GPU Programming Basics

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

Hands-on Lab #3

LAB #3 - Objective

- Test the impact of doing coalesced reads in global memory for a vector add application.

Part 1: Global Memory

First of all, you need to open 2 terminals and connect to discovery cluster.

- Terminal 1: this terminal will be used to connected to an interactive compilation node.
- Terminal 2: this terminal will be used to schedule the gpu jobs, it will always be connected to the login node.

In Terminal 1, to request an interactive node for code development and compilation, we can use the following command. Once a resource is available, you will be connected automatically into the compute node. If you are having trouble with this, go back to Lab 2 and review part 1.

```
# TYPE this command, do not copy and paste

srun --pty --export=ALL --partition=short --tasks-per-node 1 --nodes 1
--mem=2Gb --time=02:00:00 /bin/bash # Reserve the compilation node
```

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

```
cd /scratch/$USER/GPUClassS19
cp -r /scratch/gutierrez.jul/GPUClassS19/HOL3/ .
cd HOL3/
```

NOTE: Remember you will get an error trying to copy the solution. You can ignore this error.

The algorithm we are implementing is a vector add using doubles. The objective of this part is to enhance the performance of the algorithm through efficient global memory usage. Read the main function code and the basic kernel function from the following file:

```
./vadd.cu
```

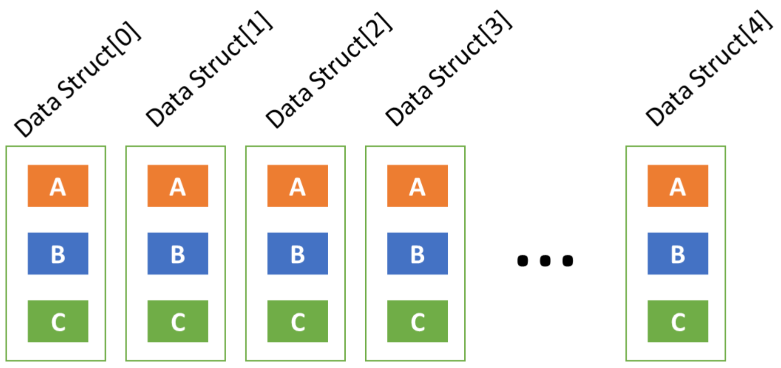
The file you will be modifying is:

```
./vadd_coalesced.cu
```

To compile the code, run the following command:

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

As you can see from the baseline code, the data is accessed using a struct. In this baseline version, we have created an **array of structs (AOS)** for the data arrays. Each struct will contain the data for the ith element, including both inputs and the output as shown in the following figure.



1. Run the baseline code (vadd) for a block size of 512 with a large vector size (E.g. 100 000 000) by submitting this job using the exec.bash script.
    a. 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
```

    b. After the file is adjusted, you can schedule it to using the LOGIN NODE (second terminal).

