




# Nvidia's Compute Unified Device Architecture (CUDA)



**Oregon State University**  
Mike Bailey  
mjb@cs.oregonstate.edu



This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/)



cuda.pptx

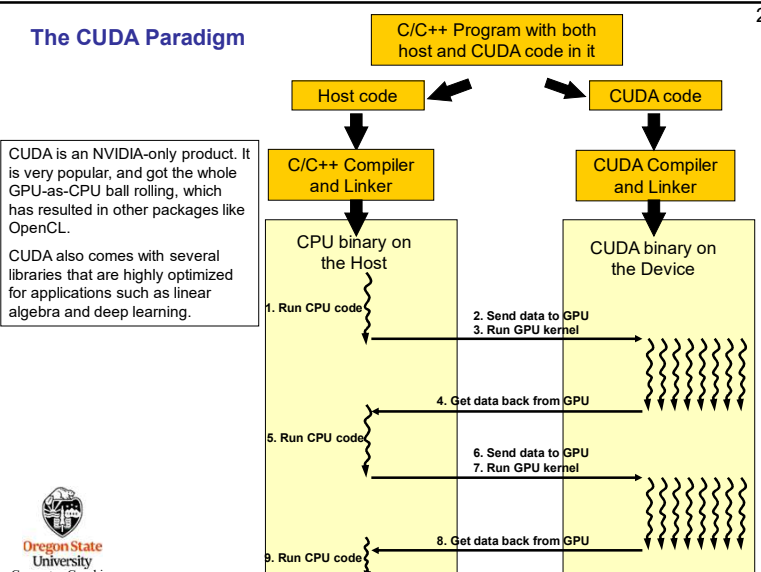
mjb - April 12, 2024

1


## The CUDA Paradigm

CUDA is an NVIDIA-only product. It is very popular, and got the whole GPU-as-CPU ball rolling, which has resulted in other packages like OpenCL.

CUDA also comes with several libraries that are highly optimized for applications such as linear algebra and deep learning.



The diagram illustrates the flow of code and data between the host and device. Host code is compiled by the C/C++ Compiler and Linker into a CPU binary on the host. CUDA code is compiled by the CUDA Compiler and Linker into a CUDA binary on the device. The execution sequence is: 1. Run CPU code; 2. Send data to GPU; 3. Run GPU kernel; 4. Get data back from GPU; 5. Run CPU code; 6. Send data to GPU; 7. Run GPU kernel; 8. Get data back from GPU.



mjb - April 12, 2024

2

## CUDA wants you to break the problem up into Pieces


If you were writing in **C/C++**, you would say:

```
void
ArrayMult( int n, float *a, float *b, float *c)
{
    for ( int i = 0; i < n; i++ )
        c[ i ] = a[ i ] * b[ i ];
}
```

If you were writing in **CUDA**, you would say:

```
__global__
void
ArrayMult( float *dA, float *dB, float *dC )
{
    int gid = blockIdx.x*blockDim.x + threadIdx.x;
    dC[gid] = dA[gid] * dB[gid];
}
```

Think of this as having an implied for-loop around it, looping through all possible values of *gid*

