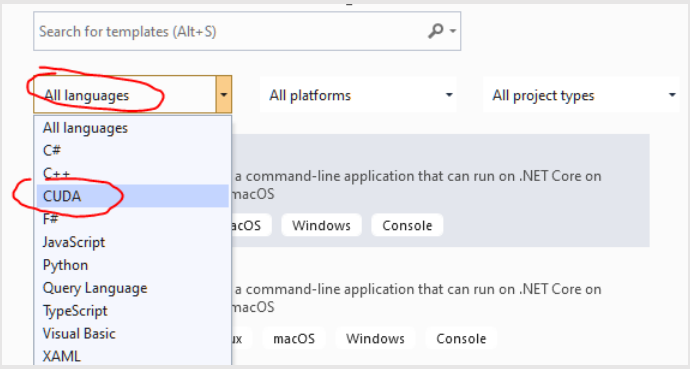
CUDA Lab 1. Getting started with CUDA using Visual Studio 2019

Objectives

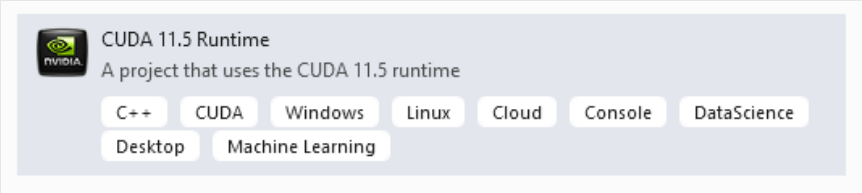
1. Set up CUDA project in Visual Studio 2019
2. Understand how to copy data to and from GPU
3. Learn how to pass computing instructions to be executed on GPU
4. Gain a basic understanding of how CUDA threads are organized

Exercise 1. Set up CUDA project in Visual Studio 2019

- If you are doing the lab from home using your own computer and your computer has a Nvidia GPU, download CUDA from Nvidia's website: https://developer.nvidia.com/cuda-toolkit-archive
  The lab descriptions are based on CUDA Toolkit 11.5, but any version start from CUDA Toolkit 11.2 should also work, including the latest version, which is CUDA Toolkit 11.6.
- Follow on-screen prompts for installation structures.
- If you do not have an Nvidia card on your own computer, you may have to use a PC in Fenner 177 graphics Lab or remotely connect to a PC in the graphics Lab.
- Once CUDA is installed on your computer, Open VS2019 and create a new project(File→ New→Project)
- Check the installed packages. If CUDA has been installed, you should find a CUDA program template by searching CUDA in the search box



Select CUDA, a CUDA program template will be displayed:



It does not matter if the CUDA version displayed on your computer is different from the one shown here.

- Select CUDA runtime program template and press the next button.

- Give a name to your project and specify where you want to save your project, which is the same as you are creating a C++ project.
    1) Now a simple CUDA program template is created. The task is to perform element-wise addition of two array of numbers: a[ ] and b[ ]. The result is stored in the array of c[ ]. (**Remark**, a 1D array of n elements is commonly referred to as a n-D vector).
- Build and run the CUDA program to check if it can be built and run properly.
- If the program is working properly, let's walk through the CUDA code and to gain a basic understanding of the basic idea of how GPU can be used as a general purpose parallel computing device.
- CUDA is an extension to C/C++ environment, which allows us to write a program containing both C/C++ and CUDA codes, where the C/C++ runs on CPU and CUDA runs on GPU, a kind of heterogeneous programming code.
- In the template CUDA program, "cuda_runtime.h" and "device_launch_parameters.h" are two header files of C++-style wrapper for the CUDA C API. We can get access to some basic functions in the CUDA libraries by including these two header files:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
```

- cudaError_t addWithCuda (...) is a helper function for using CUDA to add vectors in parallel, it returns a built-in CUDA error type.
- __global__ void addKernel(...){...}
    1) __global__ indicates this function is the kernel function, which is called from host code but executed on the device (GPU).
    2) int i = threadIdx.x;
        - threadIdx.x is the index of a thread in a thread block.
    3) The detailed explanations on CUDA thread models will be detailed in lectures and labs in future.
- The code in main( ) method can be divided into three sections:
    1) Create data;
    2) Pass data to GPU;
    3) Computing the vector addition operation in parallel on GPU by executing kernel function;
    4) Copy computational result back to CPU memory.
- To better understand how GPU does it in parallel, let's first consider how we add two arrays of numbers described in the template CUDA program in C++:

```
const int arraySize = 5;
const int a[arraySize] = { 1, 2, 3, 4, 5 };
const int b[arraySize] = { 10, 20, 30, 40, 50 };
int c[arraySize] = { 0 };
```

In this case, the CPU needs to sequentially calculate the five number one by one. The length of calculation is 5! We can write a simple C++ function shown below to complete the task:

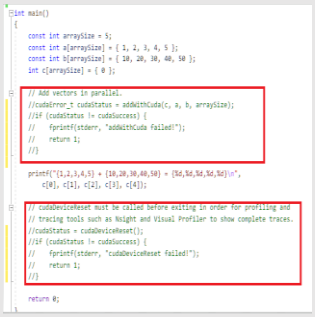
```
void add(int* c, const int* a, const int* b)
{
    for (int i = 0; i < 5; ++i)
        c[i] = a[i] + b[i];
}
```

- Now let's see how GPU does the vector addition. Unlike CPU, GPU has massive cores and each core can have several threads (recent NVidia GPUs can execute up to 16 threads/core). This means we have a team of threads to do the task. For example, to find the five element of array c[ ], one simple idea is to distribute the calculation of the five elements to five different threads: each thread handles the calculation of one addition. But the key question to answer is: who does what? How to distinct different threads? The answer is thread ID! Each thread in a 1D thread block has a thread index threadIdx.x: 0, 1, 2, …, n. You can refer this as the name of the thread. If we are employing 5 thread to find a[0]+b[0], a[1]+b[1], a[2]+b[2], a[3]+b[3], a[4]+b[4], we can complete the task in the following way without using for loop:

```
__global__ void addKernel (float* a, float* b, float* C)
{
        c[ threadIdx.x ] = a[ threadIdx.x ] + b[ threadIdx.x ];
}
```

- To better understand how CPU and GPU collaborate and get the job done, let's comment out the use of helper function addWithCuda(...) from the main( ) method and rewrite the method directly to gain some basic understanding of how CUDA does the work in parallel.

```
//cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
//if (cudaStatus != cudaSuccess) {
//    fprintf(stderr, "addWithCuda failed!");
//    return 1;
//}
… …
 //cudaStatus = cudaDeviceReset();
 //if (cudaStatus != cudaSuccess) {
 //    fprintf(stderr, "cudaDeviceReset failed!");
 //    return 1;
 //}
```



- CPU only solution: Add the C++ code add( ) method shown above and calculate the vector addition in the following way:

```
add(c, a, b);
printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
    c[0], c[1], c[2], c[3], c[4]);
```

- **CPU+GPU solution in CUDA**:
  1) Create data in C++: Specify the size and initialize the required arrays of data involved in the element-wise vector addition:

```
const int arraySize = 5;
const int a[arraySize] = { 1, 2, 3, 4, 5 };
const int b[arraySize] = { 10, 20, 30, 40, 50 };
int c[arraySize] = { 0 };
```

  2) Prepare data for GPU

a) Allocate memory required on GPU side using cudaMalloc () to store vectors a[ ], b[ ] and the sum of vector **a** and **b**: c[ ].

```
int *dev_a = 0;
int *dev_b = 0;
int *dev_c = 0;

cudaMalloc((void**)&dev_a, arraySize * sizeof(int));
cudaMalloc((void**)&dev_b, arraySize * sizeof(int));
cudaMalloc((void**)&dev_c, arraySize * sizeof(int));
```

b) Copy data a[ ] and b[ ] from host(CPU) to device(GPU) using cudaMemcpy( ):

```
cudaMemcpy(dev_a, a, arraySize * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, arraySize * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_c, c, arraySize * sizeof(int), cudaMemcpyHostToDevice);
```

Xx

3) **Perform vector addition in parallel on GPU**: add array a[ ] and b[ ] and store the value to array c[ ]. As there are five calculations to be performed:

a[0]+b[0], a[1]+b[1], a[2]+b[2], a[3]+b[3], a[4]+b[4],

five GPU threads can be employed to conduct the task. As the number of operations to be performed is very small, **one** thread block containing **five** threads is enough for this task. To execute addKernel( ) on GPU to compute all these 5 addition operations on five different threads in parallel can be issued in our C++ program in the following manner:

```
addKernel << <1,    5>>> > (dev_c, dev_a, dev_b);
```

1 thread block

5 threads /per block

A kernel is a c++ function defined in CUDA C++, an extension to C++. It is defined using the *__global__* declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using configuration syntax inside the *triple-angle-brackets* <<<... ...>>>, which specify how many threads are used and how the threads are organized. In general, it takes the following format

