CUDA

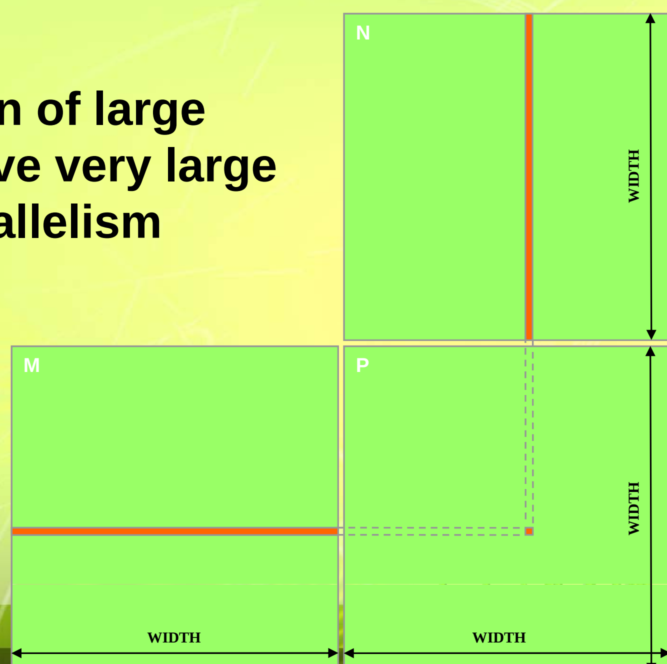
Session 2

CUDA Programming Model

Hadi Salimi and Nima Ghaemian
Distributed Systems Lab.
School of Computer Engineering,
Iran University of Science and Technology,
hsalimi@iust.ac.ir and nima@comp.iust.ac.ir

Data parallelism in matrix multiplication

- **Matrix multiplication of large dimensions can have very large amount of data parallelism**



- **Note that the dot product operations for computing different P elements can be simultaneously performed.**

CUDA program structure

CUDA Programs in general is divided into two parts:

- **The host part (CPU)**
  - **This part have a little or no data parallelism**
  - **This part is compiled with the host's standard C compilers and executed on CPU**
- **The device part (GPU)**
  - **This part has rich amounts of data parallelism**
  - **This part is compiled by the NVCC (NVIDIA C Compiler) and executed on a GPU device.**

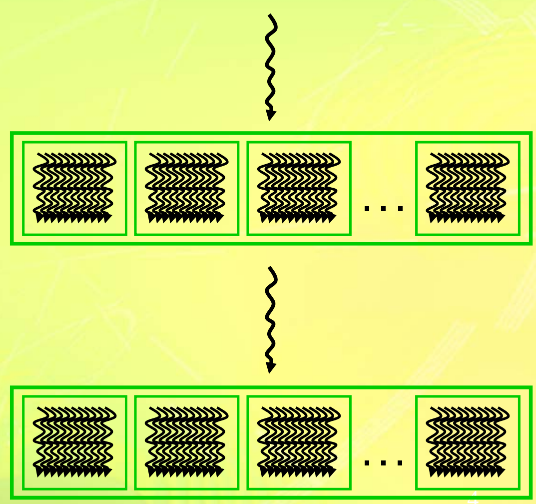
CUDA program execution

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);



Kernel functions

- The device part of the code is written using ANSI C extended with keywords for labeling data-parallel functions (*kernel functions).*

