GPU Programming Basics

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

Hands-on Lab #6

LAB #6 - Objective

- Optimize an implementation of a simple algorithm through pinned memory.
- Use unified memory to convert a CPU code into a GPU code.
- Study the impact of Dynamic Parallelism on an algorithm.

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/HOL6/ .
cd HOL6/
```

Part 1: Pinned Memory

CD into the pinned folder to work on this part of the lab:

```
cd pinned/
```

The code inside pinned folder is the solution to the global memory lab. The idea is to modify this code to use pinned memory instead of regular C memory allocations. The `vadd` and `vadd_pinned` files are the same. The following are the templates for the commands you need:

```
cudaError_t cudaMallocHost ( void** ptr, size_t size )
cudaError_t cudaFreeHost ( void* ptr )
```

To compile the baseline code:

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

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
```

Now you can use the command:

```
sbatch nvprof.bash
```

Look at the time it takes to copy the data between the host and the device and vice versa.

Modify the `vadd_pinned` file where the "HERE" marks are to use cudaMallocHost instead of regular C++ allocations (new) for the arrays where we will be copying data to and from the host. Remember to also change the free command to the cudaFreeHost to ensure the arrays are freed correctly.

To compile the code, use the following command:

```
nvcc -arch=sm_35 -lineinfo -O3 vadd_pinned.cu -o vadd_pinned
```

Once the code is compiling, profile the code by launching this script:

```
sbatch nvprof_pinned.bash
```

Compare the performance of the code with and without pinned memory. Answer the following questions.

- What is the most impacted operation?
- Is this beneficial?
- Test it for different sized vectors (as in the following table).
    - To do this you need to modify the .bash scripts and submit them again for every test.
    - You can add multiple execution commands for different vector sizes similar to what we did in the previous lab:

Vector Size	**Block Size**	**CPU Time**	**GPU Time (basic)**	**GPU Time (improved)**
100	512			
100 000	512			
100 000 000	512			
1 000 000 000	512			

- What can happen if the RAM memory on the CPU is full with pinned memory?
- What happens if you don't perform the proper freeing of memory?
- Run nvprof to recollect the metrics and compare the bandwidth metrics (add --metrics all to the .bash and resubmit). Any changes?

Part 2: Unified Memory

In this part we will explore how practical unified memory really is. In order to do so, look at the baseline code found on the unified folder.

```
cd ../unified/
```

1. The baseline code is the file main.cpp. As you can see, it is a very simple C++ code. It receives the vector size as input and initializes the input and output vectors and then prints the first 5 elements of the vector add result. No timing measurements are done here.
2. To compile this code use the following command:

   ```
   g++ -O3 main.cpp -o cpu
   ```

3. You can run this code on your compute node by using the following command:

   ```
   ./cpu 1000000
   ```

4. Remember the output from this command (just to corroborate the first 5 elements are correct).
5. Your job is to modify this code to execute on the GPU. You will use the main.cu file to do the modifications (they are the same, with the only difference being we are loading the cuda.h header file).
6. First, we will convert the "initialize" function into a kernel.
   a. To do this, we need to declare the function as a kernel by using __global__. This should go before the void (return type of the function).
   b. Next, we need to remove the for loop as each thread will now be doing a single element.
   c. Replace the for loop with a declaration for the value of i:

      ```
      int i = <>
      ```

   d. "i" should be equal to the unique thread index similar to what we have used in previous codes.
7. Do the same for the "addVectors" function.
8. In the main function we want to allocate the arrays using CUDA's unified memory functions, which means the CUDA driver will be in charge of handling where the data should be (CPU or GPU).
9. We need to use the following template function:

   ```
   cudaMallocManaged ( void** devPtr, size_t size)
   ```

10. Replace the CPU allocations with the CUDA malloc managed function.
11. Next step is to declare the size of the block and grid that we will be using.
12. Add 2 lines of code after the allocations declaring 2 dim3 variables that we will use for the kernel calls. You can set the block size to any value but I recommend 512 or 1024. Remember to declare the size of the grid based on the block size and the size of the vector (N).

    ```
    dim3 block(<size>);
    dim3 grid(<size>);
    ```

13. Now, we have to use the invocations of the "initialize" and "addVectors" function as our templates for invocating the new kernel functions. To do this, we need to add the <<<grid,block>>> part into the declaration. Remember that these functions are now kernels, so we need to indicate how many threads we will use.
14. Finally, instead of freeing the allocations using the C template "free", we need to free the memory using "cudaFree".

