APPLIED GPU PROGRAMMING

ASSIGNMENT 4
GROUP 3

Alejandro Gil Ferrer
Jaime Vallejo Benítez-Cano

The code of the following exercises is in this public gitHub repository:
https://github.com/JaimeVBC/AppliedGPUProgramming.git

Exercise 1: Hello World!

List and explain your extensions to the basic template provided by the course.

1. **KERNEL**

```
const char* mykernel =
"__kernel                                   \n"
"void hello ()                              \n"
"{int index[6] =
{get_local_id(0),get_local_id(1),get_local_id(2),get_group_id(0),get_group_id(1),get_group_id(2)};                                  \n"
"printf(\"Hello World! My thread is (%d,%d,%d) within the block (%d,%d,%d).  \\n \",index[0],index[1],index[2],index[3],index[4],index[5]); }
\n";
//";
```

This is the final implementation of our kernel. It has had variations throughout the exercise. This version is for 3D arrays but it has been barely the same for the previous implementations.
We use the array “index” to store every local and group id, using the standard function get_local_id() and get_group_id(). As we are working with 3D arrays, each one has 3 coordinates.
Finally, the information is printed out in the last line of code.

2. **Rest of the extensions (Where it said “Insert your code here”)**

   *cl_program program = clCreateProgramWithSource(context, 1, (const char**)&mykernel, NULL, &err);*

   Create a program object for the context. The program objects in this case are found at the string with our kernel defined at the beginning of the exercise.

err = clBuildProgram(program, 1, device_list, NULL, NULL, NULL);

This line compiles the program in the device.

```
if (err != CL_SUCCESS)
{
    size_t len;
    char buffer[2048];
     clGetProgramBuildInfo(program, device_list[0], CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
    fprintf(stderr, "Build error: %s\n", buffer);
    return 0;
}
```

These lines catch the errors if there are any.

cl_kernel kernel = clCreateKernel(program, "hello", &err);

This line creates a kernel object

```
size_t n_workitem[3] = {4,4,4};
size_t workgroup_size[3] = {2,2,2};
```

These are the dimensions of the grid, and the dimensions of each block inside the grid.

err = clEnqueueNDRangeKernel(cmd_queue, kernel, 3, NULL, n_workitem, workgroup_size, 0, NULL, NULL);

This line launches the kernel.

```
err = clFlush(cmd_queue);
err = clFinish(cmd_queue);
```

Finally, this code finishes everything and waits for the command queue to finish.

Just a final comment, we added the macro CHK_ERROR(err) call after every method that could encounter an error.

Exercise 2 - Performing SAXPY using OpenCL

Explain how you solved the issue when the ARRAY_SIZE is not a multiple of the block size.

To solve the issue, we launch one block more rounding up the number of blocks needed (except ARRAY_SIZE is multiple of the block size) using the following formula:

(ARRAY_SIZE + workgroup_size - 1) / workgroup_size;

In the kernel, we specify a condition where a thread will exit the kernel if its “id” is higher than the ARRAY_SIZE. The condition is the following one:

if(index < array_size_aux)

If you implemented timing in the code, vary ARRAY_SIZE from small to large, and explain how the execution time changes between GPU and CPU.

ARRAY_SIZE	GPU	CPU
1000000 (10^6)	0 ms	3.854 ms
5000000 (5*10^6)	2.992 ms	16.921 ms
10000000 (10^7)	5.40 ms	31.779 ms
50000000 (5*10^7)	28.496 ms	164.416 ms
100000000 (10^8)	49.073 ms	328.806 ms
500000000 (5*10^8)	242.828 ms	1639.889 ms

As it can be seen in the table, GPU is always faster than the CPU for values over 10^6. Maybe this is not true on a really small scale. The CPU is faster as it does not need to move the data from one way to the other. However, at a big scale, GPU is always more effective.CPU times seem to be about over 6 times slower than GPU. This does not fit mathematically since each execution can vary for really small periods of time, but it is an approximation.

