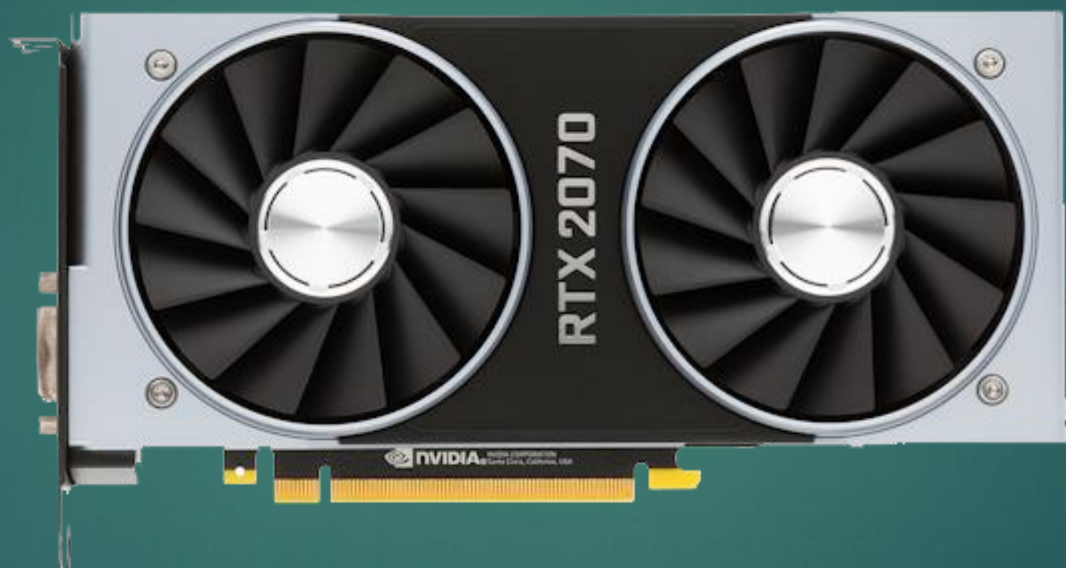
CUDA Tutorial – How to Start with CUDA?

Presentation · December 2018

DOI: 10.13140/RG.2.2.22460.49289

Ali Tourani
University of Luxembourg

CUDA Tutorial



ALI TOURANI

MSC. SOFTWARE ENGINEERING

A.TOURANI1991@GMAIL.COM

Scopes

- Introduction
- CUDA key concepts
- CUDA threads
- CUDA performance
- CUDA memories
- CUDA installation
- CUDA matrix multiplication

Introduction

- CUDA: a parallel computing platform and API model
  - Developed by NVidia
- Utilization of the power of NVidia GPUs
  - Doing graphical calculations
  - Perform general computing tasks
    - Image Processing
    - Deep Learning
    - Multiplying matrices
- Requirements:
  - C/C++/Python programming and a NVidia GPU card!

Introduction

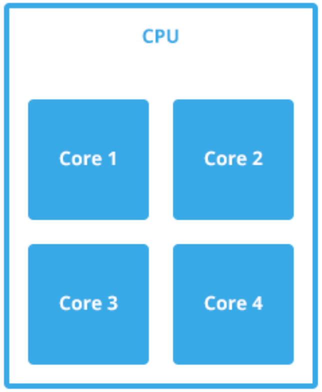
- Parallelism in the CPU
  - Instruction fetch (IF)
  - Instruction decode (ID)
  - Instruction execute (EX)
  - Memory access (MEM)
  - Register write-back (WB)

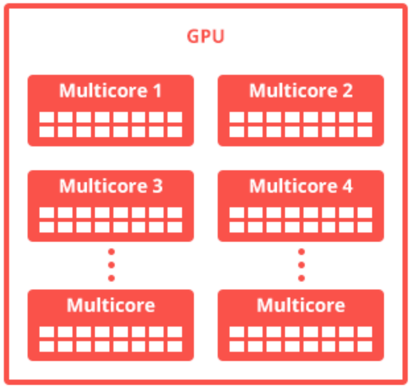
IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

- Pipelining
  - Instruction Level Parallelism (ILP)

