Learning how to access data from kernel:

We will use now **threadIdx, blockIdx,blockDim & gridDim** to calculate array indices.

A picture containing timeline

Description automatically generatedLet’s consider an array of **8 elements** with values down below and we will launch a Kernel with ***8 threads*** which are arranged in a ***1D grid with a single block***.

Timeline

Description automatically generated with medium confidenceNow, let’s launch our kernel with **2 blocks** each having **4 threads.** By doing this, **threadIdx** won’t be able to reach all of the threads by its correct location.

Text

Description automatically generatedDiagram

Description automatically generatedText

Description automatically generated with medium confidenceNow, let’s consider a one-dimensional grid with ***4 blocks***. To calculate the index for each thread in the grid, we will use an offset for each thread in each block by using the following formula (TID = threadIdx & GID = thread index we want)

Continuation on how to access data from kernel:

Diagram

Description automatically generatedLet’s consider a ***2D Grid*** with ***16 threads*** arranged in ***2 bocks*** ***in X-dim*** and ***2 blocks in Y-dim***, with each having ***4 threads in X-dim***. We will try to use the formula we used to before to print out an array of values.

As expected, the previous doesn’t access all the threads due to the addition of a new block dimension.

Graphical user interface, application

Description automatically generated with medium confidenceConsider the following. We need new formulas to access all of the following.

Text

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

Timeline

Description automatically generated

Continuation on how to access data from kernel:

Chart, box and whisker chart

Description automatically generatedA screenshot of a computer

Description automatically generated with low confidenceLet’s consider a ***2D Grid*** with ***16 threads*** arranged in ***2 bocks*** ***in X-dim*** and ***2 blocks in Y-dim***, with each having ***2 threads in X-dim*** and ***2 threads in Y-dim***. We will try to use the formula we used to before to print out an array of values. We will use the formula to calculate the ***global\_index*** from the previous example, but we will need to find the row and block offset differently to account to the thread dimension inside the block.

A picture containing icon

Description automatically generatedWith the previous in mind, we will look at the following threat slot map:

A picture containing icon

Description automatically generatedLet’s look at a single block. Previously, we used threadIdx to find out the indexes of each thread because it was 1-dimensional. However, now we have the following:

Graphical user interface

Description automatically generated with medium confidenceThe following is the formula we use to calculate the thread index:

A picture containing polygon

Description automatically generatedNow, we want to find out how to calculate the block offset (**vertical *offset between blocks (left to right***):

A picture containing diagram

Description automatically generatedFinally, we calculate the row (***horizontal offset between blocks (up to down***)) offset (row in blocks):

Memory transfer between host and device in CUDA program:

Remember:

Host code execution 🡪 Happen using the CPU and main memory or RAM

Device code execution 🡪 execute instructions in streaming multiprocessors.

Diagram

Description automatically generatedThe following is the steps of an usual CUDA program:

1. Initialize data from the host side
2. Hand over the computational intense task to the device (GPU)
3. Host execution will continue until it needs back the results from the device (it will wait until device finish its execution)
4. Transfer the results from the device back to the host

Shape

Description automatically generatedFor the previous, we need to perform two memory transfers. After initializing the data, **we need to transfer data from host to device, and then from device to host**.

CPU works with **Host Side Memory**, and the steaming multiprocessors are working with **Device Side Memory**.

We will use the following commands to transfer memory from **host** to **device**:

cudaMemcpy(destination pointer, source pointer, number of bytes we are going to copy, memory transfer direction);

memory transfer direction can be: cudaMemcpyHostToDevice **or** cudaMemcpyDeviceToHost **or** cudaMemcpyDeviceToDevice

Logo

Description automatically generated with medium confidenceIn our main program, there are some things to be aware:

We create a pointer to the integer array from the array created (usually we use ***h\_*** to indicate that this variable is a host variable)

and then we allocate memory using the malloc function. You have to ***provide the total number of bytes to the argument*** of the malloc function and ***it will return a void pointer***. Thus, **you should cast the pointer to the type that your original pointer was** (in this case, **int\***)

Text

Description automatically generatedWe will randomly initialize the array (fill it with numbers)

In the code, we will use a time\_t variable and the rand function to fill out randomly values from 0 to 255 (0xff).

Text, Word

Description automatically generated with medium confidenceNow, we create the device pointer and allocate it with the following code:

Table

Description automatically generatedUse **d\_** in the variable name to indicate that it is a device variable. Then we will use the following commands to allocate memory in the device.

**malloc** 🡪allocate memory

**memset** 🡪set values for a given memory location

**free** 🡪 reclaim specified memory location

For cudaMalloc() function, we need to ***provide a double pointer or pointer to pointer as the first argument***. So we **will cast our device pointer** (int \* d\_input) and then for the **second argument we need to specify the total size of memory being allocated**.

Usually, we may want to launch more threads than the amount of elements we need due to performance benefits. Moreover, we usually must keep our block size to a multiple of 32. By doing this, we will have threads that will do nothing. To avoid this, we will use a conditional statement to restrict any outside the scope computations to the threads we won’t use.