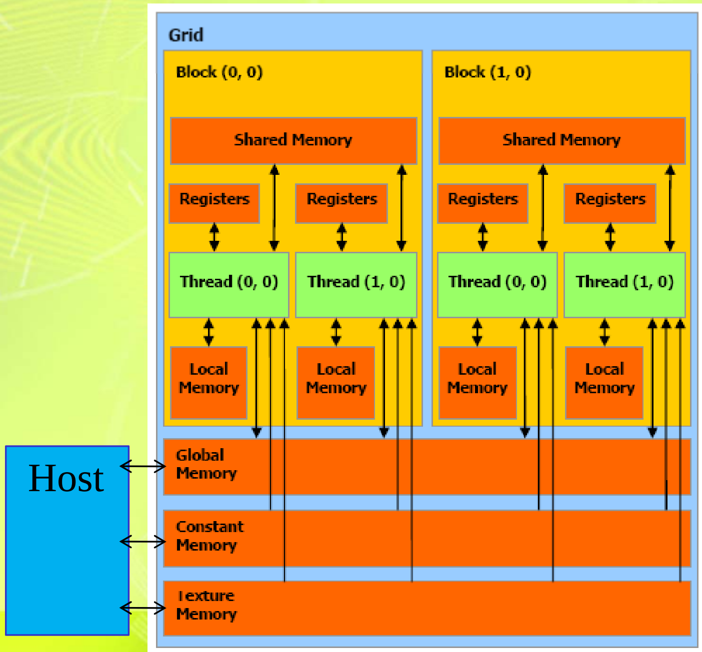
- The kernel functions generate too many threads to utilize data parallelism.

- From now on, we call “kernel functions” as “kernels” for simplification.

CUDA host code example

Int main(void) {
1. // Allocate and initialize the arrays (on the host)
// I/O to read the input arrays (on the host)
….
2. // Call a kernel function on the device
ComputeOnDevice(arrays);
3. // I/O to write the output (on the host)
// Free arrays (on the host)
…
return 0;
}

CUDA device memory model

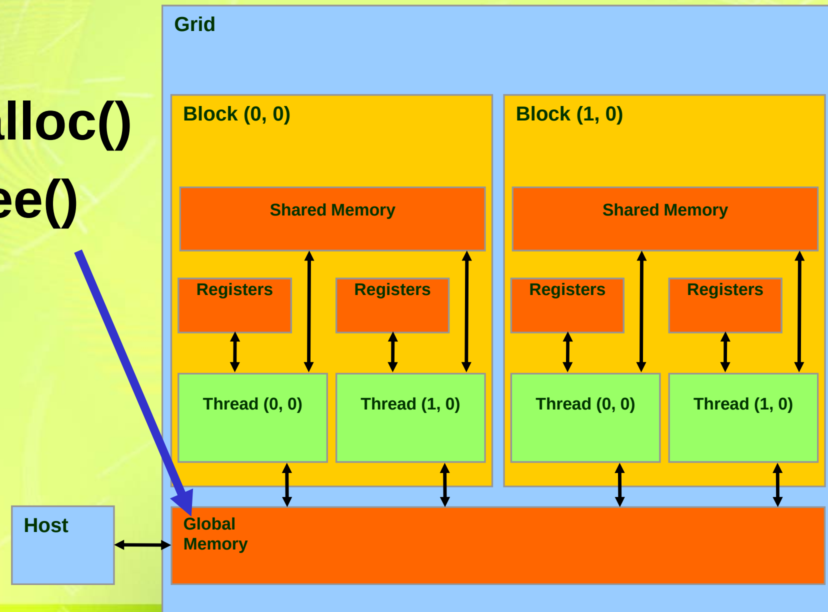


CUDA device memory model (Cont.)

- The **host code can:**
  - Read/Write per grid global, constant and texture memories (stored in DRAM)
- Each thread **(Device code) can:**
  - Read/Write per-thread registers
  - Read/Write per-thread local memory
  - Read/Write per-block shared memory
  - Read/Write per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

CUDA API functions

- cudaMalloc()
- cudaFree()



cudaMalloc()&cudaFree()

- cudaMalloc(void* p , int size): can be called from the host to allocate a piece of Global Memory for an object.
- The first parameter: the address of a pointer that needs to point to the allocated object.
- The second parameter: the size of the object to be allocated.
- cudaFree(void*): is called with the pointer to the allocated object as parameter to free the storage space from the Global Memory.

