Introduction to The CUDA Programming Model

Sarvenaz Choobdar

Introduction

- GPU: A Highly Multithreaded Coprocessor specialized for SIMD.
- CUDA Programming is designed to exploits this capability when an application
  - Has to be executed many times
  - Can be isolated as a function
  - Works independently on different data
- Applications :
  - Image processing, computational engineering, matrix algebra, convolution, correlation, sorting

CUDA Program Structure

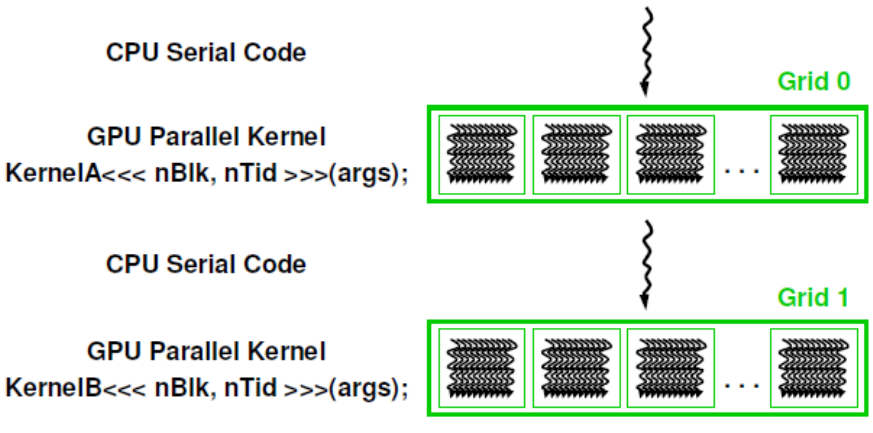
A CUDA program consists of different phases executed on

1. The host (CPU)
   - Phases which exhibit little or no data parallelism
   - C code compiled with the host's C compilers and runs as an ordinary process.
2. The device (GPU)
   - Phases which exhibit rich amount of data parallelism
   - C extended with keywords for labeling data-parallel functions, called kernels, and their associated data structures.
   - GPU code is further compiled by the NVCC

3

Execution of a CUDA program

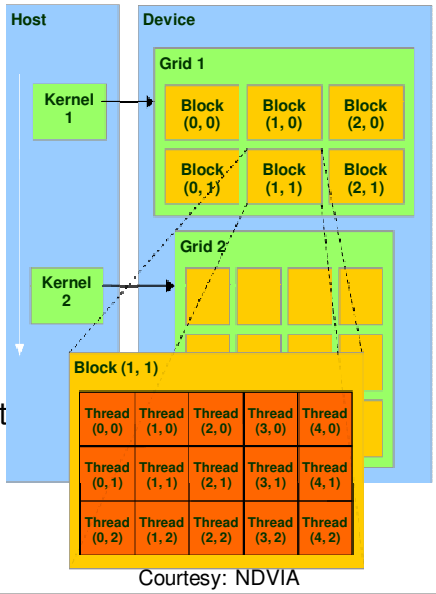
- Integrated host+device application C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code



4

Thread Batching: Grids and Blocks

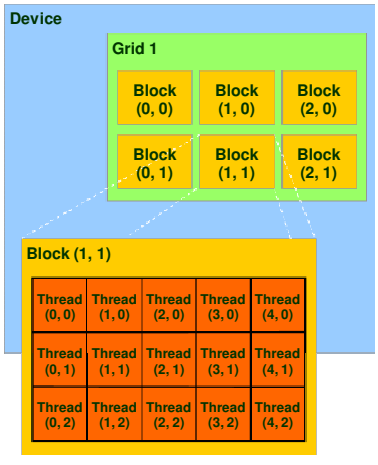
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared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



