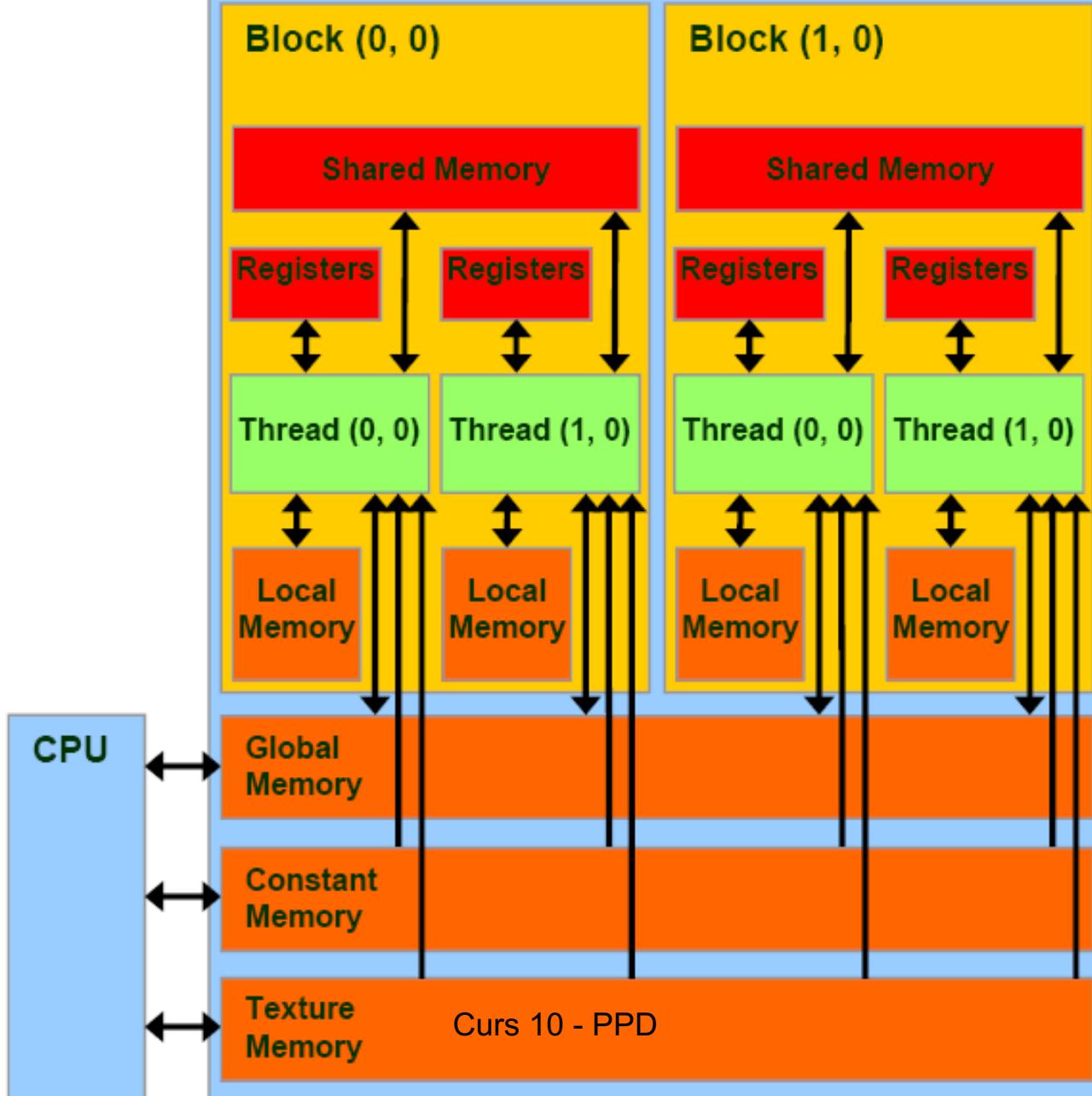
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
threadIdx.x								threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2								blockIdx.x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

