

# 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/)

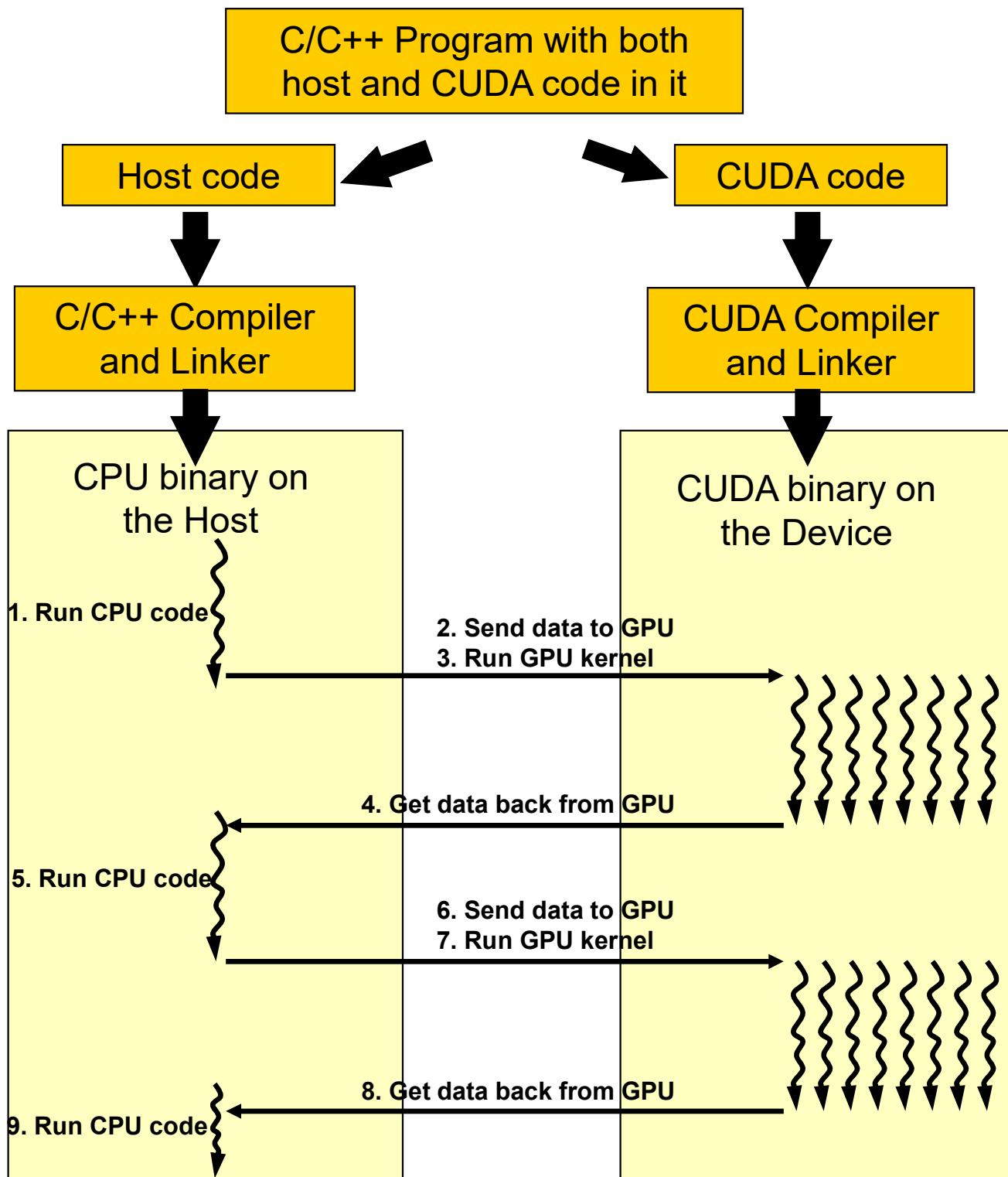


**Oregon State**  
University  
Computer Graphics

# 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.



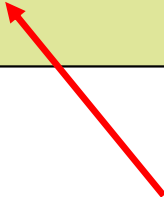
## CUDA wants you to break the problem up into Pieces