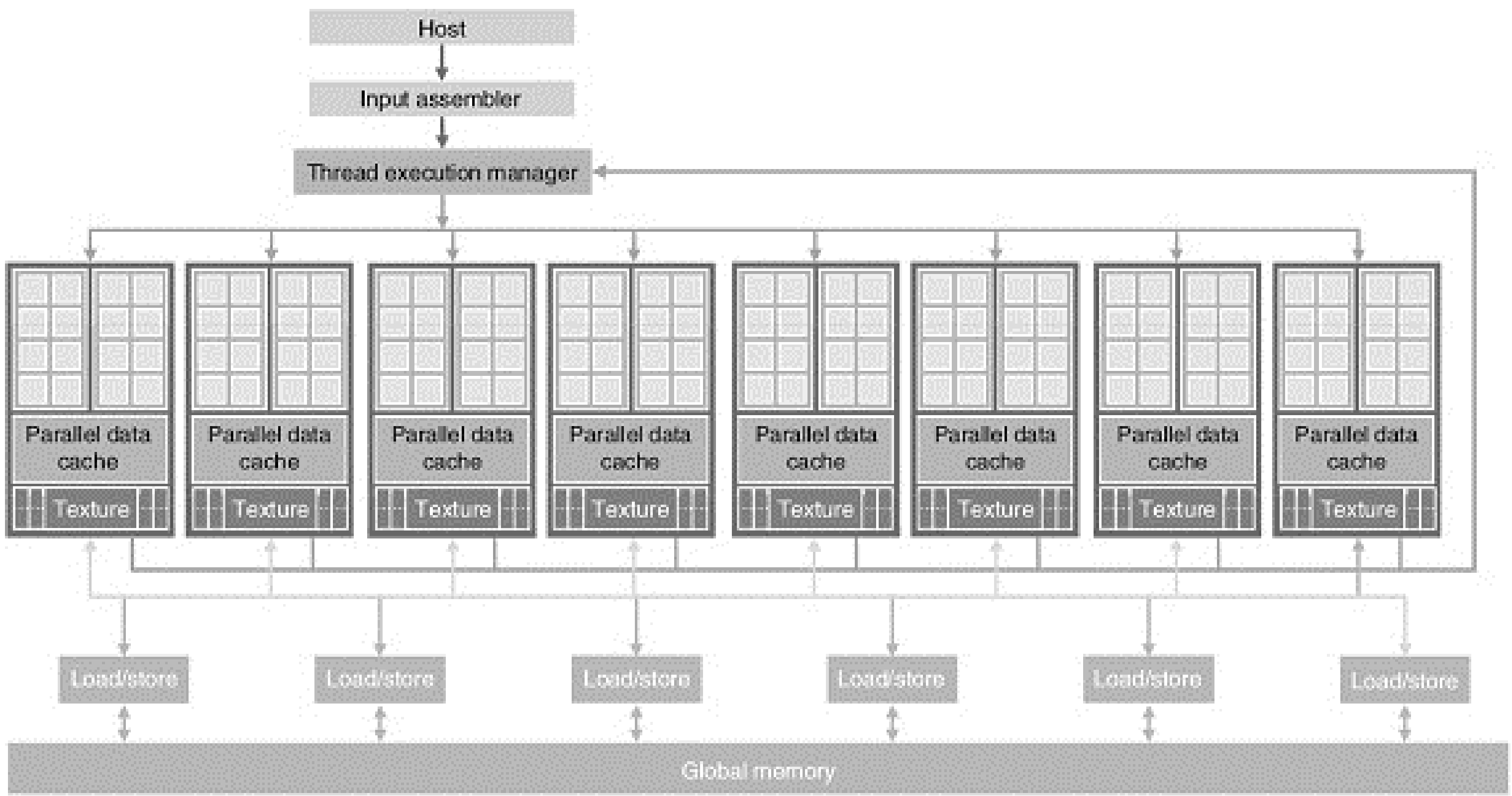
Introduction

- Parallelism in the GPU
  - Many-core processors
  - Operation of large chunks of data
  - Massively-parallel programs
  - Efficient for SPMD (Single Program, Multiple Data)
  - No virtual memory
  - No interrupts