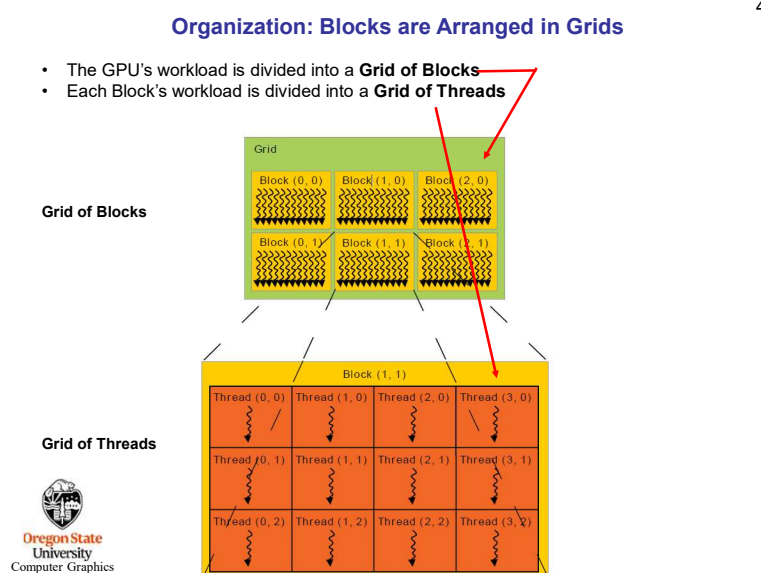


mjb - April 12, 2024


3

## Organization: Blocks are Arranged in Grids

- The GPU's workload is divided into a **Grid of Blocks**
- Each Block's workload is divided into a **Grid of Threads**



The diagram shows a 2x3 grid of blocks. Each block contains threads. A red arrow points from Block (1,1) to a detailed view of its thread grid, which is a 4x4 grid of threads labeled from Thread (0,0) to Thread (3,3).



mjb - April 12, 2024

4

### A Block is made up of a Grid of Threads

5

- The threads in a block each have *Thread ID* numbers within the Block
- Your CUDA program will use these Thread IDs to select work to do and pull the right data from memory
- Threads share data and synchronize while doing their share of the work
- Every **32** threads constitute a “Warp”. Each thread in a Warp simultaneously executes the same instruction on different pieces of data.
- But, it is likely that a Warp’s execution will need to stop at some point, waiting for a memory access. This would make the execution go idle – bad! So, it is worthwhile to have multiple Warps worth of threads available so that when one Warp blocks, another Warp can be swapped in.
- The threads in a *Thread Block* can cooperate with each other by:
  - Synchronizing their execution
  - Efficiently sharing data through a low latency shared memory
- Threads from different blocks cannot cooperate

Oregon State University Computer Graphics mjb – April 12, 2024

5

### Scheduling

6

- The hardware implements low-overhead Warp switching
  - A Warp whose next instruction has operands ready for consumption is eligible to be executed.
  - All threads in one Warp execute the same instruction at any given time, but on different data.
  - Threads in different Warps will usually be executing different instructions at any given time

This tells you that there needs to be a bunch of Warps to work on so that something is always ready to run  
If you can help it, these should be multiples of 32.

Oregon State University Computer Graphics mjb – April 12, 2024

6

### Threads Can Access Various Types of Storage

7

- Each thread has access to:
  - Its own R/W per-thread **registers**
  - Its own R/W per-thread **private memory**
- Each thread has access to:
  - Its block’s R/W per-block **shared memory**
- Each thread has access to:
  - The entire R/W per-grid **global memory**
  - The entire read-only per-grid **constant memory**
  - The entire read-only per-grid **texture memory**
- The CPU can read and write **global** and, **constant** memories

Oregon State University Computer Graphics mjb – April 12, 2024

7

### Different Types of CUDA Memory

8

Memory	Location	Who Uses
Registers	On-chip	One thread
Private	On-chip	One thread
Shared	On-chip	All threads in that block
Global	Off-chip	All threads + Host
Constant	Off-chip	All threads + Host


Oregon State University Computer Graphics mjb – April 12, 2024

8

### Thread Rules

9

- Each Thread has its own registers and private memory
- Each Block can use at most some maximum number of registers, divided equally among all Threads
- Threads can share local memory with the other Threads in the same Block
- Threads can synchronize with other Threads in the same Block
- Global and Constant memory is accessible by all Threads in all Blocks
- 192 or 256 are good numbers of Threads per Block (multiples of the Warp size)