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];  
}
```

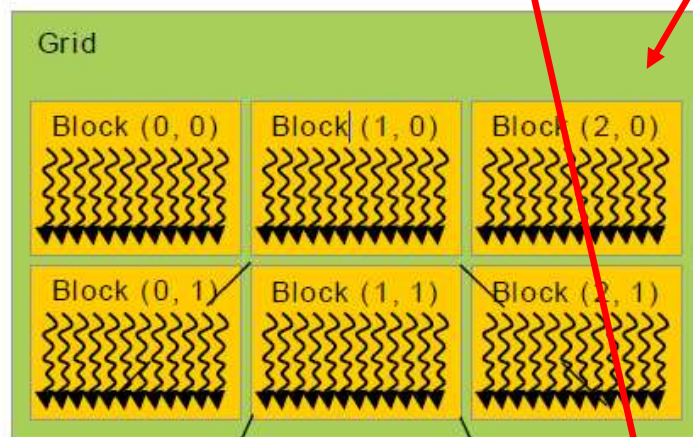


Think of this as having an implied for-loop around it,  
looping through all possible values of *gid*

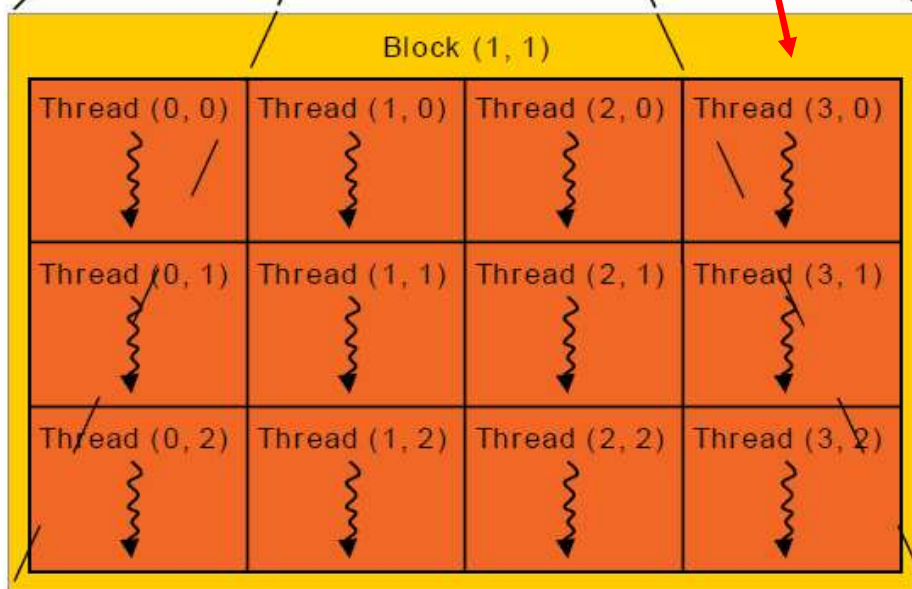
## Organization: Blocks are Arranged in Grids

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

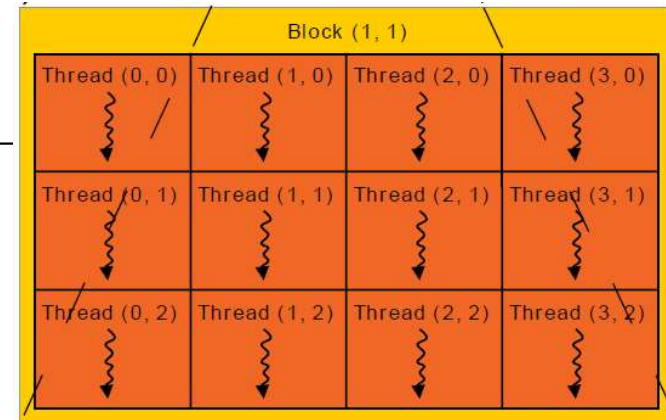
Grid of Blocks



Grid of Threads



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



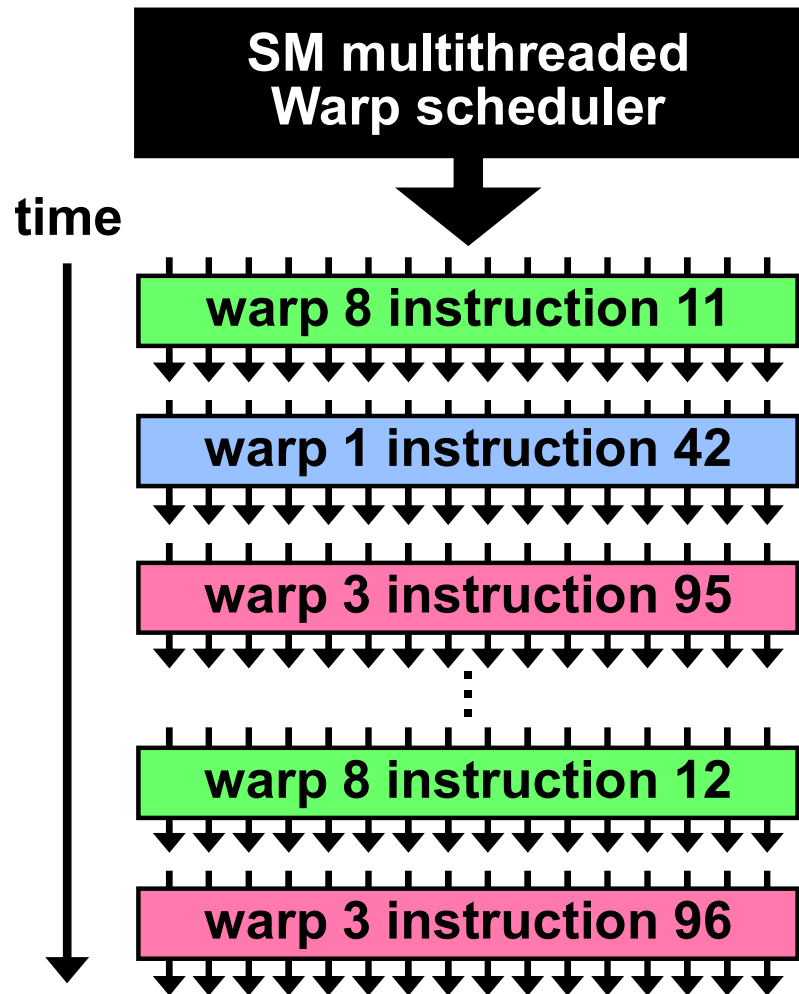
- The threads in a block each have *Thread ID* numbers within the Block
- Your CUDA program will use these Thread IDs to select work to do and pull the right data from memory
- Threads share data and synchronize while doing their share of the work
- Every **32** threads constitute a “*Warp*”. Each thread 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

## Scheduling

- The hardware implements low-overhead Warp switching

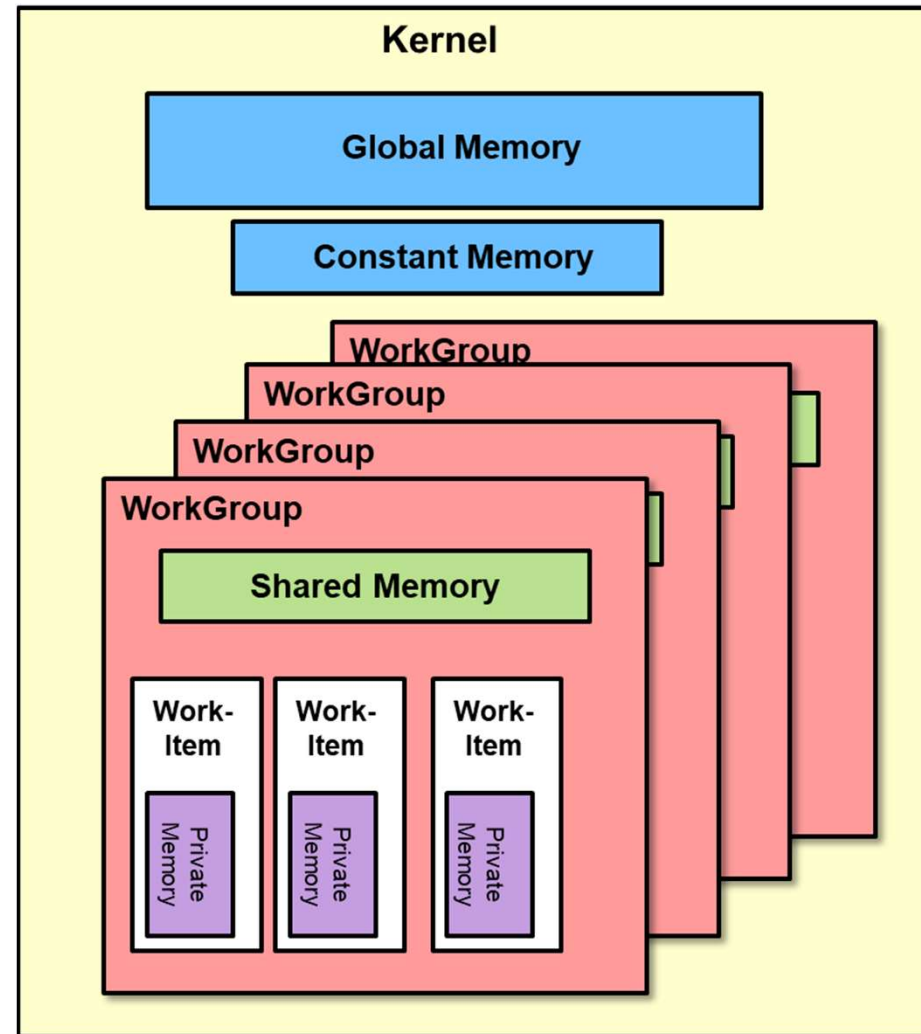
- A Warp whose next instruction has operands ready for consumption is eligible to be executed.