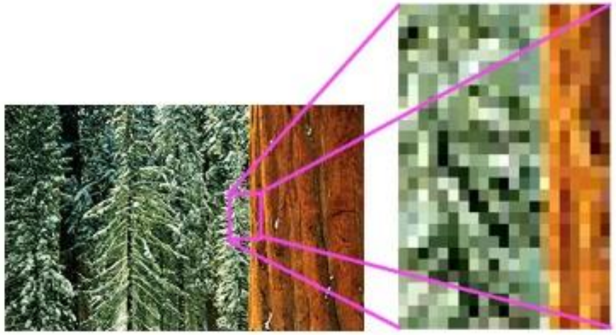


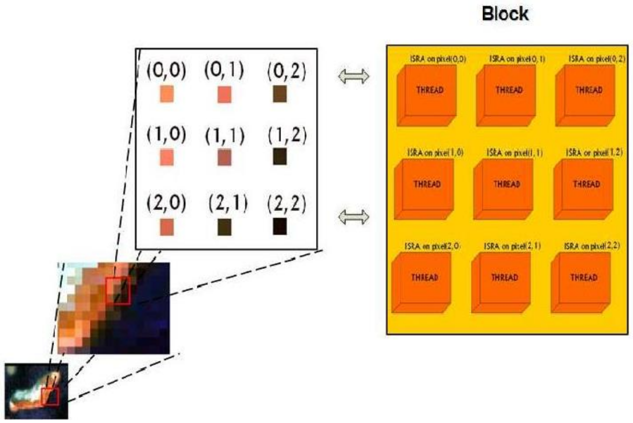
Introduction



CUDA key concepts

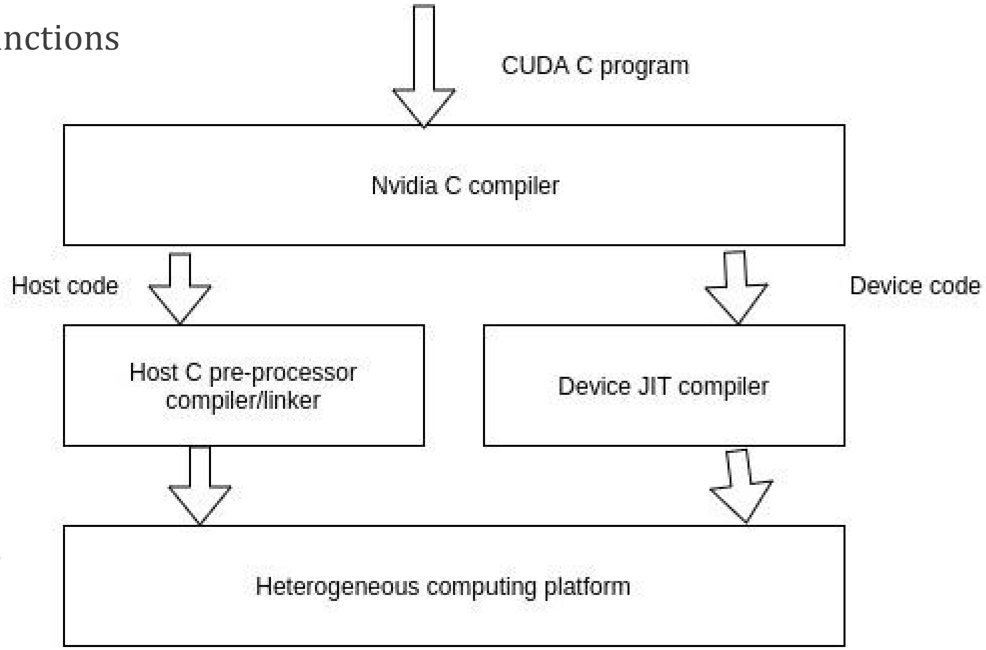
- Data Parallelism in CUDA
  - Computationally intensive applications
    - Such as rendering pixels
  - Threads (workers) as the main tools
    - Pixels to threads mapping → O(1)
    - Each thread processes one pixel