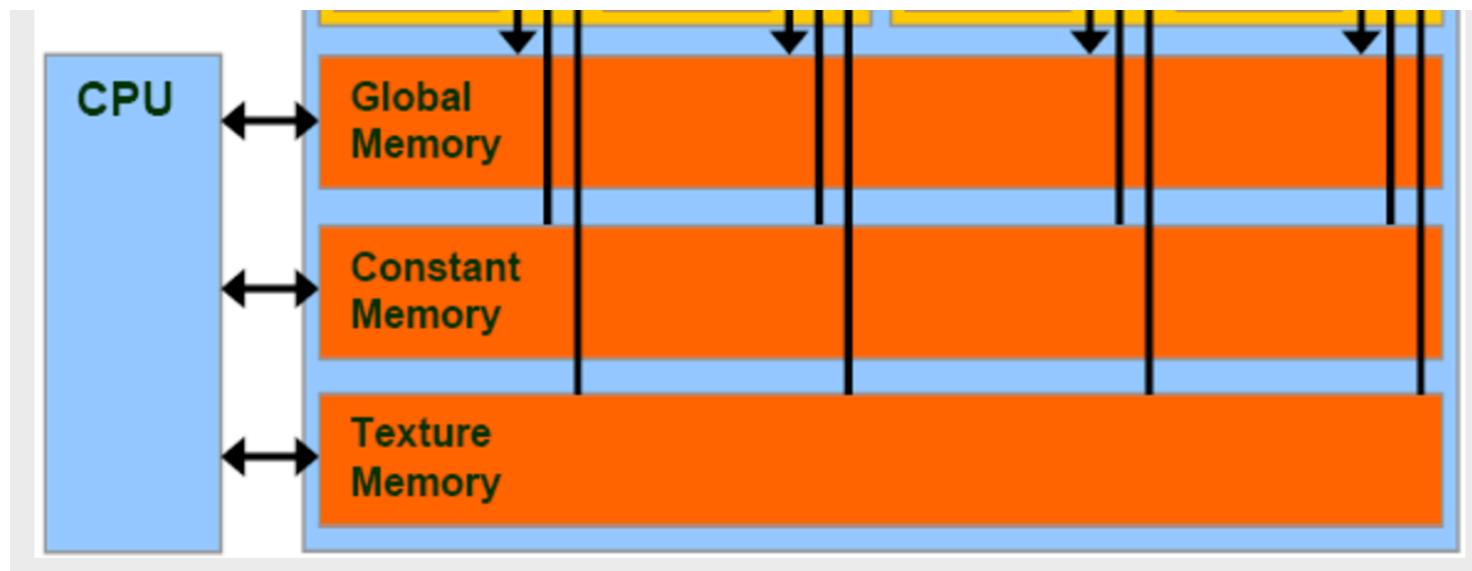
- CUDA foloseste 5 tipuri de memorie fiecare cu proprietati diferite
- Proprietati:
  - Size
  - Access speed
  - Read/write, read only