Courtesy: NDVIA

Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …

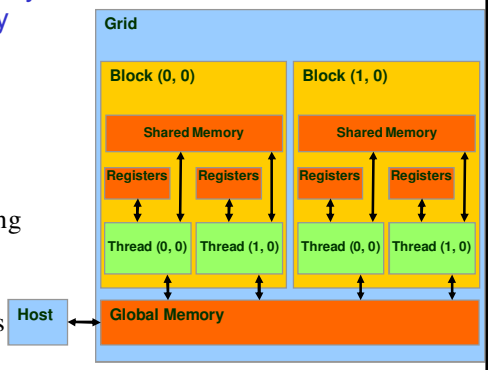


Courtesy: NDVIA

CUDA Memory Model Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access

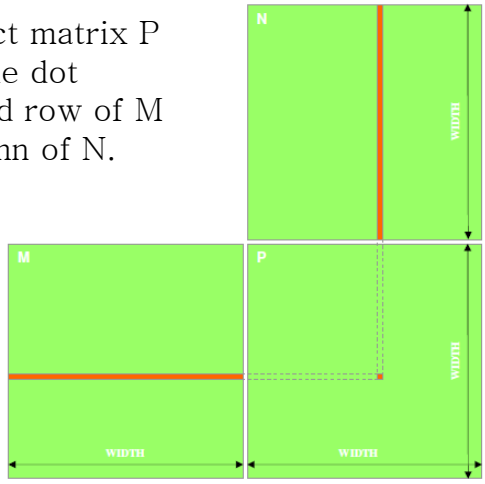


7

Data Parallelism Example: Matrix Matrix Multiply (P=P+M*N)

An element of the product matrix P is generated by taking the dot product of the highlighted row of M and the highlighted column of N.

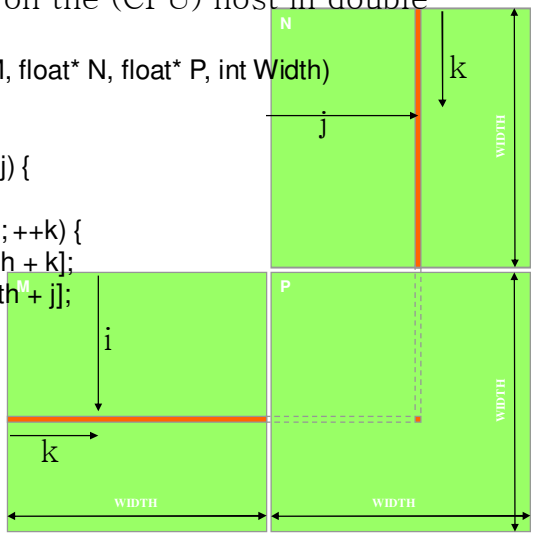
The dot product operations for computing *different (all)* P elements can be simultaneously performed.



8

Matrix Multiplication: A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Matrix Matrix Multiply Example (cont.)

- 1,000 X 1,000 matrices
  - 1,000,000 independent dot products
  - Each dot product involves 1,000 multiply and 1,000 accumulate arithmetic operations
- One thread could be used to compute one P element which would generate 1,000,000 threads
  - Very large amount of data parallelism
- By executing many dot products in parallel, a GPU device can significantly accelerate the execution of the matrix multiplication

10

CUDA ***host*** code Skelton for matrix multiplication

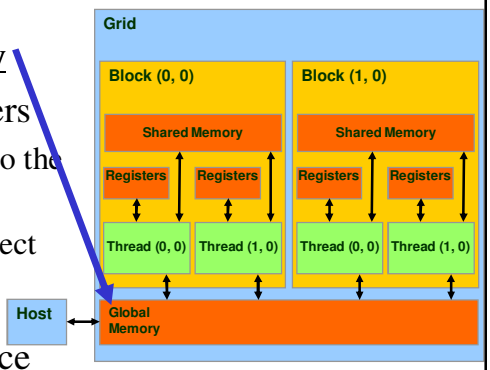
2. Executed on host and launches a kernel to perform matrix multiply on the device

```
int main(void) {
1.    // Allocate and initialize the matrices M, N, P
      // I/O to read the input matrices M and N
....

2.    // M * N on the device
      MatrixMulOnDevice(M, N, P, width);

3.  // I/O to write the output matrix P
    // Free matrices M, N, P
...
return 0;
}
```

