GPU Programming Basics

Northeastern University *for all* Julian Gutierrez
NUCAR Laboratory Nicolas Agostini, David Kaeli

Hands-on Lab #2

Objective

- Learn how to use the server.
- Write our first GPU program.
- Profile our program.

NOTE: Unless stated otherwise, work in pairs.

Part 1: How to use the Discovery Cluster

1. We have been granted an exclusive reservation (group of nodes) for us to work with. They are 20 GPU nodes with K20 GPUs.
2. Once we have completed the class, if you wish to continue using the gpus on the discovery cluster, you will have to submit your jobs to the gpu partition (available to all discovery users where it might be slow sometimes depending on how many people are using it).
3. We are first going to setup the environment in the discovery cluster. We are going to store all our codes in a general folder called scratch. Each person **must** create their own folder in scratch by running this command:

```
mkdir /scratch/$USER
```

4. The /scratch directory is a 1+ PB parallel file server which all users may utilize. Always launch submitted jobs from /scratch and transfer your data from /scratch to another resource (e.g. personal/group file server) as soon as possible. For instance, if you wrote code stored in scratch, you want to copy it back to your home directory as the data in /scratch might be removed after a couple of months.
5. In order to run any code on Discovery, you must use SLURM (Simple Linux Utility for Resource Management). The idea is that you ask SLURM for resources (e.g. what type of node, how many cores, how much memory), and then your code will execute once those resources are available. We will go into this later on in the lab.
6. Some common SLURM Commands:
    - sbatch <file name> : this will send the job to the scheduler.
    - srun : If you would prefer to work interactively on a node, you may launch and interactive session with srun.
    - squeue : see what jobs are waiting and running currently.
    - scancel <job id> : remove a running or pending job from the queue
    - scontrol <flags> : find more information about the machine configuration and job settings
    - seff <job id> : report the computational efficiency of your calculations
7. We also want to setup our environment to have access to the tools we will use (compilers, etc) (In case you haven't done so from our previous lab). You can do this by loading the necessary modules (cuda). You can add the following lines at the end of your ~/.bashrc file (in your home directory). This is a file which will be loaded every time you connect to the discovery cluster.
    a. Open file using an editor

```
vim ~/.bashrc
```

    b. Go to the end of the file and copy these lines.

```
module load cuda/9.0
```

c. Reload the file.

```
source ~/.bashrc
```

d. You can confirm you did a good job by using the following command. It should display which modules you have loaded, and cuda/9.0 should be one of them.

```
module list
```

8. Each one of you will initiate an interactive (1-core non-exclusive) session on the general partition nodes just to compile and modify the code (remember, we don't want to use the login nodes for this).
9. To request an interactive node, we can use the following command. Once a resource is available, you will be connected automatically into the compute node.

```
srun --pty --export=ALL --partition=general --tasks-per-node 1 --nodes
1 --mem=2Gb --time=02:00:00 /bin/bash
```

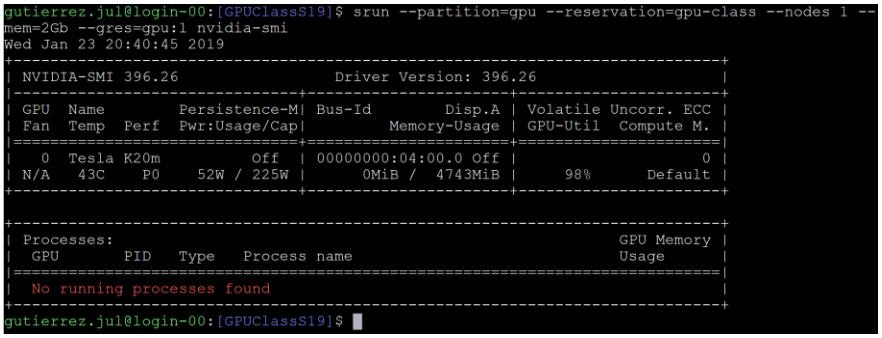
10. You can run the squeue command to guarantee that you are using a compute resource. This will print the nodes allocated for the user:

```
squeue -u $USER # Print the nodes that #USER has reserved
JOBID   PARTITION     NAME     USER ST      TIME  NODES NODELIST(REASON)
678595    general     bash gutierre  R      0:48      1 c2133
```

11. From now on, you can compile your work.
12. Remember, the node that we are connected to doesn't have a gpu, but a way to detect if the gpu is there is by running the "nvidia-smi" command (this prints general information about any cuda device in the machine). We are going to run this command in a node with a GPU in it. Notice the --gres=gpu:1 in the command, this means we are requesting a GPU resource from the node as well:

```
srun --partition=gpu --reservation=GPU-CLASS-SP20 --nodes 1 --mem=2Gb
--time=00:00:10 --gres=gpu:1 nvidia-smi
```

13. The output should look similar to this. Note that sometimes the command can take some time before it completes:



When you are done with everything, type the following command to make the resources available to other users.

```
exit # this will take you to the gateway computer again
```

After completing all steps of this lab (Part 1, Part2, Part 3, Part 4): At the login node, consider verifying that no jobs are running and/or cancel any job that is running:

```
squeue -u $USER # Print the nodes that #USER has reserved
JOBID  PARTITION    NAME     USER ST     TIME  NODES NODELIST(REASON)
678595   general    bash gutierre  R     0:48      1 c2133


scancel 678595   # Cancel the job with id 678595
```

!!!! Please, Remember to **ALWAYS** release the nodes. Always !!!!