## GPU Grid



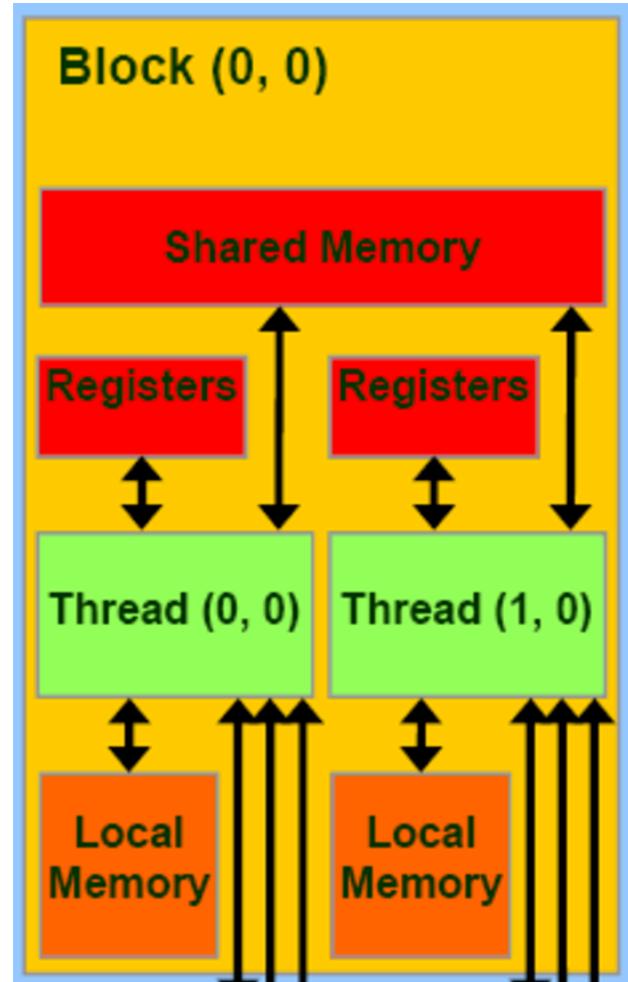
# Memory Types

- 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

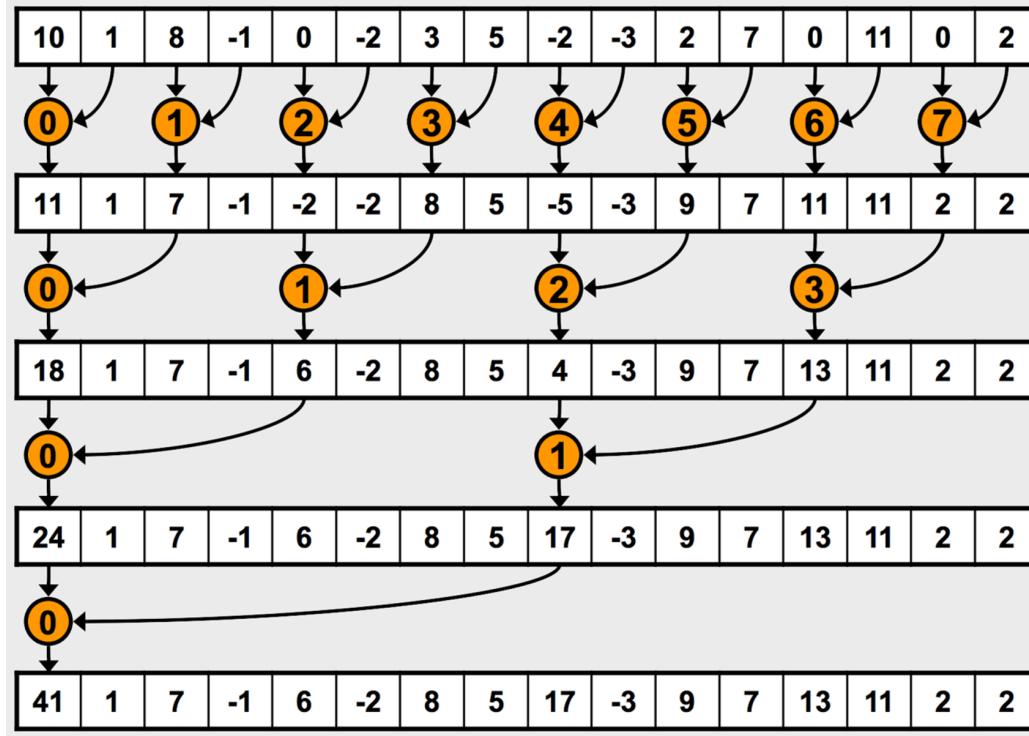
- Shared memory is very fast, but usually only has 49KB (can be configured to 64KB).
- Actually, shared memory is the same as “L1 cache” of CPU, but controllable by user.
- One block has one shared memory, that’s one reason why we manage the threads in grid and block way!