11

CUDA Device Memory Allocation

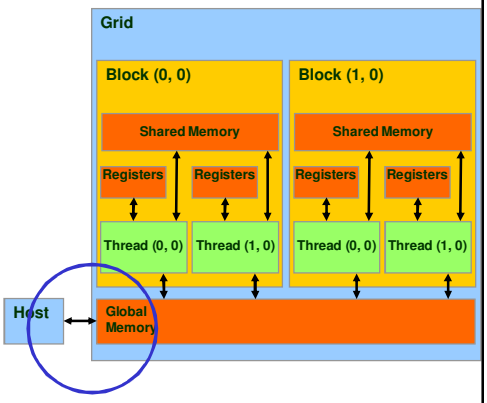
- cudaMalloc()
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



12

CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device



13

```
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 invocation code – to be shown later
    …
3.  // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }
```

14

CUDA Keywords

15

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

16

Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

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

KernelFunc<<< DimGrid, DimBlock>>>; // execution
   configuration
```

17

Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
  dim3 dimGrid(1, 1);
   dim3 dimBlock(Width, Width);


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

18

Cuda Thread ID Keywords

- All threads execute the same kernel code
  - There needs to be a mechanism to allow them to direct themselves towards the particular parts of the data structure they are designated to work on.
- Keywords “threadIdx.x” and “threadIdx.y” refer to the thread indices of a thread
  - allow a thread to access hardware registers associated with it at runtime that provides the identity to the thread.
- We will refer to a thread as Thread $_{\text{threadIdx.x, threadIdx.y}}$

19

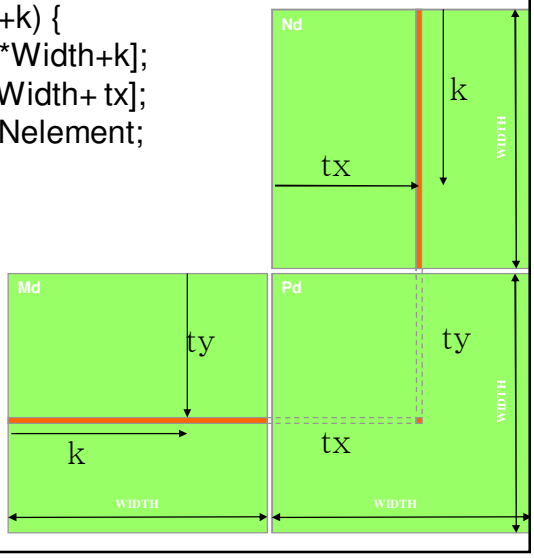
Kernel Function (details)

```
// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

20

Kernel Function (cont.)

```
for (int k = 0; k < Width; ++k) {
    float Melement = Md[ty*Width+k];
    float Nelement = Nd[k*Width+ tx];
    Pvalue += Melement * Nelement;
}

Pd[ty*Width+ tx] = Pvalue;
}
```




```
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 invocation code – to be shown later
    …
3.  // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
    }
```

Some Useful Information

23

where do threads, blocks and grids actually get executed?

- With respect to Nvidia's G80 GPU chip:
  - Grid → GPU: An entire grid is handled by a single GPU chip.
  - Block → MP: The GPU chip is organized as a collection of multiprocessors (MPs), with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple MPs.
  - Thread → SP: Each MP is further divided into a number of stream processors (SPs), with each SP handling one or more threads in a block.

Installation

- Download CUDA (http://www.nvidia.com/object/cuda_get.html)
  - CUDA driver
  - CUDA toolkit
  - CUDA SDK (optional)
-

Run

- Linux:
  - compile the cpp files this way:
    g++ -c *.cpp
  - Then compile the cu files:
    nvcc -c *.cu
  - Finally, link these together and make the final executable:
    g++ -o runme *.o
- Windows:
  - Visual C++ Express Edition

Useful links

- Slides of Prof. Wen-Mei Hwu of UIUC
  - http://courses.ece.uiuc.edu/ece498/al/Syllabus.html
- http://llpanorama.wordpress.com/cuda-tutorial/
- http://forums.nvidia.com/index.php?showtopic=30273

Thanks!