- All threads in one Warp execute the same instruction at any given time, but on different data.
- Threads in different Warps will usually be executing different instructions at any given time

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.



## Threads Can Access Various Types of Storage

- 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



## Different Types of CUDA Memory

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





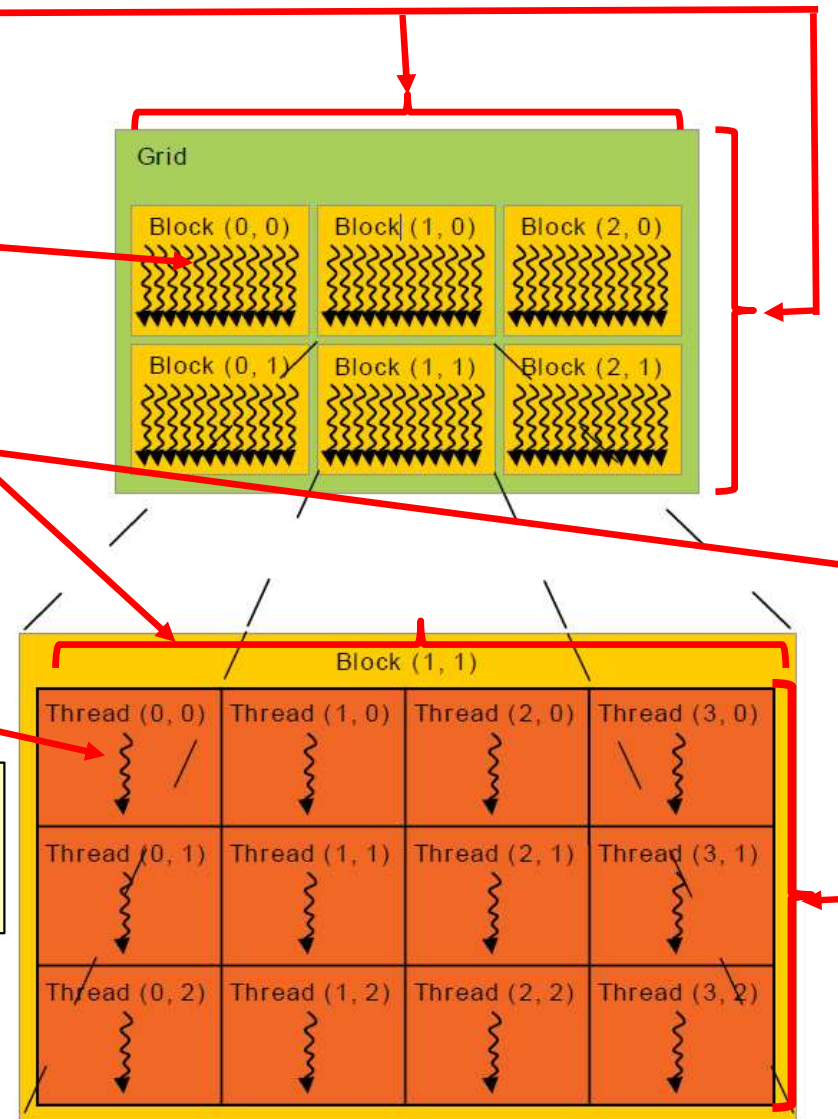
# Thread Rules

- 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)



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

- `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:

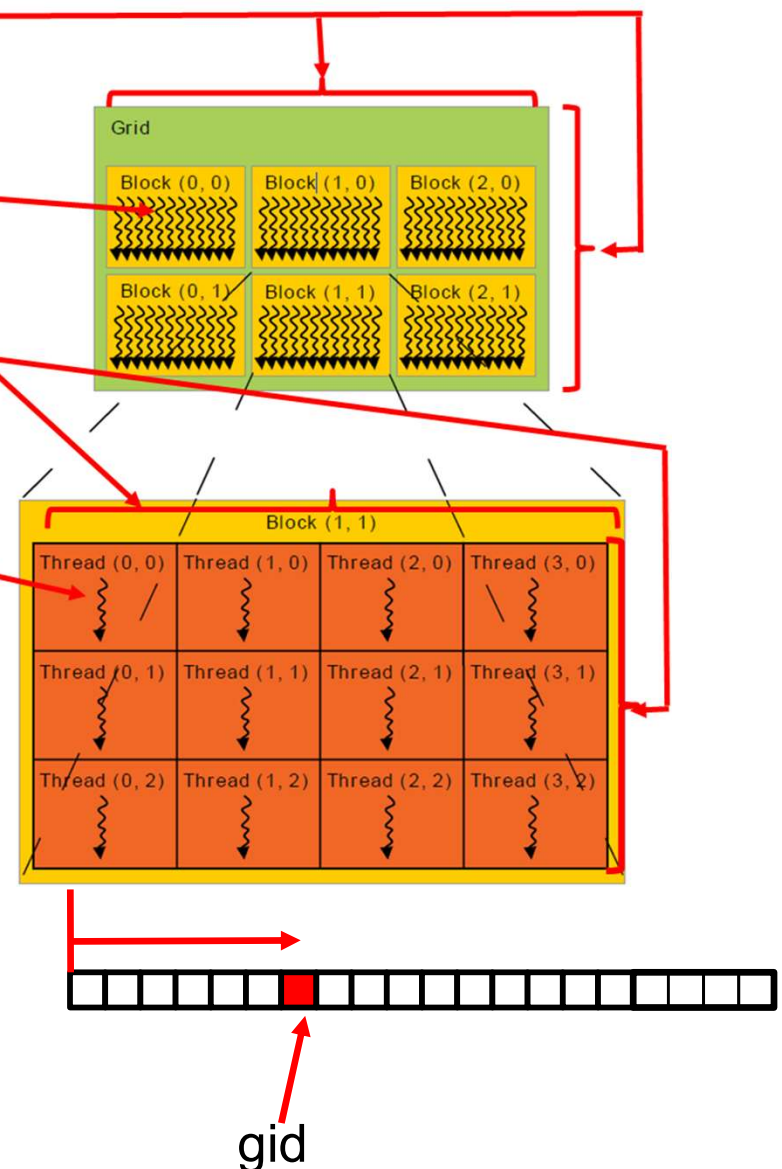
```
typedef int[3] dim3;
```

(it's not really – it is actually defined within the CUDA compiler)



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

- `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 blockThreads = blockDim.x*blockDim.x;
int gid = blockThreads + threadIdx.x;
C[gid] = A[gid]*B[gid];
```