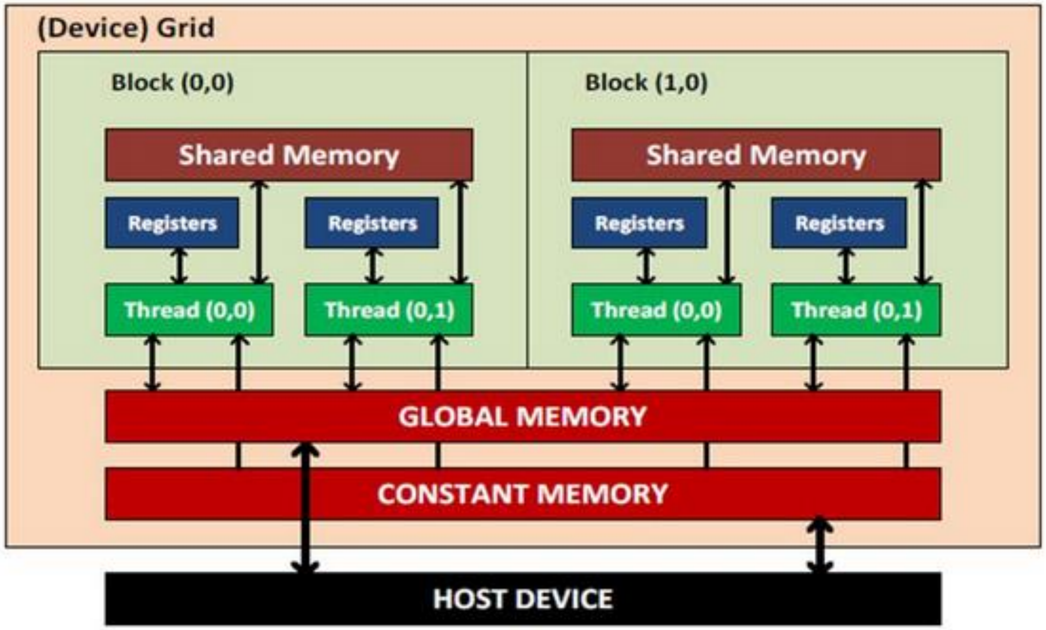
CUDA key concepts

- Structure of CUDA (for a C program)
  - NVCC (NVidia C Compiler)
    - A compiler to understand API functions
  - Host code
    - C code - run by CPU
    - Compiled by GCC
  - Device code
    - C code - run by GPU
    - Special keywords needed
    - **Kernels**: data-parallel functions



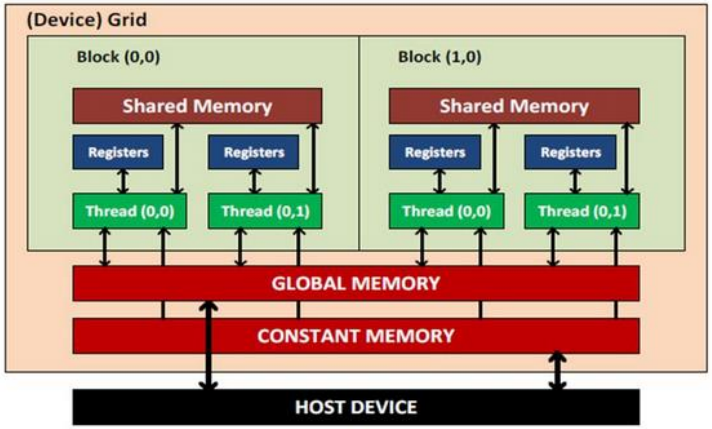
CUDA key concepts

- Device Global Memory (DRAM)
  - A typical GPU comes with its own global memory



CUDA key concepts

- Device Global Memory - API
  - Allocating memory on the device:
    - Transfers data from the host to the device memory
  - Kernel execution on the device:
    - Transfers the result from the device memory to the host
  - Free-up allocated memory on the device:
    - Transfer data to and from the device memory



CUDA key concepts

- CUDA programming keywords:

	EXECUTED ON THE –	CALLABLE FROM –
__device__ float function()	GPU (device)	CPU (host)
__global__ void function()	CPU (host)	GPU (device)
__host__float function()	GPU (device)	GPU (device)

- cudaMalloc() allocate memory on the host
- cudaFree() release objects from device memory
- cudaMemcpy() memory data transfer

CUDA key concepts

- CUDA programming keywords:

```
void vecAdd(float* A, float* B, float* C,int N) {
   int size=N*sizeOf(float);
   float *d_A,*d_B,*d_C;
   cudaMalloc((void**)&;d_A,size);
```

```
cudaMemcpy(d_A,A,size,cudaMemcpyHostToDevice);
```

```
cudaMemcpy(C,d_C,size,cudaMemcpyDeviceToHost);
```

```
cudaFree(d_A);

cudaFree(d_B);

cudaFree(d_C);
```