```
kernel_name <<<number_of_blocks, number_of_threads/per block>>>(argument list)
```

More detailed introduction about the thread organization configuration will be given in next week's lecture. If you want to know more about how to organize CUDA thread hierarchy now, you can have a look at the information shown at https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy.

Now let's replace the ordinary C++ solution add( ) with CUDA solution:

4) Copy computation result back from GPU to CPU:

a) First, synchronize threads' execution to wait all the kernels to finish their calculation

```
cudaDeviceSynchronize();
```

b) Copy computation result back from GPU to CPU memory using cudaMemcpy ():

```
cudaMemcpy(c, dev_c, arraySize * sizeof(int), cudaMemcpyDeviceToHost);
```

5) Now GPU has done its work, the allocated buffers on GPU can now be released

```
cudaFree(dev_c);
cudaFree(dev_a);
cudaFree(dev_b);
```

```
//addCuda(c, a, b);

int* dev_a = 0;
int* dev_b = 0;
int* dev_c = 0;
cudaMalloc((void**)&dev_a, arraySize * sizeof(int));
cudaMalloc((void**)&dev_b, arraySize * sizeof(int));
cudaMalloc((void**)&dev_c, arraySize * sizeof(int));

cudaMemcpy(dev_a, a, arraySize * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, arraySize * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_c, c, arraySize * sizeof(int), cudaMemcpyHostToDevice);

addKernel << <1, 5 >> > (dev_c, dev_a, dev_b);
cudaDeviceSynchronize();
cudaMemcpy(c, dev_c, arraySize * sizeof(int), cudaMemcpyDeviceToHost);

cudaFree(dev_c);
cudaFree(dev_a);
cudaFree(dev_b);

printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
    c[0], c[1], c[2], c[3], c[4]);
```

6) Run the CUDA program and observe if it does the work properly without using the method addWithCuda( ) provided in the original template.

Exercise 2. CUDA error checking

When we issue a call to a CUDA function, we may not sure if an exertion is completed successfully. We can do it in the following way.

1) Declare a CUDA error type variable:

```
cudaError_t cudaStatus;
```

For each CUDA function used in the program, namely, cudaMalloc( ), cudaMemcpy( ), cudaDeviceSynchronize(), we can check if each of them is executed successfully without any error by specifying cudaStatus as the return of the function. For instance,

```
cudaMalloc((void**)&dev_a, arraySize * sizeof(int));
```

can be rewritten in the following way

```
cudaStatus = cudaMalloc((void**)&dev_a, arraySize * sizeof(int));
```

If cudaMalloc( ) is executed properly, an error type **cudaSuccess** will be returned, which indicates no errors. Thus, we can use the value of cudaStatus to check, for instance, the memory corresponding to variable dev_a has been allocated properly in the following way:

```
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMalloc failed!");
```

```
        goto Error;
    }
```

2) For each of CUDA function called and executed in the CUDA program, add code to check if they are all executed successfully.

Exercise 3. Check time range.

Knowing how long a kernel takes to execute is essential when assessing the performance your parallel algorithms. There are several different ways to measure and observe kernel performance.

1) Increase the size of array from 5 to a very big number progressively, say, 512, 1024, ….
2) Initialize the two large data arrays.
3) Specify different number of blocks and different number of threads in << < … >> > and observe the performance of your CUDA program. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to **1024 threads** (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities).
4) Create two CUDA events corresponding to the start time and the end time of an execution of a cuda program:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

They can be destroyed in the following way:

```
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

5) Start the timer immediately before the execution of CUDA program and stop the timer immediately after its execution.

```
cudaEventRecord(start, 0);
addKernel << <1, arraySize >> > (dev_c, dev_a, dev_b);
cudaEventRecord(stop, 0);
```

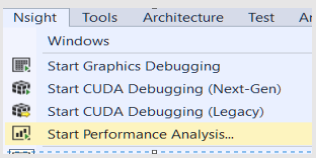
6) Calculate the time elapsed

```
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

7) Print out the elapsed time:

```
printf("Time elapsed the execution of kernal: %fn", elapsedTime);
```

5) You can also observe the time required for launching the kernel function using Nvidia Nsight CUDA performance tool provided in Visual studio 2017.



Exercise 4. Browsing CUDA samples

Go to the C:ProgramData → NVIDIA Corporation→ CUDA Samples folder. There is a list of samples provided in the CUDA package. Have a look at the samples and get a basic understanding at what CUDA can do in general.