# CUDA Memories characteristics

To summarize

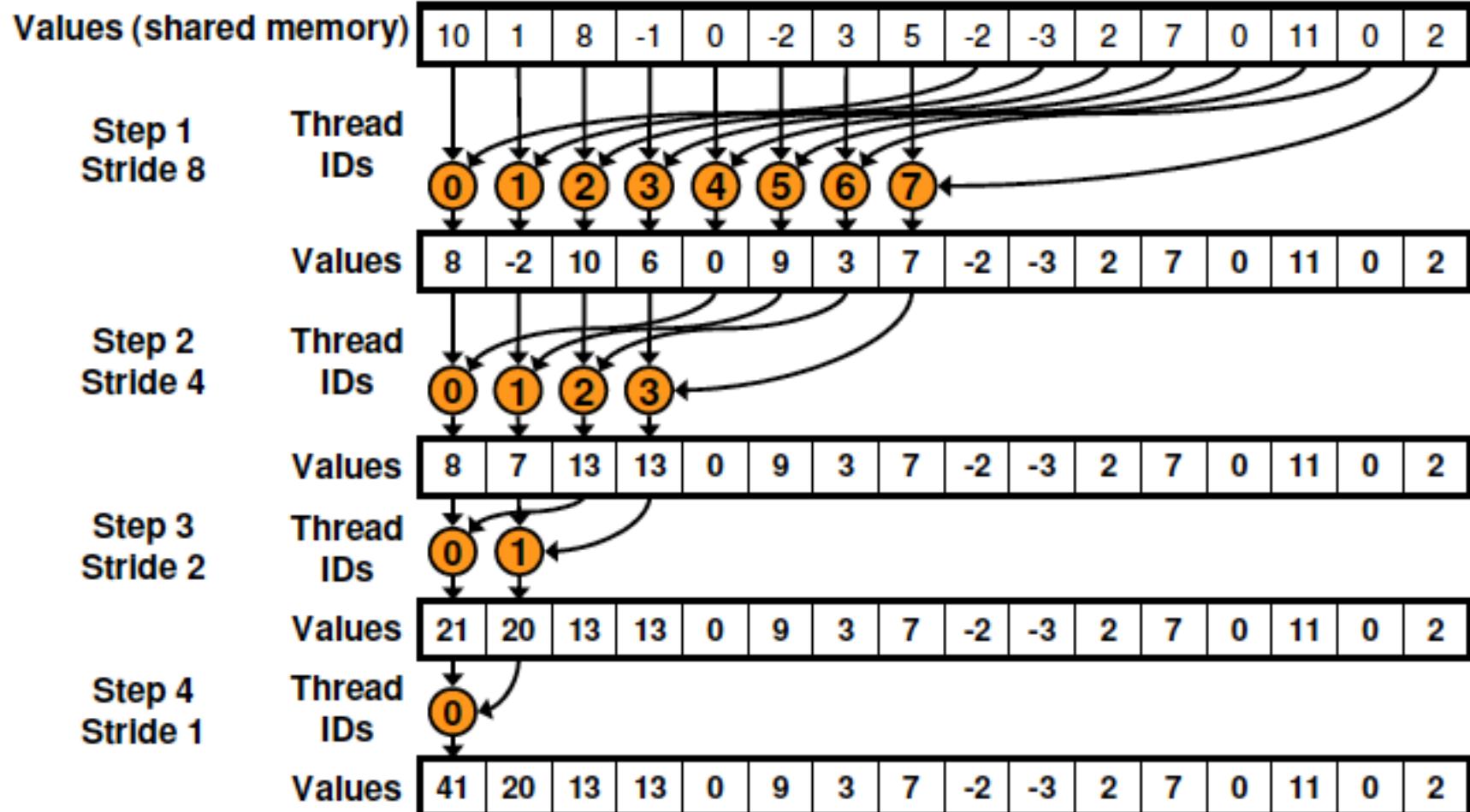
Registers	Per thread	Read-Write	
Local memory	Per thread	Read-Write	
Shared memory	Per block	Read-Write	For sharing data within a block
Global memory	Per grid	Read-Write	Not cached
Constant memory	Per grid	Read-only	Cached
Texture memory	Per grid	Read-only	Spatially cached

