# Assignment 3 Kernel layout and branching impact

# Overview

There are three main files

|  |  |
| --- | --- |
| File Name | Purpose |
| **assignment.cu** | Run kernels on GPU to perform several math operations (elementwise Add, elementwise Subtract, elementwise Multiply, elementwise Mod) between two 2D array. |
| **branchingimpact.cu** | Similar to assignment.cu but the switch case is moved into the kernel function to select which math operation to run based on an input into the kernel. The intension is to see if this will cause any execution speed reduction for each same math operation. |
| **branchingimpact1.cu** | This file created three different kernels to perform similar operation. All the kernels are expected to be run with 32 threads (a warp) in total. All of them are expected to perform 16 adds and 16 subtract operations. However, the branching is different based on index. (see Figure 1 and the code for details). The intension is to see if this kind of conditionally branching will slow down the execution speed of the overall program on GPU. |

|  |
| --- |
| Diagram  Description automatically generated with medium confidence |
| Figure 1 kernel concept in branchingimpact1.cu |

A run.sh has been provided as a sequence of commandline commands to

* Build appropriate executable
* run to couple of tests.

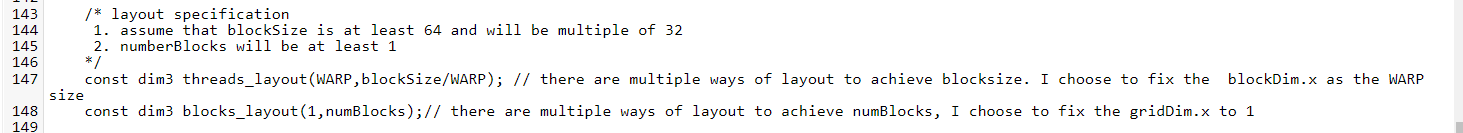
# Tests and Result Discussion

## Test 1: Parallel Computing on GPU, Array elementwise math

assignment.exe 512 256 specifies that

* total number of threads is 512.
* Number of threads per block is 256

In assignment.cu, I set up the threads\_layout and blocks\_layout as shown below



So the blocks layout in the grid is 1xnumBlocks where numBlocks=2.

The threads\_layout in a block is 32x8 since 256/32 gives 8.

|  |
| --- |
| A picture containing diagram  Description automatically generated |
| Figure 2 kernel concept *in* Graphical view of layout |

Figure 2 shows the graphical view of the kernel layout. This block contains 8 Warp with each Warp representing 32 threads. Ideally, each thread will be scheduled to run on a separate streaming processor (SP) with each Warp scheduled to run on a multi-stream processor (SM).

This layout is by no means optimum, it is just my random selection to full-fill the 512 total threads.

Figure 3 shows the result of test1. We can see that Array0 is initialized with elements going from 0 all the way up to total number of threads. Array 1 is initialized with random number among 0 and 3. The kernel execution time is also captured using std::chrono::high\_resolution\_clock::now().

|  |
| --- |
|  |
| Figure 3 Test 1: Parallel Computing- Array mathematics, Add, Subtract, Multiply, Mod |

While the result looks correct, one thing I did observe is that the time takes to run the same kernel will be different each time you call it. I don’t understand why but a future improvement could be made to call the same kernel 1000 times and get the total time or average time of execution.

### Test 1.1: Parallel Computing- Array mathematics with different array size, num of threads

#### Test 1.1.1: 1x512 array and 512 total threads

In this test, we changed the total number of threads from 512 down to 256 from commandline. Notice that due to the fact I statically create the array based on macro definition,

#ifndef ARRAY\_SIZE\_X

#define ARRAY\_SIZE\_X 512 // column of the 2D array// this can be defined in Makefile through commandline overide (-D flag for compiler)

#endif

We actually have to recompile the code with -D option to redefine Macro ARRAY\_SIZE\_X to match the intended total number of threads. If the array is dynamically allocated, this won’t be needed. This can be a future improvement.

|  |
| --- |
|  |
| Figure 4 Test 1.1.1: Parallel Computing- Array mathematics with different array size, num of threads |

#### Test 1.1.2: 1x1024 array and 1024 total threads

A portion of shell script is copy&pasted here for your convenience.

echo "-----------------------------------------------------------------------------------------------------------"

echo "Test 1.1.2: Parallel Computing- Array mathematics with different array size, num of threads :"

echo "-----------------------------------------------------------------------------------------------------------"

# notice that to execute properly, the total number of thread has to match the underline array size. So a recompile with a redefined ARRAY\_SIZE\_X is needed.

nvcc assignment.cu -std=c++11 -DARRAY\_SIZE\_X[=1024] -L /usr/local/cuda/lib -lcudart -o assignment.exe

assignment.exe 1024 256

|  |
| --- |
|  |
| Figure 4 Test 1.1.2: Parallel Computing- Array mathematics with different array size, num of threads |

|  |  |  |  |
| --- | --- | --- | --- |
| Table 1 Summary of execution time with different total number of threads (fixed block size=256) | | | |
| Threads  Operation | 256 | 512 | 1024 |
| Add | 27889 ns | 24719 ns | 25022 ns |
| subtract | 12605 ns | 14349 ns | 15730 ns |
| multiply | 12010 ns | 11523 ns | 18412 ns |
| mod | 14377 ns | 10209 ns | 16502 ns |

From the result shown in Table 1, considering the fact that execution time varies each time you call the kernel, we can conclude that this is true parallel computing because the execution time is on the same order when number of threads increases by 2x and 4x.