## For a 2D problem:

```
int blockNum = blockIdx.y*gridDim.x + blockIdx.x;
int blockThreads = blockNum*blockDim.x*blockDim.y;
int gid = blockThreads + threadIdx.y*blockDim.x + threadIdx.x;
C[gid] = A[gid]*B[gid];
```

## Types of CUDA Functions

	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



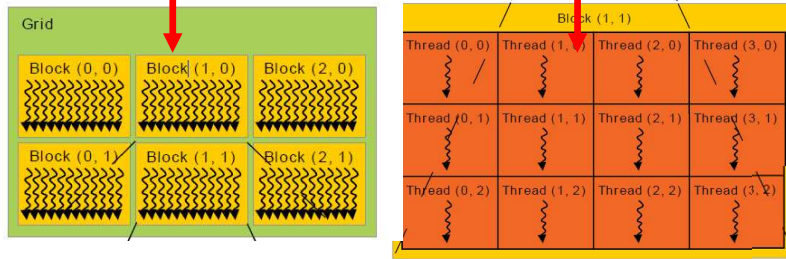
# The C/C++ Program Calls a CUDA Kernel using a Special <<...>> Syntax

These are called "chevrons"

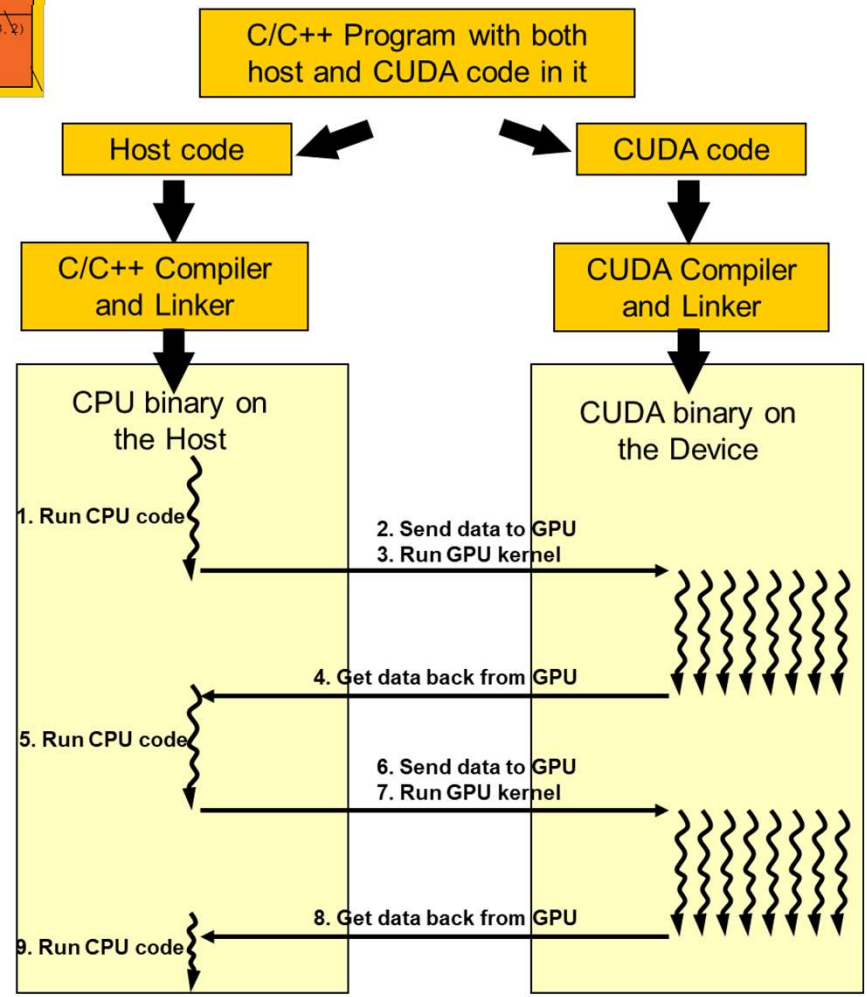
dim3

dim3

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

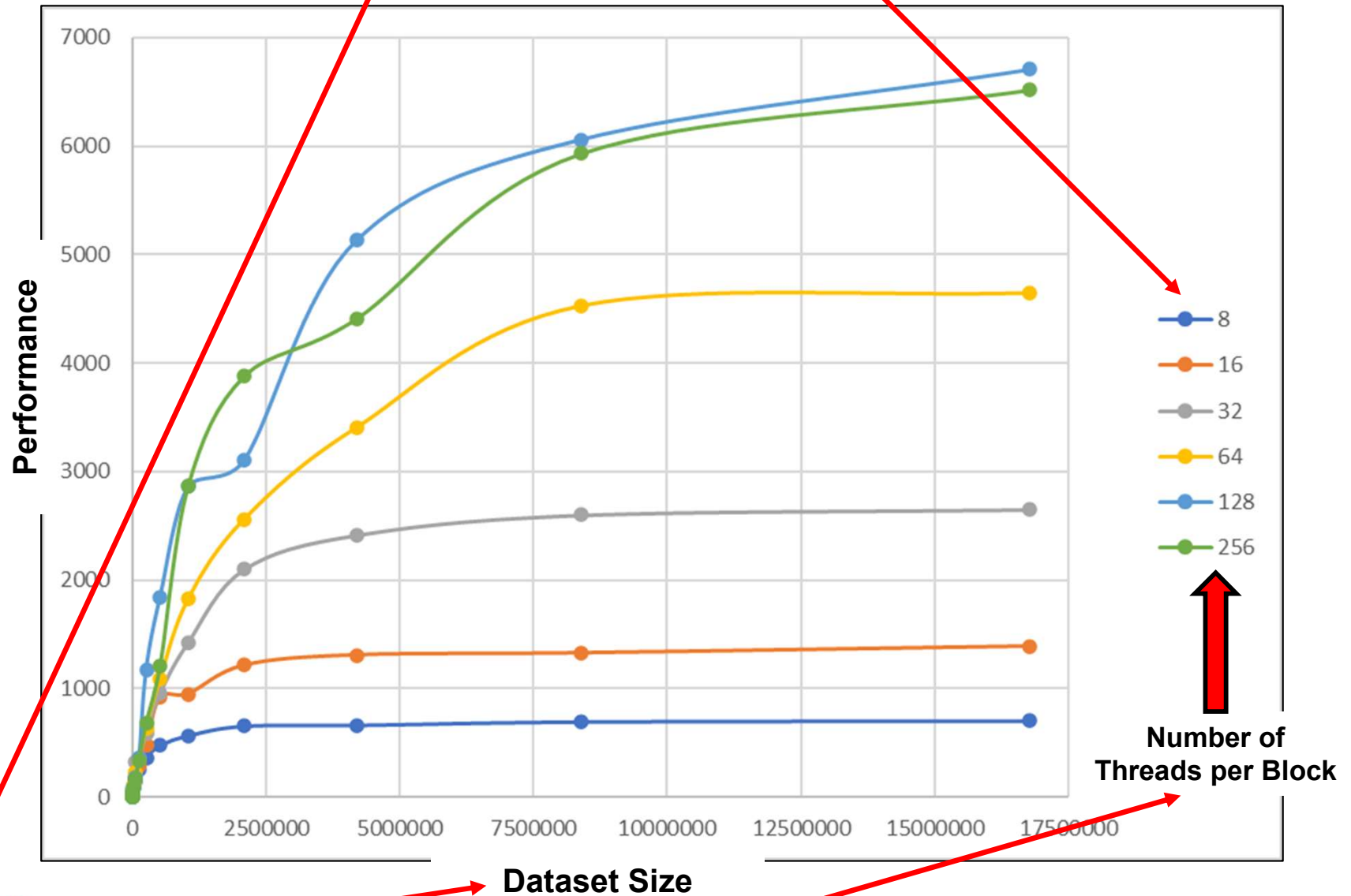


Note that this is just like calling the C/C++ function:  