Example

Const int Dim=100; *//Dimension of the 2D array*

float *Od *//d means that the pointer is on the device*

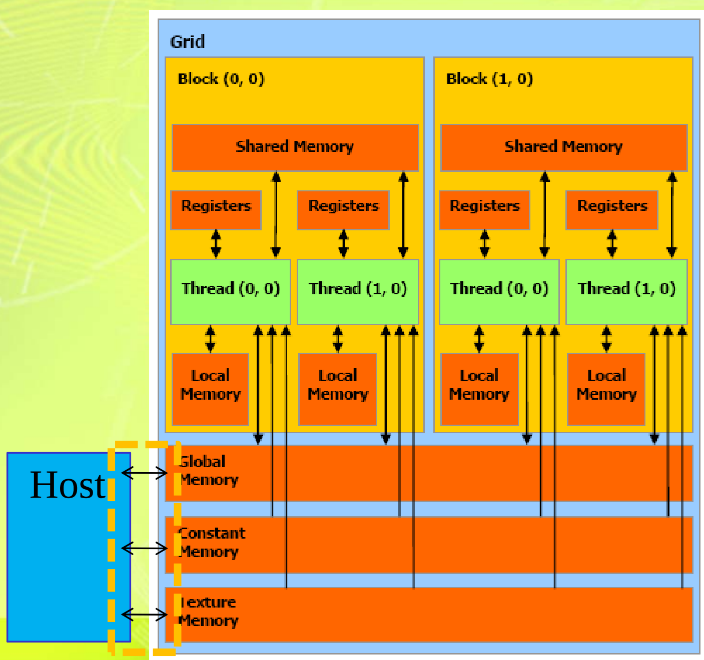
int size = Dim * Dim * sizeof(long);

cudaMalloc((void)&Od, size);**

… *//do something on the devic*

cudaFree(Od);

CUDA Host-Device Data Transfer



cudaMemcpy()

cudaMemcpy(void* , void* , size , type)

- memory data transfer
- parameters:
  - First parameter: Pointer to destination
  - Second parameter: Pointer to source
  - Third parameter: Number of bytes copied
  - Fourth parameter: Type of transfer:
    - Host to Host
    - Host to Device
    - Device to Host
    - Device to Device

Example 1

- cudaMemcpy(Od, O, size, cudaMemcpyHostToDevice);
- // Copy the size bytes of the O object on the host to the Od object on the device.

- cudaMemcpy(O, Od, size, cudaMemcpyDeviceToHost);
- //Copy the size bytes of the Od object on the device to the O object on the host.

Example 2

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
int size = Width * Width * sizeof(float);
1. // Load M and N to device memory
cudaMalloc(Md, size);
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMalloc(Nd, size);
cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
// Allocate P on the device
cudaMalloc(Pd, size);
2. // Kernel code
…
3. // Read P from the device
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
// Free device matrices
cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

SPMD

- **SPMD:**

  All threads of a parallel phase execute the same code, thus CUDA programming can be categorized as SPMD (Single-Program Multiple-Data).

- **"__global__" keyword :**
- **This keyword indicates that the function executed on the device as a kernel function and is Callable from the host only to create a grid of threads that all execute the kernel function.**

Built-in Variables

- **gridDim**

  This variable is of type dim3 and contains the dimensions of the grid.
- **blockIdx**

  This variable is of type uint3 and contains the block index within the grid.
- **blockDim**

  This variable is of type dim3 and contains the dimensions of the block.
- **threadIdx**

  This variable is of type uint3 and contains the thread indexwithin the block.

Grid and block dimension

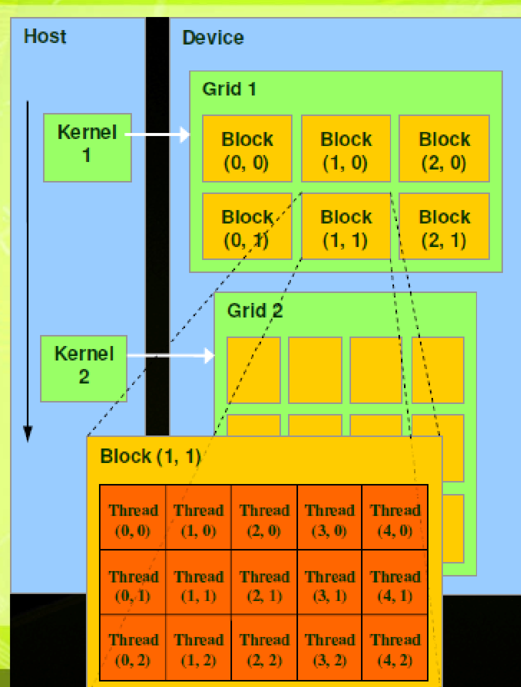
- **Each grid has one or more thread blocks.**
- **All blocks in a grid have the same number of threads that organized in the same manner.**
- **Each grid has a unique two dimensional coordinate given by the CUDA specific keywords blockIdx.x and blockIdx.y.**
- **The coordinates of threads in a block are uniquely defined by three thread indices: threadIdx.x, threadIdx.y, and threadIdx.z.**