### Test 1.2: Parallel Computing- Array mathematics with a different block size

#### Test 1.2.1: 512 threads per block

A portion of shell script is copy&pasted here for your convenience.

echo "-----------------------------------------------------------------------------------------------------------"

echo "Test 1.2.1: Parallel Computing- Array mathematics with a different blocksize :"

echo "-----------------------------------------------------------------------------------------------------------"

nvcc assignment.cu -std=c++11 -DARRAY\_SIZE\_X[=512] -L /usr/local/cuda/lib -lcudart -o assignment.exe

assignment.exe 512 **512**

For simplicity, the bash window will not be captured for this and the following test. An summary table similar to Table 1 will be provided.

#### Test 1.2.2: 128 threads per block

A portion of shell script is copy&pasted here for your convenience.

echo "-----------------------------------------------------------------------------------------------------------"

echo "Test 1.2.2: Parallel Computing- Array mathematics with a different blocksize :"

echo "-----------------------------------------------------------------------------------------------------------"

# notice that to solely changing the blocksize does not require the recompile.

assignment.exe 512 128

|  |  |  |  |
| --- | --- | --- | --- |
| Table 2 Summary of execution time with different block size (fixed total threads=512) | | | |
| Block size  Operation | 128 | 256 | 512 |
| Add | 27971 ns | 26647 ns | 23930 ns |
| subtract | 9010 ns | 9407 ns | 17489 ns |
| multiply | 7491 ns | 7929 ns | 16953 ns |
| mod | 6110 ns | 7258 ns | 16248 ns |

The observation from Table 2 is the block size did not have impact on the execution time until it is increased to 512. I don’t know the logic behand it but my guess is that it might be related to hardware (GPU) architecture. Since a block will be scheduled to a stream multiprocessor (SM) ideally, maybe the SM of underline GPU is not able to support 512 threads per block. It is also interesting that all the add takes longer time. I wonder if it is due to the fact that it is the first kernel gets ran. Maybe some sort of initialization work will be done there.

## Test 2: Branching Effect

### Test 2.1: Branching impact without warp divergence

Based on my current understanding, warp divergence describes the fact that the threads within the same warp needs to execute different instructions based on certain conditions. If this definition is correct, the code in branchingimpact.cu will not cause warp divergence.

In assignment.cu, four separate kernels are created. The switch case based branching is host code. There is zero branching code running on the GPU.

In branchingimpact.cu, there is only one kernel. The kernel will accept an integer input. And if that integer is 0, kernel perform array add, if the integer is 1, kernel perform array subtract and so on so forth. Basically, a switch case is moved inside the kernel which will be run on the GPU.

|  |  |  |
| --- | --- | --- |
| Table 3 Summary of execution time with different executable (both with 512 total threads, 256 blocksize) | | |
| File  Operation | assignment.exe 512 256 | branchingimpact.exe 512 256 |
| Add | 24684 ns | 24113 ns |
| subtract | 9723 ns | 13409ns |
| multiply | 7399 ns | 12068 ns |
| mod | 10160 ns | 9855 ns |

Table 3 shows that moving switch case into the kernel did not affect the execution speed that much.

My suspicion is that when there is no warp divergence, there won’t be too much impact from branching. But I am not sure.

### Test 2.2: Branching impact with warp divergence

|  |
| --- |
|  |
| Figure 6 total execution time of calling three different kernels 1000 times |

Refer to the actual code and Figure 1 to understand the differences among the three kernels. But my basic expectation is that kernel 1 should induce the most amount of warp divergence, the kernel 2 should induce the least amount of warp divergence. So my expectation was that the kernel 1 will takes much longer to run. However, result shown in Figure 6 did not support my thought. So I don’t know why kernel 1, kernel 2, kernel 3 showed almost the same execution speed on GPU. This looks to me that the branching does not have an impact on GPU at all.

# Summary of future improvements

1. Try to understand why kernel execution speed could vary a lot among different calls
2. Due to static array allocation based on macro, each time we change total number of threads, it requires a recompile with an updated macro definition on ARRAY\_SIZE\_X*. Basically* ARRAY\_SIZE\_X\*ARRAY\_SIZE\_Y mustmatch total number of threads specified at command line. Is there a better way to implement this? Dynamically allocates the array sounds like a machine gun, too heavy.
3. Still confused on why the experiment result shows that branching has no impact on kernel execution speed on GPU
4. Separate the C function into .c, C++ function into .cpp, and leave kernel function within .cu and learn how to compile them separately and link together to be a full executable.
5. Pass 2D array as pointers into a print function to print it out