CUDA key concepts

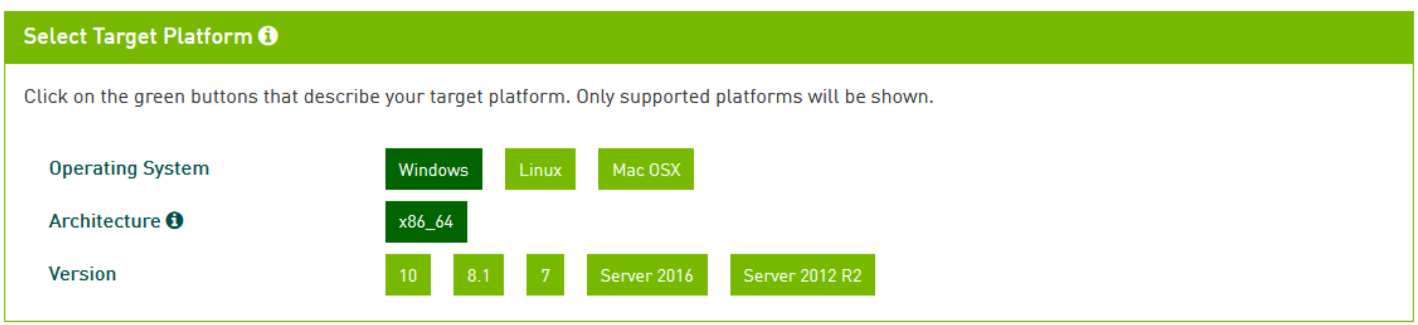
- CUDA programming keywords:
  - blockIdx — index of a block in a grid
  - threadIdx — index of a thread in a block
  - gridDim — the dimensions of the grid
  - blockDim — the dimensions of the block
  - dim3 — a data structure (like int) for threads
  - kernelName<<<#blocks , #threads>>>(parameter1, parameter2, ...)

Hint: threads → blocks → grids.

CUDA installation

- Requirements (commonly):
  - A CUDA-enabled NVidia GPU
  - A supported version of Microsoft Windows
  - A supported version of Visual Studio
  - The latest CUDA toolkit

    https://developer.nvidia.com/cuda-downloads



CUDA installation

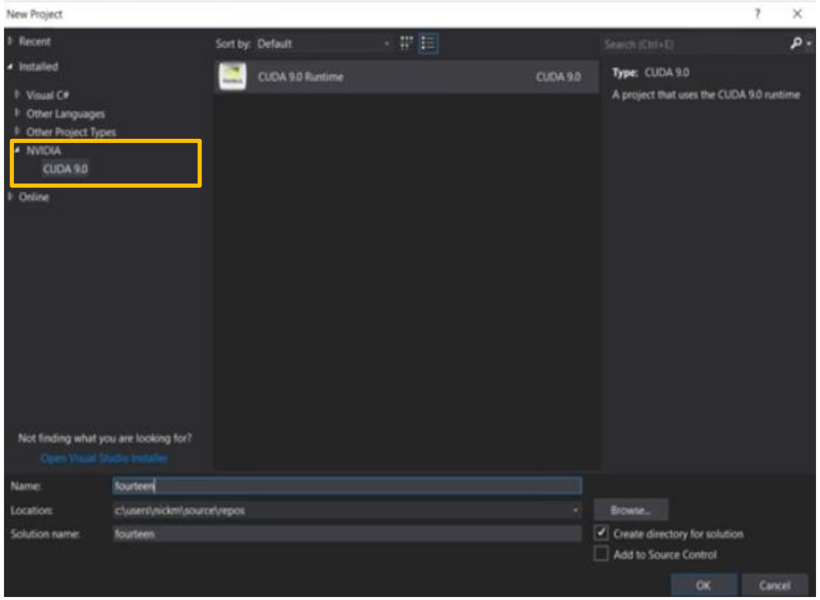
- Requirements (commonly):

Visual Studio Version	Native x86_64 support	X86_32 support on x86_32 (cross)
2017	YES	NO
2015	YES	NO
2015 Community edition	YES	NO
2013	YES	YES
2012	YES	YES
2010	YES	YES

Windows version	Native x86_64 support	X86_32 support on x86_32 (cross)
Windows 10	YES	YES
Windows 8.1	YES	YES
Windows 7	YES	YES
Windows Server 2016	YES	NO
Windows Server 2012 R2	YES	NO

CUDA installation

- Setting-up Visual Studio for CUDA:
  - Open Visual Studio → New → Project
  - Select NVidia from left pane
  - Creates a project with default codes
  - Tool set (project properties)
    - Select suitable platform toolkit
  - under CUDA C/C++:
    - Select Common
    - Set the CUDA Toolkit Custom Directory
  - Build the project!



