GPU Programming Basics

Northeastern University — *for all* — Julian Gutierrez
NUCAR Laboratory — David Kaeli

Hands-on Lab #2

Objective

- Optimize an implementation of a simple algorithm through better memory management.
- Use available tools to detect any potential places for improvement.

NOTE: Unless stated otherwise, work in pairs.

Part 1: Improving performance

Copy the following folder to your home directory (or folder of your choosing):

```
/scratch/gutierrez.jul/GPUClass/HOL2/DELIVERABLE/
```

The algorithm is similar to the first lab, but in this case, it computes the following formula:

C[i] = (A[i-1]-A[i+1])*B[i] + A[i]*Constant

- If i=0, i-1 = last element on the vector.
- If i=vector_size-1, i+1 = first element on the vector.

The idea is to enhance the performance of the algorithm through the use of shared memory, and any other optimization you can think of. Read the main function code and the basic kernel function, and think of ways to improve it.

Baseline code is the following file:

```
./Baseline.cu
```

File you need to modify is:

```
./Modified.cu
```

Try to answer these questions:

1- CudaMemset: What does this do, and why can it be important?
2- Why set the block size as a define, but the vector size is dynamically assigned?
3- What should we copy to shared memory?
4- What happens if block size is different than vector size?
5- Should we use constant memory?
6- What happens on the borders of the vector? TID = 0 or TID = vector_size-1
7- Do we need to synchronize?

What we need to do to use shared memory:

1- Declare shared memory
    a. What size should the array in shared memory be?
2- Bring in the data from global memory into shared memory
    a. Including border (padding at the beginning and the end of the array)

3- Synchronize
4- Do calculation using the values in shared memory
5- Write output

What else can we do to further improve this?

To compile the code, run the following command:

```
nvcc -arch=sm_35 -O3 Baseline.cu -o Baseline
```

Once you're able to successfully compile and run the code without errors. Try at least the following vector sizes and block sizes.

Vector Size	**Block Size**	**CPU Time**	**GPU Time (basic)**	**GPU Time (improved)**
100	32			
100	128			
100	512			
100	1024			
100 000	32			
100 000	128			
100 000	512			
100 000	1024			
100 000 000	32			
100 000 000	128			
100 000 000	512			
100 000 000	1024			

- What impact does block size have on the performance?
- Does it perform better using shared memory?
- Remember, in GPU time we are including: Memory allocation, memory set, memory copy, kernel runtime, memory copy.
- **Optional**: Try running it for 1 000 000 000. What happens? NOTE: Beware, computing resources might not be enough.

Part 2: CUDA-memcheck

CUDA-memcheck is a very useful tool to check if your code is working correctly. It looks for illegal accesses in memory (accessing an illegal pointer, or when you accessed an array beyond its size, etc).

To run the tool, you do the following:

```
cuda-memcheck <command used to execute the code>
```

When you run it with your code, are there any errors? If so, fix them.

Part 3: NVPROF

As stated in class before, nvprof is a great tool to profile your application and understand what is happening and how to improve it. Use the following code for the next tests.

```
/scratch/gutierrez.jul/GPUClass/HOL2/DELIVERABLE/Nvprof.cu
```

Notice any differences with the previous version?

Compile the code:

```
nvcc -arch=sm_35 -O3 Nvprof.cu -o Nvprof
```

Run code to make sure it's working. Run it with a vector size of 100 000 000 and make sure the block size is set to 512:

```
./Nvprof 100000000
```

Run NVPROF using this command:

```
nvprof ./Nvprof 100000000
```

This will print out a lot of information. Look at the percentages and answer these questions:

- what information do you find useful?
- What insights do you get from this?
- How would you improve this code? What would you improve first?

Now let's look at some of the metrics by running the following command:

```
nvprof -m all  ./Nvprof 100000000 &> nvprof.log
```

This command will show all metrics and save the output into nvprof.log.

Open the file and look at the results.

- Which metrics do you think are useful?
- If you look at gld_efficiency and l2_l1_read_hit_rate, what do you think is happening on the memory? Is there something we can do to improve this?
- How about gst_efficiency?
- Do this analysis for the improved version. (Ask the professor to provide it in case you were unsuccessful in completing the task). Notice the differences in these metrics, and conclude what is happening:
  - GLD efficiency
  - L2 Read Hit Rate
  - L2 Throughput
  - GLD Transactions
  - CF issued
  - Stall Memory Dependency
  - Stall sync
  - Any other
- Conclude: Is it better or not to use shared memory in this algorithm? Why or why not?

Note: Please remember to ask as many questions as possible. We are here to help as much as we can.