**KernelFunction( arg1, arg2, ... );**  
 except that we have designated it to run on the GPU with a particular block/thread configuration.



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

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



Oregon State

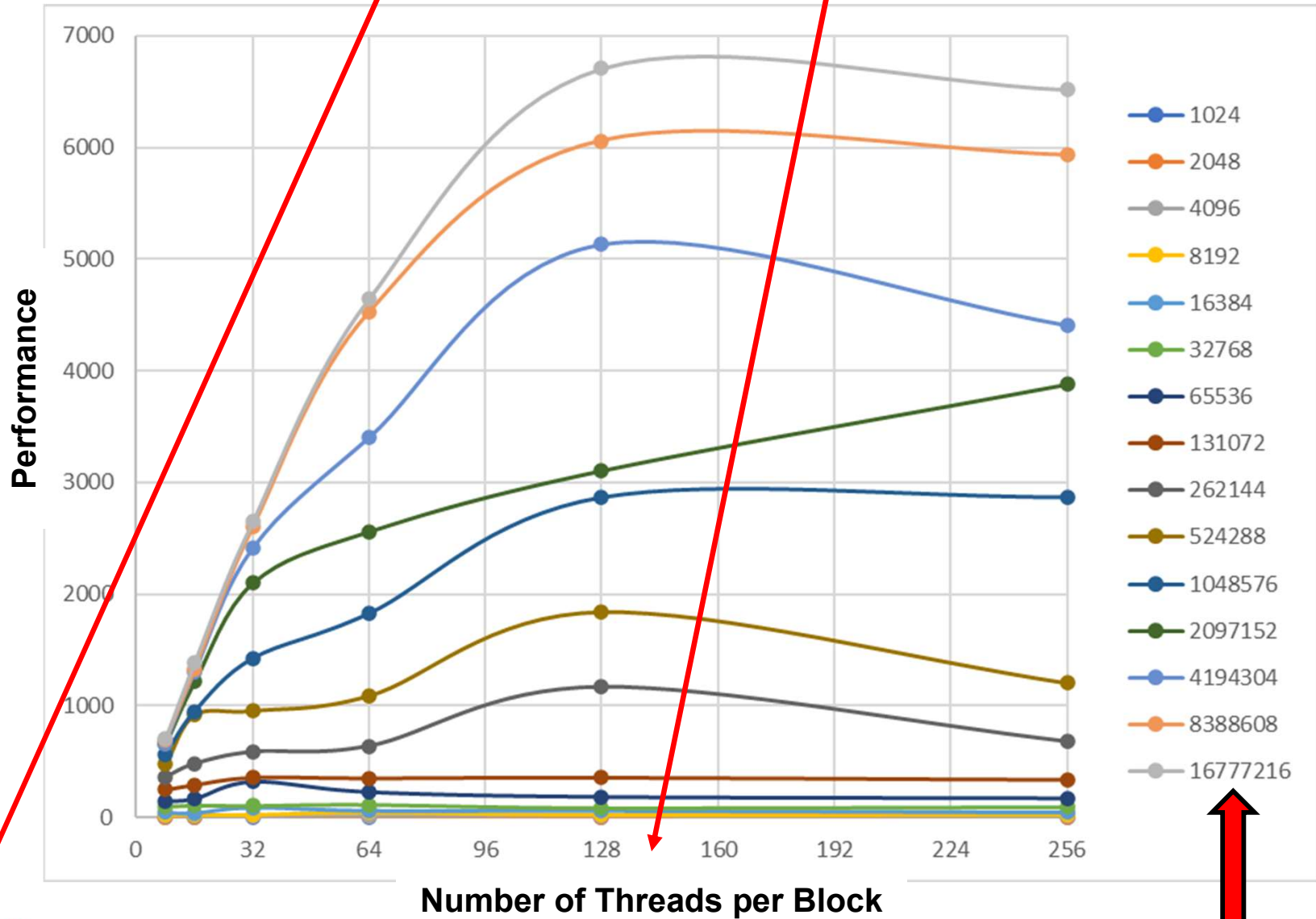
Dataset Size

**NumBlocks = DataSetSize / NumThreadsPerBlock**



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

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



Oregon State University

$$\text{NumBlocks} = \text{DataSetSize} / \text{NumThreadsPerBlock}$$

Dataset Size

# Getting CUDA Programs to Run under Linux

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.



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.