Part 2: Writing our first GPU Program

Copy the baseline for our code from the following directory either to your home directory in the server or into your machine (we will be editing this file).

Using the following commands as a guideline (feel free to take a different approach):

```
cd /scratch/$USER/
mkdir GPUClassS19
cd GPUClassS19
cp -r /scratch/gutierrez.jul/GPUClassS19/HOL2/ .
cd HOL2/
vim VectorAdd.cu
```

The code we will be working on is meant to do a vector add on the CPU, do the same vector add with the GPU and compares the results at the end.

1. Open the file. Find the comments "HERE" and fill the code with the necessary commands to make it work.
2. The following are the templates for the commands that you'll need (based on what we saw in the previous class):

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count,
cudaMemcpyKind kind )
      kind: cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost
kernel_name <<<grid_size, block_size>>>(argument list);
cudaFree ( void* ptr);
```

3. Look at the kernel function and the arguments it receives. Write the code to calculate the vector add:
   a. Ci = Ai + Bi
4. To compile the code, run the following command:

```
nvcc -arch=sm_35 -O3 VectorAdd.cu -o vAdd
```

5. Once your code is compiling without errors, we can run the code. To run the code, we must submit a job to use the GPU. The script we will use is the one in the folder named exec.bash. Look at what it does and update the fields reservation and gres to use the resources available to our class (you will have to do it for every .bash file in this class):

```
vim exec.bash
```

```
#SBATCH --reservation=GPU-CLASS-SP20
```

```
#SBATCH --gres=gpu:1
```

6. Make sure the directory is correct, then submit it using the following command:

```
sbatch exec.bash
```

7. Depending on how many people are running at the same time, this could take a couple of minutes, though hopefully it will run almost immediately. You can use the following command to check if the status:

```
squeue -u $USER
```

8. Once it has run, open the output files named "exec.out". Make sure it gives the correct result. If there are any errors, fix them. You can use CUDA-MEMCHECK (next section) to debug.

a. Observe that there are some commands that are preceded by a +. This is just to indicate the command that was executed.

9. Take the following observations into account and think of the answers to these questions while testing the code.
    a. Grid Size and Block Size should be chosen based on the variables already available.
    b. Notice cudaMalloc receives a **variable. What does this mean?
    c. Why are we allocating an array called c_gpu on the CPU?
    d. Test different vector sizes and block sizes to see if the result is still correct.
        i. Compare the performance when you increase the block size and when you decrease it.
        ii. Do this by modifying the exec.bash script and adding more lines to execute different scenarios. E.g.:
            1. ./vadd 10000000 32
            2. ./vadd 10000000 64
            3. ./vadd 10000000 128
            4. ./vadd 10000000 256
            5. ./vadd 10000000 512
            6. Submit the script again and wait for it to complete
        iii. What could be affecting the behavior?
        iv. Does GPU perform better than CPU? Try different vector sizes (100, 100 000, 100 000 000)
        v. Try running using different block sizes for a big vector. How does that affect performance? Remember the 1024 maximum threads per block limit.
        vi. Should the execution time include the time it takes to allocate and copy data?
10. Was it easier to implement the GPU version compared to writing an equivalent program on the CPU?

NOTE: You can check for the status of the nodes in the partitions by using the following commands:

```
sinfo -p gpu
squeue -p gpu
```

Part 3: CUDA-MEMCHECK

CUDA memory checker 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). In addition, it will let you know if there were any issues with any Cuda command.

To run the tool, you use the following command:

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

Use the memcheck bash script to submit a job to run cuda-memcheck with your executable. When you run it with your code, are there any errors? If so, fix them.

```
sbatch memcheck.bash
```

Part 4: NVPROF

NVPROF is a great tool to profile your application and understand what is happening and how to improve it. Use the executable you have compiled and works correctly to do the following tests.

We will use the nvprof.bash script to submit a job that runs nvprof. Make sure the run is using a vector size of 100 000 000 and the block size is set to 512.

```
sbatch nvprof.bash
```

As an example, this is the command the script should execute to run NVPROF:

```
nvprof ./VectorAdd 100000000
```

This will print out a lot of information. When you submit the script and it executes, look for "NORMAL RUN" results. 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?
- Notice that the code runs slower with the profiler, why is that?

Now, another way of running nvprof is using the following command (the script runs it as well):

```
nvprof --print-gpu-trace ./VectorAdd 100000000
```

- From the previous run outputs, look for "GPU TRACE". Any useful information displayed running it like this?
- Imagine your code was much more complex. Would this information give you an insight as to how your code is behaving?

Now, the final command we used in the script is to recollect performance metrics. Look for "METRICS" in the same file output. Look at some of the metrics by running the following command (It will calculate all of the metrics):

```
nvprof -m all  ./VectorAdd 100000000
```

Open the output 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?
- There are many metrics. Try to find which ones are the most useful.
- If you want to compare different runs, try setting the block size to a small value (32) and collecting the metrics, and then run it with a larger value (1024).
    - Can you notice any differences in the metrics? In the speed?

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