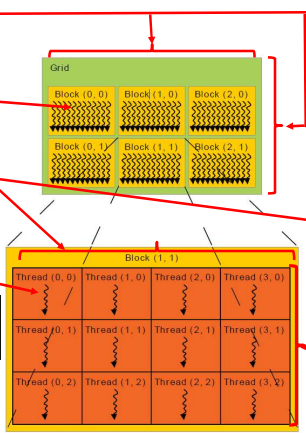
mjb - April 12, 2024

9


### A CUDA Thread can Query where it Fits in its “Community” of Threads and Blocks

10

- `dim3 gridDim;`
  - Dimensions of the blocks in this grid
- `dim3 blockIdx;`
  - This block's indexes within this grid
- `dim3 blockDim;`
  - Dimensions of the threads in this block
- `dim3 threadIdx;`
  - This thread's indexes within the block



Note: It is as if `dim3` is defined as:  
`typedef int[3] dim3;`  
 (it's not really – it is actually defined within the CUDA compiler)



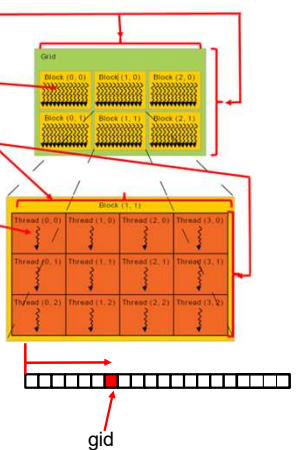
mjb - April 12, 2024

10

### A CUDA Thread needs to know where it Lives in its “Community” of Threads and Blocks


11

- `dim3 gridDim;`
  - Dimensions of the blocks in this grid
- `dim3 blockIdx;`
  - This block's indexes within this grid
- `dim3 blockDim;`
  - Dimensions of the threads in this block
- `dim3 threadIdx;`
  - This thread's indexes within the block



**For a 1D problem:**  
`int blockDim = blockDim.x*blockDim.x;`  
`int gid = blockDim + threadIdx.x;`  
`C[gid] = A[gid]*B[gid];`

**For a 2D problem:**  
`int blockDim = blockDim.y*gridDim.x + blockDim.x;`  
`int blockDim = blockDim.y*gridDim.x*blockDim.y;`  
`int gid = blockDim + threadIdx.y*blockDim.x + threadIdx.x;`  
`C[gid] = A[gid]*B[gid];`



mjb - April 12, 2024

11


### Types of CUDA Functions

12

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

`__global__` defines a kernel function – it must return `void`

Note: “`__`” is 2 underscore characters



mjb - April 12, 2024

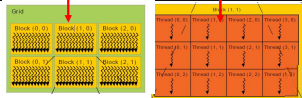
12

The C/C++ Program Calls a CUDA Kernel using a Special `<<...>>` Syntax 13

These are called "chevrons"

dim3      dim3

```
KernelFunction<<< NumBlocks, NumThreadsPerBlock >>>( arg1, arg2, ... );
```



C/C++ Program with both host and CUDA code in it

Host code      CUDA code

C/C++ Compiler and Linker      CUDA Compiler and Linker

CPU binary on the Host      CUDA binary on the Device

1. Run CPU code
2. Send data to GPU
3. Run GPU kernel
4. Get data back from GPU
5. Run CPU code
6. Send data to GPU
7. Run GPU kernel
8. Get data back from GPU
9. Run CPU code

Note that this is just like calling the C/C++ function:  
**KernelFunction( arg1, arg2, ... );**  
 except that we have designated it to run on the GPU with a particular block/thread configuration.

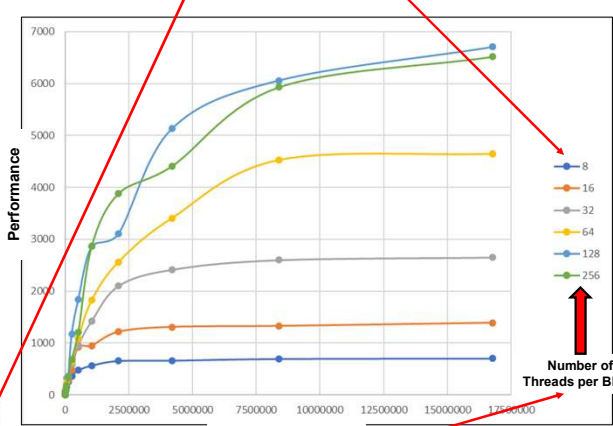
Oregon State University Computer Graphics

mjb - April 12, 2024

13

One of my own Experiments with Number of Threads Per Block 14

```
KernelFunction<<< NumBlocks, NumThreadsPerBlock >>>( arg1, arg2, ... );
```