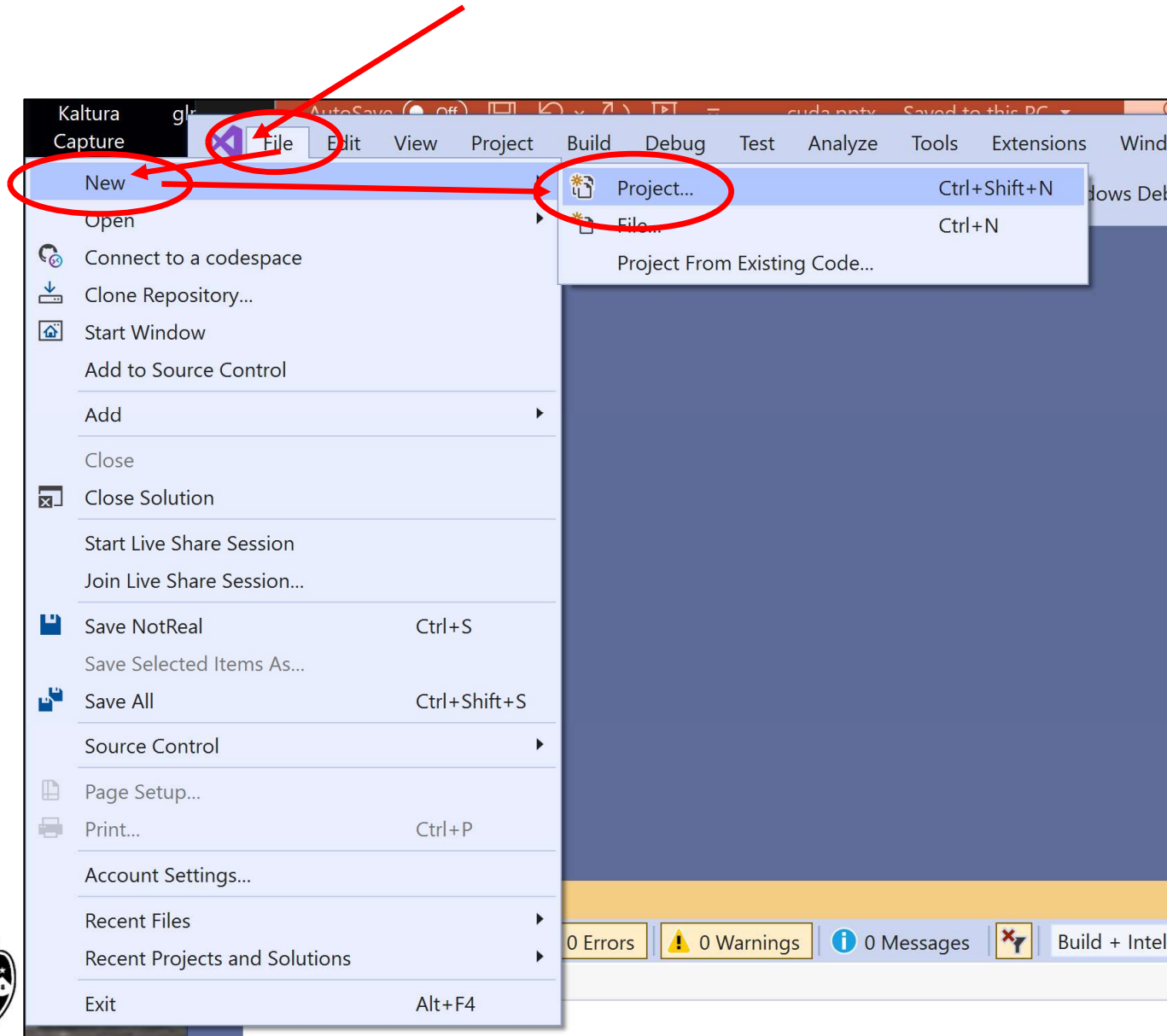
Install ***Visual Studio 2022 Enterprise***

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

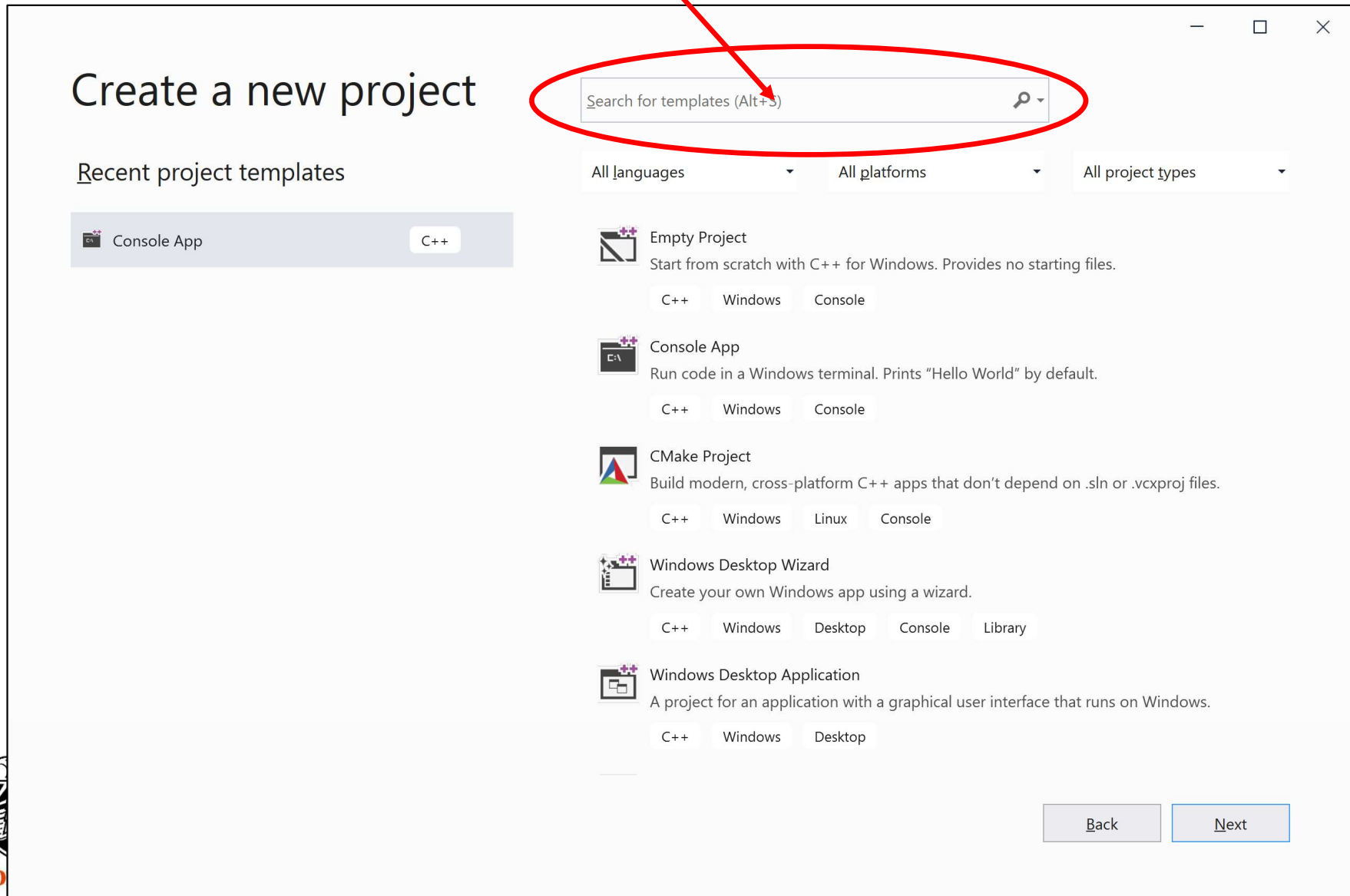
[https://developer.nvidia.com/cuda-downloads?target\\_os=Windows&target\\_arch=x86\\_64&target\\_version=11&target\\_type=exe\\_local](https://developer.nvidia.com/cuda-downloads?target_os=Windows&target_arch=x86_64&target_version=11&target_type=exe_local)