# Exemplu: Reduction



- !!! `__syncthreads()` is needed

# O alta varianta... dar similară *(better for CUDA)*



```

#define BLOCK_SIZE 512 // can be changed
#define NUM_OF_ELEMS 1024// can be changed
__global__ void total(float * input, float * output, int len) {
    // Load a segment of the input vector into shared memory
    __shared__ float partialSum[2*BLOCK_SIZE];
    int globalThreadId = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int t = threadIdx.x;
    unsigned int start = 2*blockIdx.x*blockDim.x;
    if ((start + t) < len) {    partialSum[t] = input[start + t];  }
    else {    partialSum[t] = 0.0;  }
    if ((start + blockDim.x + t) < len) {    partialSum[blockDim.x + t] = input[start + blockDim.x + t];  }
    else {    partialSum[blockDim.x + t] = 0.0;  }
    // Traverse reduction tree
    for (unsigned int stride = blockDim.x; stride > 0; stride /= 2)
    {
        __syncthreads();
        if (t < stride)    partialSum[t] += partialSum[t + stride];
    }
    __syncthreads();
    // Write the computed sum of the block to the output vector at correct index
    if (t == 0 && (globalThreadId*2) < len)
    {
        output[blockIdx.x] = partialSum[t];
    }
}

```

```

int main(int argc, char ** argv)
{
    int ii;
    float * hostInput; // The input 1D vector
    float * hostOutput; // The output vector (partial sums)
    float * deviceInput;
    float * deviceOutput;
    int numInputElements = NUM_OF_ELEMS; // number of elements in the input list
    int numOutputElements; // number of elements in the output list
    hostInput = (float *) malloc(sizeof(float) * numInputElements);
    //initialization
    for (int i=0; i < NUM_OF_ELEMS; i++) {
        hostInput[i] = i; // set the input values
    }
    numOutputElements = numInputElements / (BLOCK_SIZE<<1);
    if (numInputElements % (BLOCK_SIZE<<1)) { numOutputElements++; }
    hostOutput = (float*) malloc(numOutputElements * sizeof(float));
    //Allocate GPU memory
    cudaMalloc((void **) &deviceInput, numInputElements * sizeof(float));
    cudaMalloc((void **) &deviceOutput, numOutputElements * sizeof(float));
    // Copy memory to the GPU
    cudaMemcpy(deviceInput, hostInput, numInputElements * sizeof(float), cudaMemcpyHostToDevice);
}

```

```

// Initialize the grid and block dimensions here
dim3 DimGrid( numOutputElements, 1, 1); //numOutputElements = no of blocks!
dim3 DimBlock(BLOCK_SIZE, 1, 1);
//each block compute a local sum - results are stored into deviceOutput[numOutputElements]
//*********************************************************************
// Launch the GPU Kernel here
total<<<DimGrid, DimBlock>>>(deviceInput, deviceOutput, numInputElements);
//*********************************************************************
// Copy the GPU memory back to the CPU here
cudaMemcpy(hostOutput, deviceOutput, numOutputElements * sizeof(float), cudaMemcpyDeviceToHost);
/* Reduce output vector on the host*/
for (ii = 1; ii < numOutputElements; ii++) {
    hostOutput[0] += hostOutput[ii];
}
printf("Reduced Sum from GPU = %f\n", hostOutput[0]);
// Free the GPU memory here
cudaFree(deviceInput);
cudaFree(deviceOutput);
free(hostInput);
free(hostOutput);
return 0;
}