Bonus Exercise - OpenCL Particle Simulations

1. **and 2. Measure the execution time of the CPU and GPU version, varying the number of particles.**

NUM_ITERATIONS = 1000

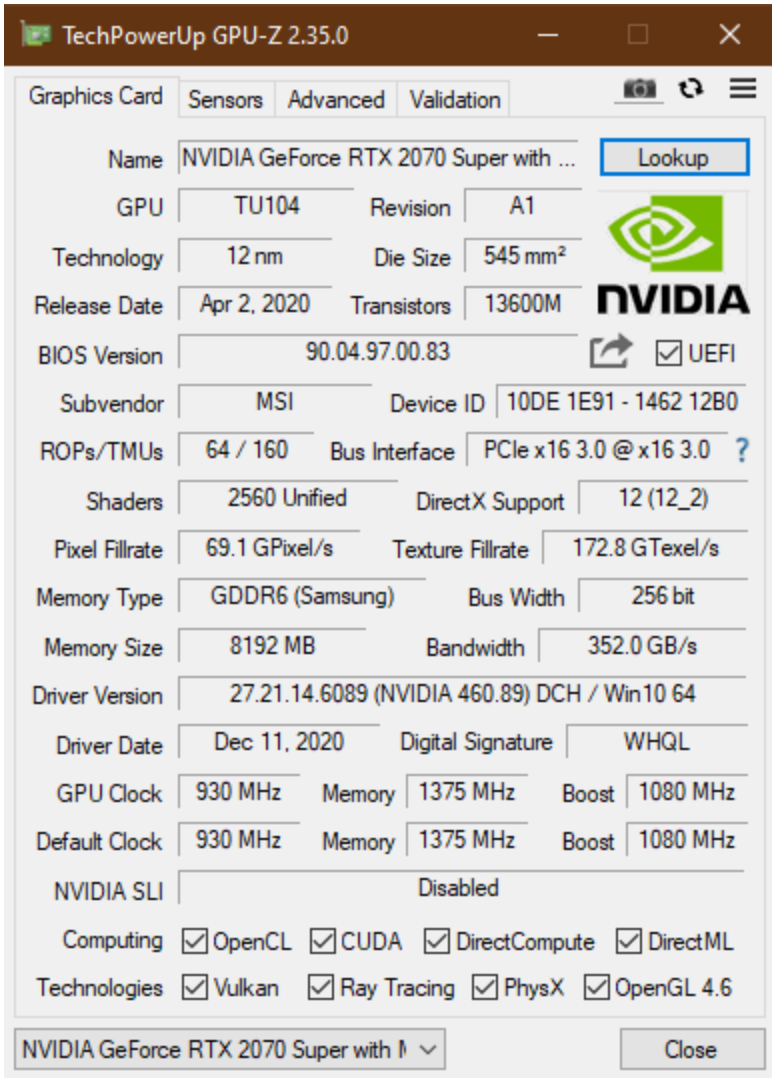
ARRAY_SIZE (number of particles)	GPU (16TPB)	GPU (32TPB)	GPU (64TPB)	GPU (128TPB)	GPU (256TPB)	CPU
100 (10^2)	12.965 ms	10.967 ms	10.986 ms	11.968 ms	11.883 ms	1.994 ms
1000 (10^3)	11.964 ms	10.969 ms	12.936 ms	13.806 ms	11.964 ms	21.736 ms
10000 (10^4)	11.98 ms	13.613 ms	14.959 ms	11.877 ms	14.327 ms	222.479 ms
100000 (10^5)	23.114 ms	17.93 ms	24.738 ms	24.258 ms	24.944 ms	2250.558 ms
1000000 (10^6)	220.778 ms	218.351 ms	227.448 ms	229.062 ms	229.473 ms	23133.717 ms
10000000 (10^7)	2148.594 ms	2146.099 ms	2156.142 ms	2261.932 ms	2174.151 ms	231462.963 ms

3. Generate one or more performance figures based on your measurements in 1 and 2. Include it in the report with a description of the experimental setup (e.g., GPU used) and the observations obtained from the results. Explain how the execution time of the two versions changes when the number of particles increases. Which block size configuration is optimal?

We are not sure that we have understood the question but what we conclude from the results in 1 and 2 is that for values around 100 elements, CPU performs better since GPU memory transfer operations take more time than the calculus itself. From 1000 and higher numbers of elements, we can observe that GPU performance is always faster than CPU performance.

We can also guess that the best thread block configuration is 32 TPB (Threads Per Block) in our case.

The GPU description is shown in the next picture.