Performance

Dataset Size

Number of Threads per Block

$NumBlocks = DataSetSize / NumThreadsPerBlock$

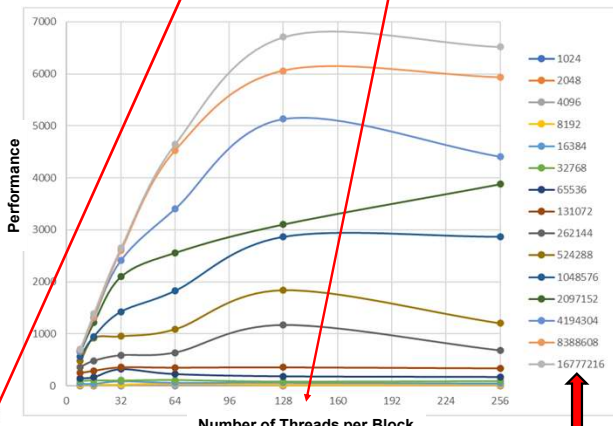
Oregon State University Computer Graphics

mjb - April 12, 2024

14

One of my own Experiments with Number of Threads Per Block 15

```
KernelFunction<<< NumBlocks, NumThreadsPerBlock >>>( arg1, arg2, ... );
```



Performance

Number of Threads per Block

Dataset Size

$NumBlocks = DataSetSize / NumThreadsPerBlock$

Oregon State University Computer Graphics

mjb - April 12, 2024

15

Getting CUDA Programs to Run under Linux 16

This is the Makefile we use:

```
CUDA_PATH = /usr/local/apps/cuda/cuda-10.1
CUDA_BIN_PATH = $(CUDA_PATH)/bin
CUDA_NVCC = $(CUDA_BIN_PATH)/nvcc

arrayMul: arrayMul.cu
$(CUDA_NVCC) -o arrayMul arrayMul.cu
```

This is the path where the CUDA tools are loaded on our Oregon State University systems.

Or, without the Makefile syntax:

```
/usr/local/apps/cuda/cuda-10.1/bin/nvcc -o arrayMul arrayMul.cu
```

We also have the CUDA-11 and CUDA-12 tools loaded for your use. You can use them if you want. But, given the wide breadth of different Nvidia cards around campus, **CUDA-10** seems to be the one that will run **everywhere!** I recommend you use it.

Oregon State University Computer Graphics

mjb - April 12, 2024

16

## Getting CUDA Programs to Run under Visual Studio

17

1. Install Visual Studio if you haven't already. If you are an OSU student, go to:

<https://azureforeducation.microsoft.com/devtools>

Click the blue **Sign In** button on the right.  
Login using your ONID@oregonstate.edu username and password.  
Install **Visual Studio 2022 Enterprise**

2. Install the CUDA toolkit for Windows. It is available here:

[https://developer.nvidia.com/cuda-downloads?target\\_os=Windows&target\\_arch=x86\\_64&target\\_version=11&target\\_type=exe\\_local](https://developer.nvidia.com/cuda-downloads?target_os=Windows&target_arch=x86_64&target_version=11&target_type=exe_local)