15. You are all set. Compile the code using the following command:

    ```
    nvcc -arch=sm_35 -lineinfo -O3 main.cu -o gpu
    ```

16. Run the code on the GPU by submitting it using the following script (remember to run the code on the GPU we must submit a script):

    ```
    sbatch exec.bash
    ```

17. Look at the results, are they the same as the output from the CPU? Yes or not?
18. You might run into a key problem. Remember what we discussed in class before that kernel invocations are asynchronous, meaning when the CPU runs the kernel call command, it will tell the GPU to do that and move on immediately. This means that in our code, the printf statements will execute before the kernel has even started on the GPU! What is this creating? Do you get an output or an error?
19. To avoid this problem, we need to add one last command. We need to synchronize after the kernel invocations to wait for the kernels to complete before we move on. Add the following function call after the kernel calls, and before the for loop with the print statements:

    ```
    cudaDeviceSynchronize();
    ```

20. Recompile and resubmit the script to run on the GPU. Make sure you get the correct result.
21. Compare the differences between writing the code for CPU and for the GPU. Notice anything interesting?

Part 3: Dynamic Parallelism

In this part we will explore how to implement a program using Dynamic Parallelism. In order to do so, look at the baseline code found on the dp folder.

```
cd ../dp/
```

NOTE: Remember to make sure you are in a compute node by running the srun command from the beginning.

1. Notice the size of all the data (E.g. lines 81-84). When passing the number of matrices and matrix size to the program, we have to be careful of how much data can actually fit into the GPU.
2. Observe what the algorithm is trying to do. This is a synthetic benchmark (doesn't have real purpose more than to measure performance).
3. Notice certain parts of the code that have omp on them.
    a. What is OMP?
    b. What do these code snippets do?
4. Compile the code as it is:
   ```
   nvcc -arch=sm_35 -lineinfo -O3 baseline.cu -o baseline
   ```
5. And run it using this script (100 matrices, with size 3000x3000):
   ```
   sbatch exec.bash
   ```
6. Look at the time it takes to run each stage.
7. Compile with omp flags.
   ```
   nvcc -arch=sm_35 -lineinfo -O3 baseline.cu -o baseline -Xcompiler "-fopenmp"
   ```
    a. What does -Xcompiler do?
    b. What does -fopenmp do?
8. Resubmit the script. Did it Help? If so, where, why?
    a. It might have had an impact in performance on one of the time measurements.
    b. This is because OMP is spawning 8 threads, and the script we are using to submit the job to the GPU node is only requesting 1 cpu-core (default). If we want to use more cpu cores to speed up the process of initializing the data, we need to modify the batch script and add the following line:
       ```
       #SBATCH --tasks-per-node 8
       ```
    c. This will allocate 8 cpu cores for our job and now we can use them to speed up the initialization.
    d. Why does running 8 threads on 1 CPU run slow?
9. Why do we want to mix OMP and CUDA together?
10. Copy file and name it Modified.cu
    ```
    cp baseline.cu modified.cu
    ```
11. Write a kernel function to implement the second loop. This means we are going to replace the first loop inside the kernel calculation, with another kernel invocation.
    a. Create a new kernel function named `second_calculation`.
    b. This second calculation should do all the same work done inside the for loop. (hint: you can basically copy the same code into the new function).
    c. What should be the index used for the threads executing this loop?
    d. Think about the arguments you need to pass to this new kernel function.
    e. Consider what should the size of the grid and the block be?

    i. Use the same block_size as the others.

f. Compile the code.
```
nvcc -arch=sm_35 -O3 modified.cu -o modified -Xcompiler "-fopenmp"
```
g. It should give a compilation error. Why could this be?

h. If you use the flag -rdc=true, does that fix it? What does -rdc do?

i. Compare results to the original implementation (Use the table below as guideline).

j. Run the code using the following script:

```
sbatch exec_mod.bash
```

k. Think of the advantages of using Dynamic Parallelism.

l. Are there any scenarios where it performs better than the sequential (compare kernel time)?

m. **IMPORTANT:** What is the impact of the number of matrices with DP. Compare the results when changing from 100 to 1000, and from 2000 to 5000. Is there a sweet spot?

    i. Modify the exec_mod script to run all of the tests below in a single launch to speed up the process.

n. **Find the reason you get these results.**

o. How does the block size affect? Try changing it to find a better one.

p. What are your conclusions for Dynamic Parallelism.

q. How would it compare if all the loops were assigned as one single kernel call, and the if statements inside it?

    i. Optional. Implement it.

12. Optional: Write a kernel function that implements the 3rd loops (2 functions).

    a. Does this achieve better performance.

13. Optional: How could we handle data arrays bigger than what can fit in the GPU.

Compare results guideline.

- Run these tests taking in mind the memory size of the GPU (it could cause the program to crash)

Number of Matrices	Matrix Size	Sequential (Approx.)	Baseline		DP with 1 loop	
			All GPU	Kernel	All GPU	Kernel
100	100					
100	1000					
100	3000					
1000	1000					
2000	750					
5000	500					

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