```
sbatch exec.bash
```

    c. Once it has run, open the output files named "exec.<jobid>.out".
    d. Make sure everything is working.
    e. NOTE: The parallel execution time is how long it takes the GPU to do the vector add while the sequential execution time is how long it takes the CPU to do the vector add.
    f. Write down the execution time for the GPU kernel.
2. Run nvprof to recollect certain metrics. We can submit the nvprof.bash script to do this:

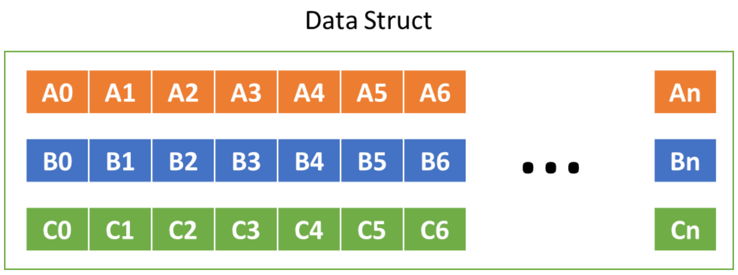
```
sbatch nvprof.bash
```

3. Open the nvprof.bash file to see which metrics it is recollecting and analyze these metrics.
    a. What does the gst_efficiency and gld_efficiency tells us? What does that percentage represent? You can use the internet to get a better understanding of these metrics.
    b. How would you rate the performance of the memory for this implementation?
    c. What should we look for in these metrics if we want to improve the performance?

Once you have familiarized yourself with the baseline code, we will proceed to do changes to the `vadd_coalesced.cu` code.

Follow these steps to restructure the code to better coalesce the memory reads and writes in global memory.

1- We need to do a change in the structure of our arrays. We need to create a **struct of arrays (SOA)** (as shown in the following image) and in order to do so; we now need to declare the variables inside the struct as pointers so that we can later allocate them in global memory.
    a. Inside the struct data at line 7, convert the variables to pointers.



2- Given that we are now accessing the data struct as a single struct with arrays inside, change the way we are indexing the data inside the kernel, by moving the indexes to the correct pointers at line 25.

3- Allocating the data is now more complicated, as we need to independently allocate each array inside the struct. To do so, we need to allocate it in the CPU and the GPU. This has been done for the data that will reside on the CPU (lines 58-64).

   a. Modify the code at 76 to do the same allocation for the GPU using cudaMalloc.
   b. This means we are no longer using data_gpu as a pointer. Similar to how it is used on the CPU (line 58), we need to declare data_gpu as a struct and then allocate the memory for each array separately, using the pointers inside data_gpu.
   c. We are passing the struct to the GPU and the kernel will access the pointers inside the struct.
   d. NOTE: The size of the allocations that you are doing has changed. Now you are allocating only individual elements, not the whole data structure.
   e. This means we want to create a struct for the pointers on the GPU and then allocate the memory space for each one of these pointers, similar to what was done on the CPU starting at lines 58.
   f. You will need 3 cudaMalloc commands.
   g. NOTE: Notice that a struct named data_gpu_on_cpu has been created to copy the result from the GPU to the CPU.
   h. NOTE: Notice that the indexing in the initialization input part has been modified to work appropriately. This can be helpful for the kernel indexing part.
   i. NOTE: Remember the size of the data we want to allocate. Same goes for the copies of the data.

4- Copy the input data from the CPU to the GPU using cudaMemcpy.

   a. At line 81, make sure that you are using the appropriate pointers when doing the copy.
   b. You will need two copies, one for copying a and one for copying b.

5- Copy the output data from the GPU to the CPU.

   a. At line 119, make sure you are also using the appropriate pointers.
   b. You will need one copy for c. Remember, which variables are the pointers to the data.

6- Notice the Error checker is also using the appropriate indexing of the data.

7- Free the gpu pointer data using cudaFree.

   a. At line 141, make sure you deallocate the arrays in the GPU as good practice.

To compile the new version (Make sure no compilation errors happen):

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

To run cuda-memcheck on the new version (Make sure all accesses are correct and output is correct):

```
sbatch memcheck_coalesced.bash
```

To run nvprof on the new version:

```
sbatch nvprof_coalesced.bash
```

To just run the new version:

```
sbatch exec_coalesced.bash
```

Once you're able to successfully compile and run the code without errors, including cuda-memcheck, measure the performance by running the application stand alone and comparing the execution time to the baseline. Also, run the nvprof script for this code (using the appropriate bash script). Do you see any differences in the metrics? Try explaining why.

In the development of the code, try to answer these questions:

1- Why set the block size as a define, but the vector size is dynamically assigned?
2- What happens if block size is different from the vector size?
3- Should we use constant memory?
4- What else can we do to further improve the performance?
5- Visualize how the data is stored in memory when using SOA or AOS. This will help understanding the performance differences.
6- Which way would work best on a CPU?
7- Which version is easier to implement?

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