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

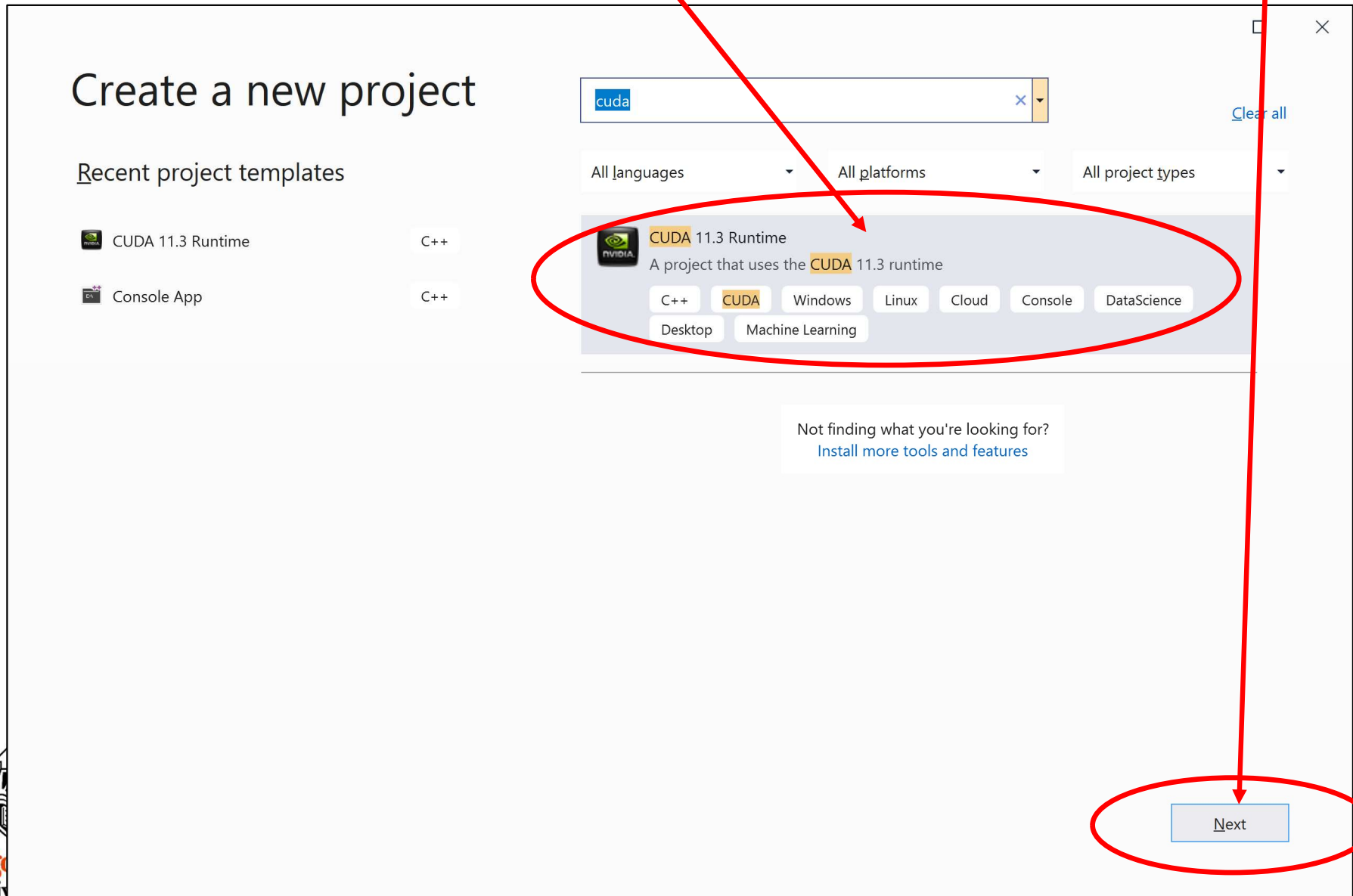


Then, in this *templates* box, type: **CUDA**



After a few seconds, you will then see this.

Click **Next**.



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

Configure your new project

CUDA 11.3 Runtime C++ CUDA Windows Linux Cloud Console DataScience Desktop Machine Learning