```

# Coordonare -> 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.

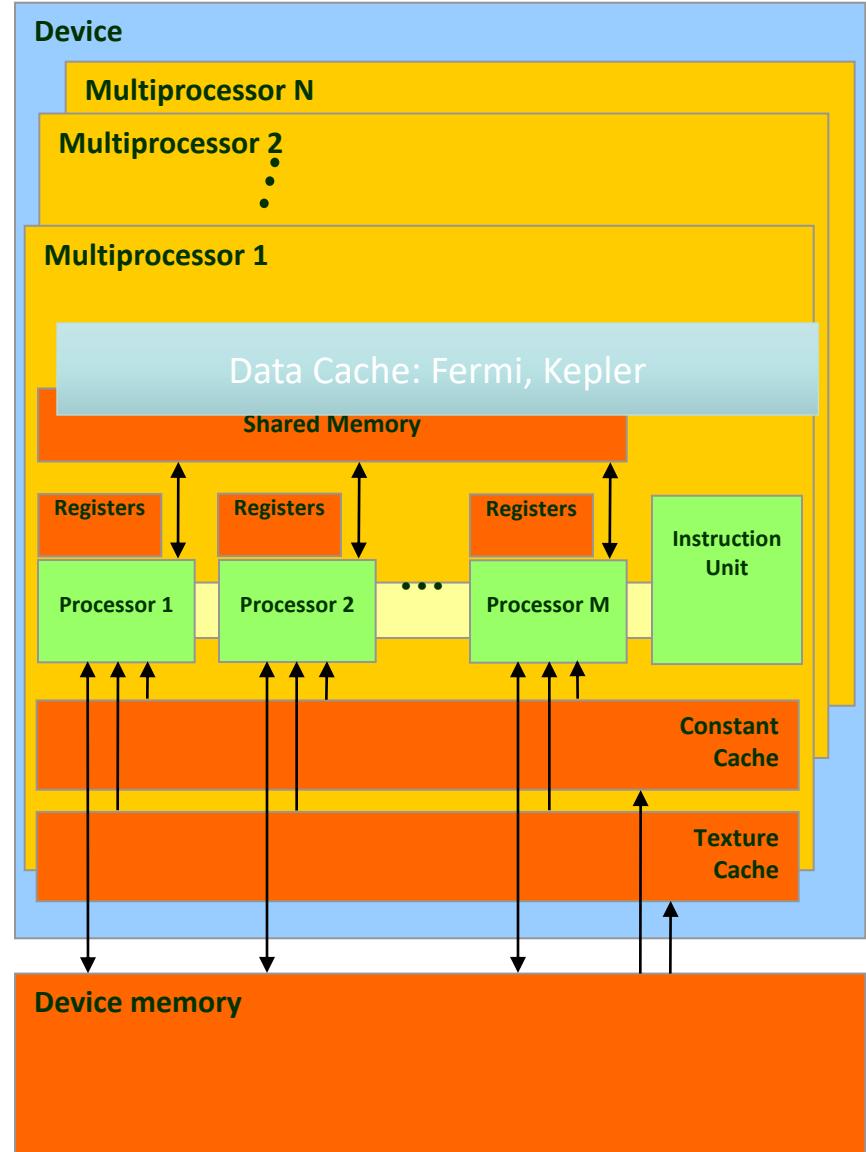
# NVIDIA GPU Execution Model

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

- Model introdus de catre Nvidia
- Combina executia de tip SIMD din interiorul unui Block (pe un SM) cu executia SPMD intre Block-uri (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 de threaduri 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 SM.**
    - De fapt threadurile ruleaza si in paralel dar si pipelined
      - fiecare SM 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, thread-urile 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 SM pot executa aceeasi instructiune in paralel.
  - Daca un thread dintr-un presupus wrap executa o instructiune diferita de celealte, atunci acel 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’
      - pe masura ce threadurile devin tot mai heterogene 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 definirea a 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 shared memory – acces rapid
- Alegerea potrivita a numarului de blocuri
- Array alignment ( alignment at 64 byte boundary)
- Continuous memory access
- Folosirea functiilor din CUDA API