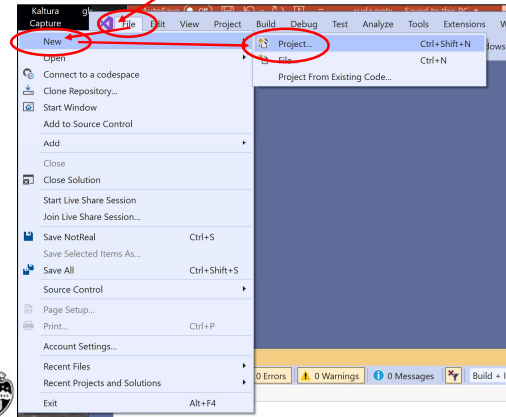
mjb - April 12, 2024

17

## Getting CUDA Programs to Run under Visual Studio

18

From the main screen, click **File** → **New** → **Project...**



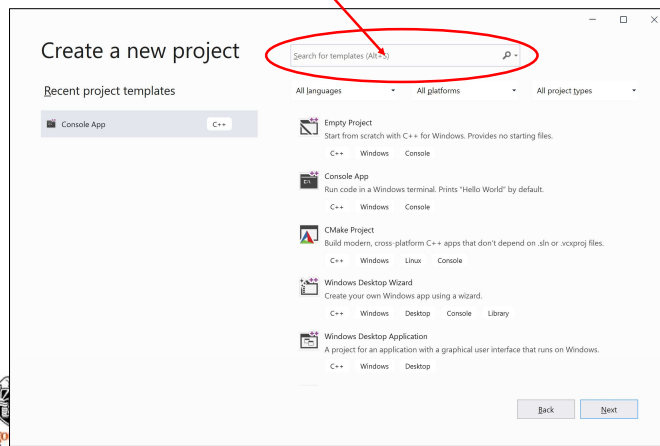
mjb - April 12, 2024

18

## Getting CUDA Programs to Run under Visual Studio

19

Then, in this *templates* box, type: **CUDA**



mjb - April 12, 2024

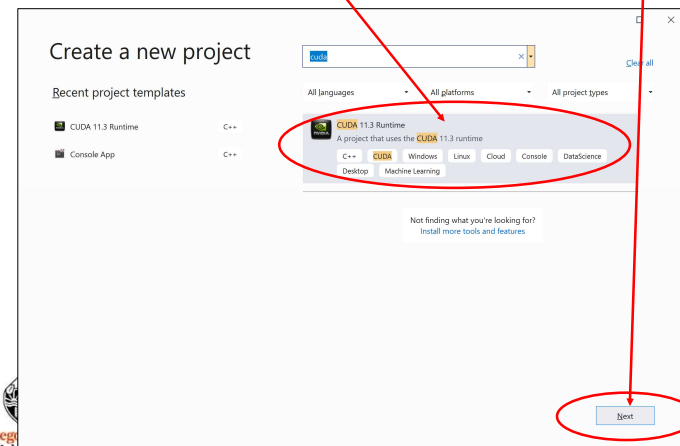
19

## Getting CUDA Programs to Run under Visual Studio

20

After a few seconds, you will then see this.

Click **Next**.