Calling a kernel

```
// Setup the execution configuration
dim3 dimBlock(Dim1, Dim2, Dim3);
dim3 dimGrid(gridDim.x, gridDim.y);
/*The values of gridDim.x and gridDim.y can be
anywhere between 1 and 65,536.*/

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md,
Nd, Pd);
```

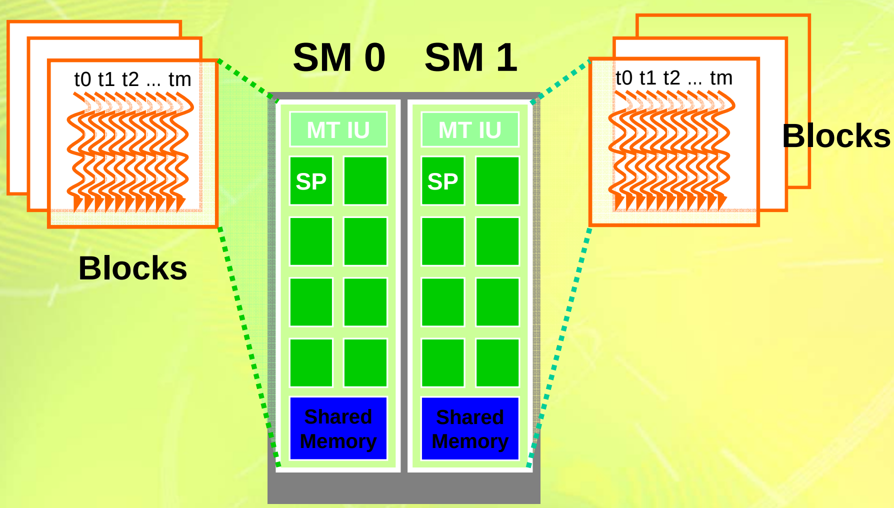
Programming Model (SPMD + SIMD)



Programming Model (SPMD + SIMD)

- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate

Thread Assignment



Each block is assigned to a SM (Streaming Multiprocessor)

Hardware Constraints

GeForce 8800 hardware constraints:

- **512 threads per thread block**
- **8 thread blocks per SM**
- **768 threads per SM > 768x16=12,288 threads totally!**
- **16,384 bytes of shared cache per SM**
- **8,192 total registers per SM (shared among all thread blocks assigned to that SM)**

Example 1

Which is false?

1. **dim3 dimBlock(32, 8, 2);**
2. **dim3 dimBlock(16, 3, 8);**
3. **dim3 dimBlock(8, 13, 4);**
4. **dim3 dimBlock(8, 9, 8);**

The answer is: 4

because the number of threads per block is greater than 512, it defines 576(8x9x8) threads per block!

Example 2

Which is false?

The threads can be assigned to each SM could be in the form of:

1. **4 blocks of 156 threads each**
2. **3 blocks of 256 threads each**
3. **8 blocks of 228 threads each**
4. **12 blocks of 64 threads each**

The answer is: 4

because the number of blocks per SM is greater than 8!

Web Resources

- **CUDA ZONE: http://www.nvidia.com/object/cuda_education.html**
- ***David Kirk and Wen-mei W. Hwu,* Lecture Notes of Programming Massively Parallel Processors, University of Illinois, Urbana-Champaign, *2009***
- **Rob Farber, "CUDA, Supercomputing for the Masses", *Dr. Dobb's Journal, can be found at:***

  http://www.ddj.com/hpc-high-performance-computing/

Acknowledgment

- **Special Thanks to Mr. Ebrahim Khademi for his technical comments.**

Any Questions?

Any Question?