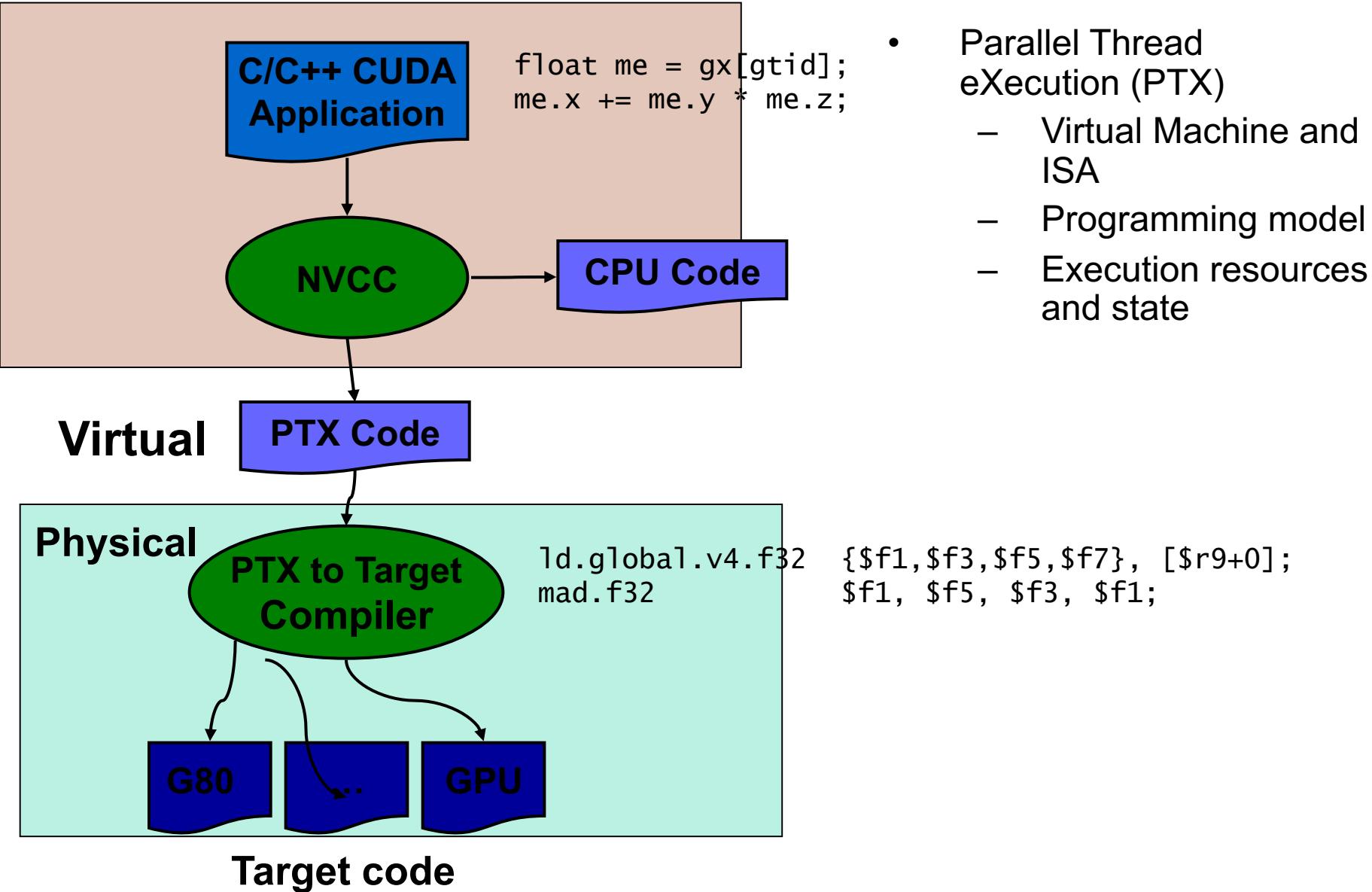
# Floating Point Operations

- 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



# 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>