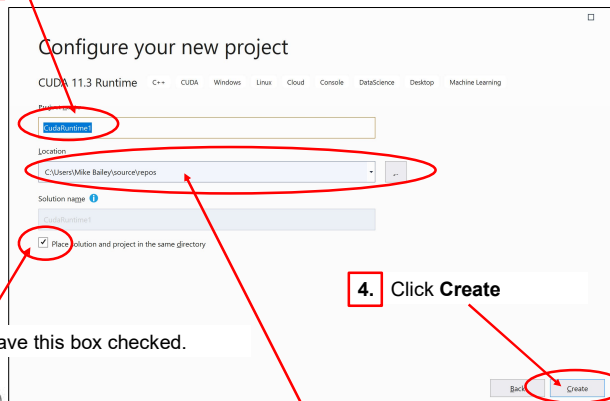
mjb - April 12, 2024

20

## Getting CUDA Programs to Run under Visual Studio

21

2. Give the name you want for the folder and project



3. Leave this box checked.

1. Navigate to the folder you want to contain this project folder.



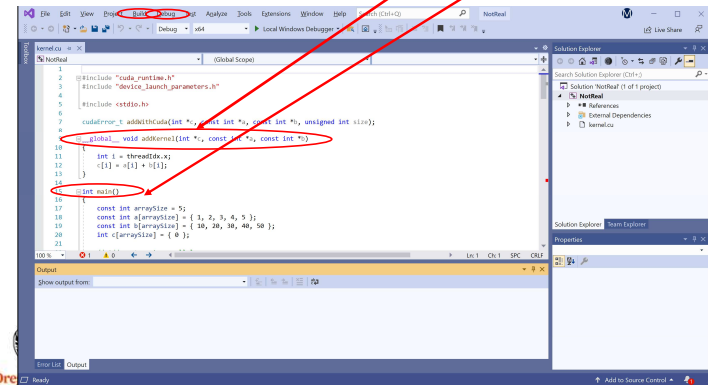
mjb - April 12, 2024

21

## Getting CUDA Programs to Run under Visual Studio

22

1. Visual Studio then "writes" a program for you. It has both CUDA and C++ code in it. Its structure looks just like our notes' examples.
2. You can click **Build** → **Build** to compile it, both the C++ and the CUDA code.
3. You can click **Debug** → **Start Without Debugging** to run it.
4. You can then either modify this file, or clear it and paste your own code in.

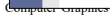
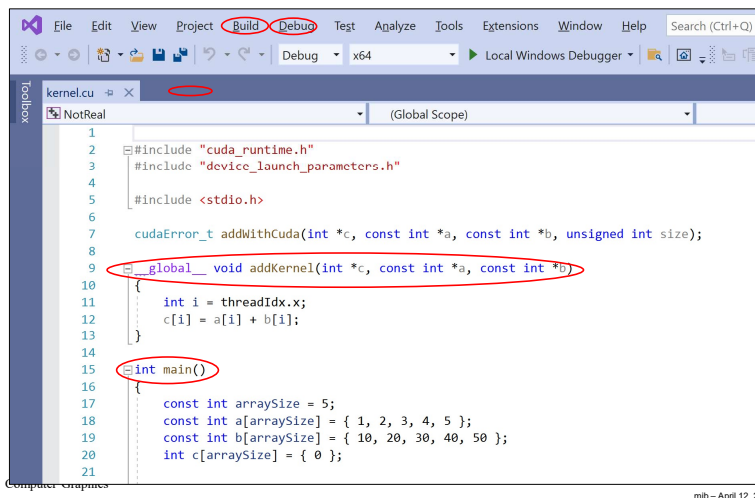


mjb - April 12, 2024

22

## Getting CUDA Programs to Run under Visual Studio

23



mjb - April 12, 2024

23

## Using CUDA and OpenMP Together

24

This is the Makefile we use on Linux:

```
CUDA_PATH = /usr/local/apps/cuda/cuda-10.1
CUDA_BIN_PATH = $(CUDA_PATH)/bin
CUDA_NVCC = $(CUDA_BIN_PATH)/nvcc

arrayMul: arrayMul.cu
$(CUDA_NVCC) -o arrayMul arrayMul.cu -Xcompiler -fopenmp
```

Or, on Linux, but without the Makefile syntax:

```
/usr/local/apps/cuda/cuda-10.1/bin/nvcc -o arrayMul arrayMul.cu -Xcompiler -fopenmp
```

Or, in Visual Studio:

1. Go to the Project menu → Project Properties
2. Change the setting Configuration Properties → C/C++ → Language → OpenMP Support to "Yes (/openmp)"



We also have the CUDA-11 and CUDA-12 tools loaded for your use. You can use them if you want. But, given the wide breadth of different Nvidia cards around campus, **CUDA-10** seems to be the one that will run **everywhere!** I recommend you use it.

mjb - April 12, 2024

24

## Using Multiple GPU Cards with CUDA

25

```
int deviceCount;  
cudaGetDeviceCount( &deviceCount );  
  
...  
  
int device;    // 0 ≤ device ≤ deviceCount - 1  
cudaSetDevice( device );
```