Project name

CudaRuntime1

Location

C:\Users\Mike Bailey\source\repos

Solution name

CudaRuntime1

Place solution and project in the same directory

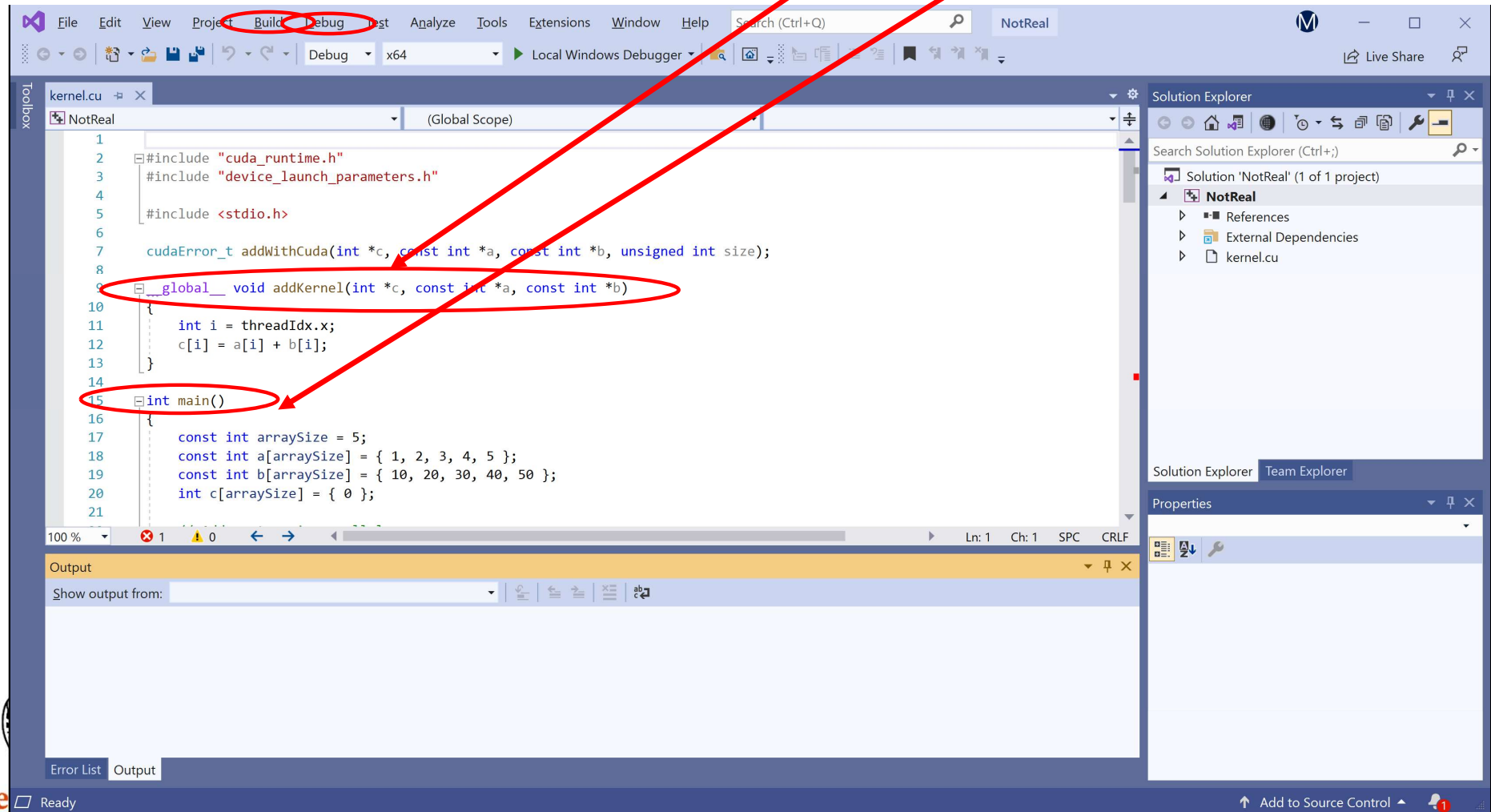
Back Create

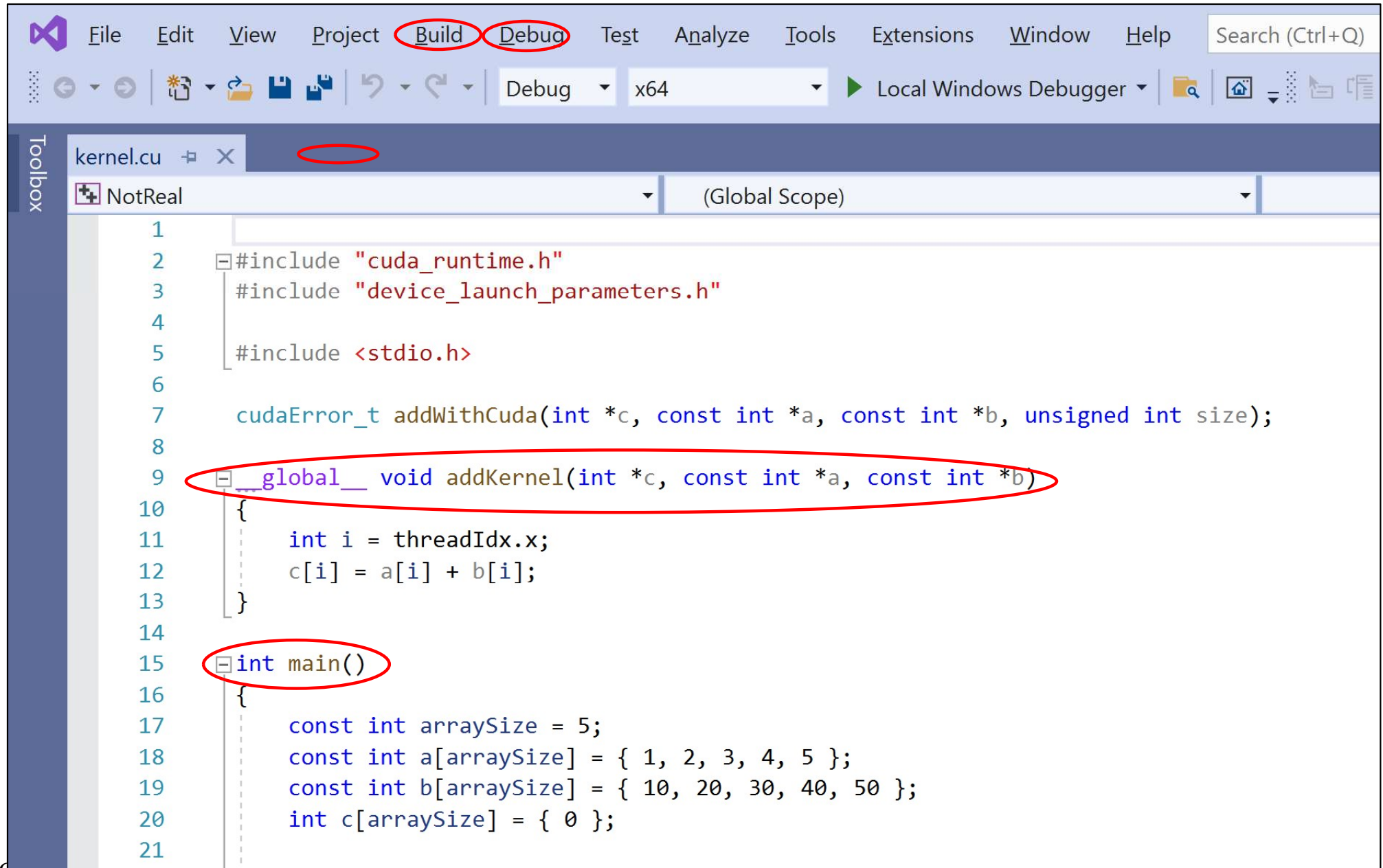
**4.** Click **Create**

**3.** Leave this box checked.

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

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.





```
1
2  #include "cuda_runtime.h"
3  #include "device_launch_parameters.h"
4
5  #include <stdio.h>
6
7  cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size);
8
9  __global__ void addKernel(int *c, const int *a, const int *b)
10 {
11     int i = threadIdx.x;
12     c[i] = a[i] + b[i];
13 }
14
15 int main()
16 {
17     const int arraySize = 5;
18     const int a[arraySize] = { 1, 2, 3, 4, 5 };
19     const int b[arraySize] = { 10, 20, 30, 40, 50 };
20     int c[arraySize] = { 0 };
21
```

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.



## Using Multiple GPU Cards with CUDA

```
int deviceCount;  
cudaGetDeviceCount( &deviceCount );
```

...

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