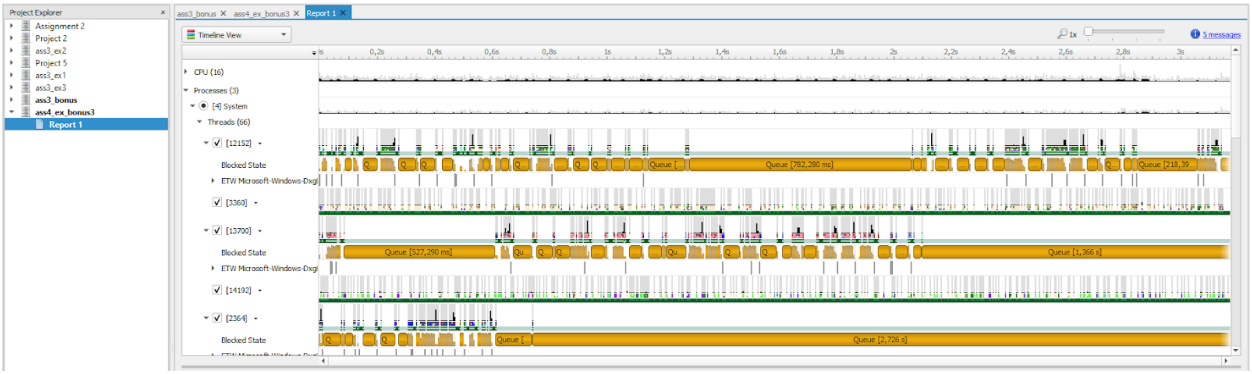


4. Currently, the particle mover is completely offloaded to the GPU, only with data transfer at the beginning and end of the simulation. If the simulation involves CPU dependent functions (i.e. particles need to be copied back and forth every time step), would your observation in 3) still hold? How would you expect the performance will change in terms of GPU execution? Make an educated guess using the GPU activities timing provided by nvprof.

We cannot have really checked the GPU memory transfer operation with the nvprof. We got the profile but it is not really clear for us how much time it takes to transfer the memory from the CPU to the GPU and vice versa.

However, our guess is that our observation in 3 would not hold, since GPU access memory from the CPU can be higher than the calculus time, overall if this memory transfer has to be performed once per iteration. This would end up in such a larger time for the whole performance.

To solve this, having bigger blocks, that is to say, more threads per block, could solve this issue since it could decrease the number of accesses to the GPU memory.



Exercise 4 - Discuss, Reflect, Contrast

- **Reflect on the useability and ease-of-use of OpenCL versus Nvidia CUDA.**

After several exercises using both CUDA and OpenCL, we came to the conclusion that both are easy to use. Once we started to understand the concepts, programming just became a matter of syntax. The differences between both programming languages are something to take into consideration, but we did not have much trouble writing OpenCL code after having learned CUDA.

We can say the main difference is that OpenCL is much more verbose. Just to run a simple program, the devices and platforms of the system have to be identified, which are 3 lines of code each. Once it is done, the user has to initialize plenty of objects such as the context, the command queue, the program... and provide a lot of instructions too, like building the program, catching errors, setting the arguments of the kernel, flush, finish… In addition, we did not like that the kernel code had to be written in a string because it meant that we had to be writing “” and \n in each line of code.

Memory creation, kernel launch or waiting instructions are some of the code that both OpenCL and CUDA have in common.

Unlike CUDA functions, standard functions in OpenCL have so many arguments. It is sometimes not very intuitive to fill the arguments properly. We don’t really like that every program starts with the same lines of code (identify devices etc) because it feels too verbose. Not hard, but tedious. This happens even more when we just have programmed in CUDA, where most of the code is directly related to the exercise itself.

However, we did like to have learned the basics of both languages. Although we would go for CUDA, OpenCL is useful not only in NVIDIA GPUs, but in any other GPU. This would probably make OpenCL programmers even more valuable in some occasions.

To conclude, we wanted to add that it has been a great course, very useful. In addition, we are planning to continue with high performance computing in the near future. Hopefully, using some of the concepts learnt in our final degree project.

- **Contrast the performance of OpenCL contra Nvidia CUDA. Was the performance of your written the same, or did you find any anomalies? If you did find any anomalies, what could have caused these?**

TABLE SAXPY 64 TPB

ARRAY_SIZE	GPU OpenCL	GPU CUDA
100.000	0.000 ms	0.000 ms
1.000.000	0.997 ms	0.000 ms
10.000.000	4.987 ms	0.985 ms
100.000.000	48.787 ms	7.978 ms

TABLE PARTICLES CUDA 512 TPB

NUM_ITERATION: 1000

NUM_PARTICLES	GPU OpenCL	GPU CUDA
10.000	21.892 ms	14.958 ms
100.000	27.925 ms	43.883 ms
1.000.000	219.244 ms	259.317 ms
10.000.000	2157.586 ms	2056.501 ms

In both tests, it can be seen that CUDA implementations always take less time to perform than OpenCL. We guess that this occurs because OpenCL is translated into CUDA when a NVIDIA GPU is being used, as was mentioned in the lectures.