CUDA Matrix Multiplication

- How to store matrices?
  - Row-major (C, C++, CUDA, …)
  - Column-major (Fortran, …)

M0,0	M0,1	M0,2	M0,3
M1,0	M1,1	M1,2	M1,3
M2,0	M2,1	M2,2	M2,3
M3,0	M3,1	M3,2	M3,3

M0,0	M0,1	M0,2	M0,3	M1,0	M1,1	M1,2	M1,3	M2,0	M2,1	M2,2	M2,3	M3,0	M3,1	M3,2	M3,3

M0,0	M1,0	M2,0	M3,0	M0,1	M1,1	M2,1	M3,1	M0,2	M1,2	M2,2	M3,2	M0,3	M1,3	M2,3	M3,3

CUDA Matrix Multiplication

- Addressing in CUDA
  - element(x,y) → x*width + y
  - Each element should be mapped to a thread

```
row=blockIdx.x*blockDim.x+threadIdx.x;
col=blockIdx.y*blockDim.y+threadIdx.y;
```

M0,0	M0,1	M0,2	M0,3
M1,0	M1,1	M1,2	M1,3
M2,0	M2,1	M2,2	M2,3
M3,0	M3,1	M3,2	M3,3

  - All threads in the same block have the same block index (e.g. yellow one)

blockDim.x=4 blockDim.y=1 blockIdx.x=0 threadIdx.x=0/1/2/3

CUDA Matrix Multiplication

- Final Kernel:

```
__global__ void simpleMatMulKernell (float* d_M, float* d_N, float* d_P, int width) {
        int row = blockIdx.y*width+threadIdx.y;
        int col = blockIdx.x*width+threadIdx.x;
        if(row<width && col <width) {
                float product_val = 0;
                for (int k=0; k<width; k++) {
                        product_val += d_M[row*width+k]*d_N[k*width+col];
                }
                d_p[row*width+col] = product_val;
        }
}
```

CUDA Matrix Multiplication

```
__global__ void MatrixMulTiled(float *Md, float *Nd, float *Pd, int width) {
   __shared__ float Mds[width_tile][width_tile];
   __shared__ float Nds[width_tile][width_tile];
   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   //identify what pd element to work on
   int row = by * width_tile + ty;
   int col = bx * width_tile + tx;
   float product_val = 0;
   for(int m = 0; m < width/width_tile; m++) {

      //load the tiles
      Mds[ty][tx] = Md[row*width + (m*width_tile + tx)];
      Nds[ty][tx] = Nd[col*width + (m*width_tile + ty)];
      _syncthreads();
      for(int k=0; k < width_tile; k++) {
         product_val += Mds[ty][k] * Nds[k][tx];
      }
      pd[row][col] = product_val;
   }
}
```

CUDA Matrix Multiplication

- Sample:
  - Final matrix: d_P (3x3)
  - for element (2,1):
    - row=2 column=1

2	4	1			4	8	9
8	7	4	x		1	7	0
7	4	9			2	5	4

```
product_val = 0 + d_M[2*3+0] * d_N[0*3+1]
product_val = 0 + d_M[6]*d_N[1] = 0+7*8=56
```

```
product_val = 56 + d_M[2*3+1]*d_N[1*3+1]
product_val = 56 + d_M[7]*d_N[4] = 84
```

```
product_val = 84+d_M[2*3+2]*d_N[2*3+1]
product_val = 84+d_M[8]*d_N[7] = 129
```

Any Questions?

Resources

- Tutorials-point website (fetched December 2018), see: https://www.tutorialspoint.com/cuda/cuda_performance_considerations.htm
- D. De Donno, A. Esposito, L. Tarricone and L. Catarinucci, “Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD,” IEEE Antennas and Propagation Magazine, vol. 52, no.3, June 2010.
- J. Sanders and E. Kandrot, “CUDA by Example,” NVIDIA, 2008.