APPLIED GPU PROGRAMMING

ASSIGNMENT 3
GROUP 3

Alejandro Gil Ferrer
Jaime Vallejo Benítez-Cano

The code of the following 4 exercises is in this public gitHub repository:
https://github.com/JaimeVBC/AppliedGPUProgramming.git

Exercise 1: CUDA Edge Detector using shared memory

a) **Explain how the mapping of GPU thread and thread blocks (which is already implemented for you in the code) is working.**

The BLOCK_SIZE is set to 16 so each bloch has 16*16 = 256 threads per block.
The width and the height of the image should be multiple of BLOCK_SIZE because the grid is obtained dividing them by BLOCK_SIZE.
This configuration is used because each thread will calculate only one pixel of the image. The kernel is launched with the same number of blocks that the grid has, and the id of the thread determines which pixel from the image will be operated.

b) **Explain why shared memory can (theoretically) improve performance.**

When we use shared memory, only the first thread of the block copies all the information from the image to the block and after that, each thread finds its neighbours inside the shared memory (8 accesses each).
If we do not use shared memory, those accesses would be inside the general memory, causing loads of simultaneous accesses in the same memory. We know that most of the pixels have their neighbours in their block, so... why finding them in the general image when they can find them quicker in the shared memory?

c) **Explain why the resulting image looks like a "grid" when the kernel is simply copying in pixels to the shared block. Explain how this is solved and what are the cases.**

It is true that the kernel is copying pixels but it is copying less pixels (14x14) than before (16x16), leaving the last two rows and columns of each block “free” causing a black grid in the image. This happens because the algorithm we want to use, accesses the pixels that are two columns at the right and two rows below from each pixel that is being processed.

If we had processed all the pixels in the block (16x16), the kernel would have exceeded the memory trying to read values over the limits, that is to mean, read access violation. This happens because those values won't be in the same block as the pixel that is being processed.

For this reason, we use a shared memory, bigger than the block size, as we add another two extra columns and two extra rows (18x18) so any pixel can find their neighbours. We first appreciated the black grid, because those extra columns and rows had non-initialized data. Once we noticed, we copied the correspondent data from the image into the shared memory.

4. There are several images of different sizes in the image folder. Try running the program on them and report how their execution time relates to file sizes.

We have run the program with the different images. However, the gaussian filter function failed for every of them except for the first one in the example. We don't know if this could be due to the size of the images. We have revised the code several times but haven't found the bug.

Anyway, we suppose that time will increase with the image size since more calculation will be needed to be performed. However, if there are enough "blocks" for all of the pictures, as the blocks will compute in parallel, there shouldn't be a strong difference since all the blocks will end more or less in the same time.

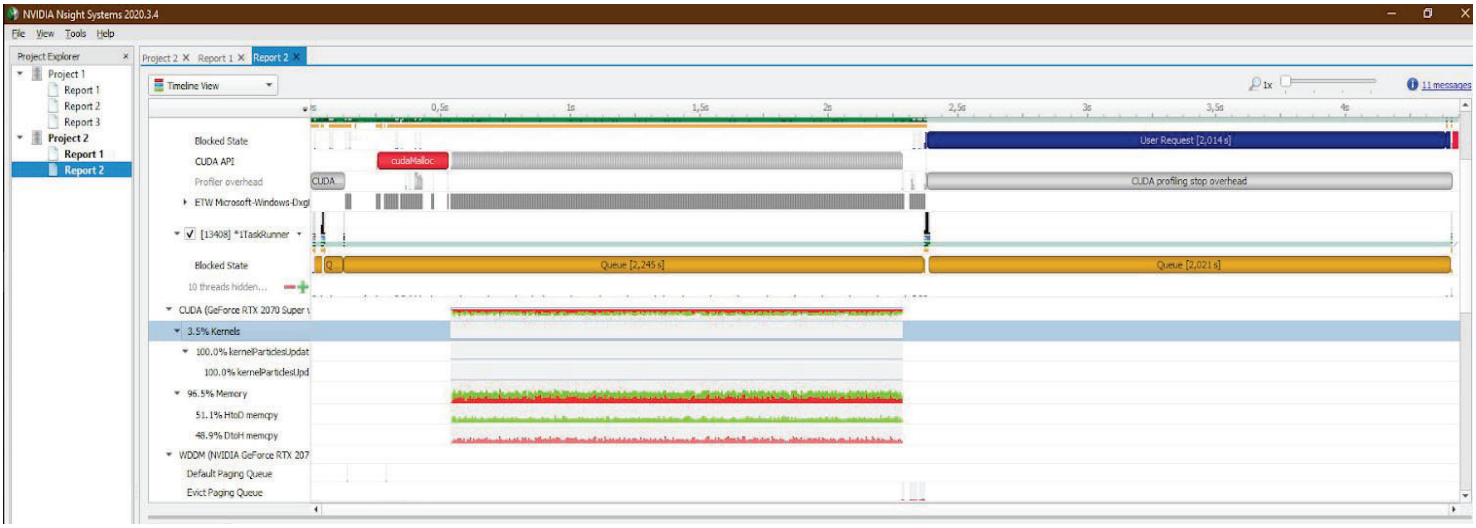
Exercise 2 - Pinned and Managed Memory

a) What are the differences between pageable memory and pinned memory, what are the tradeoffs?

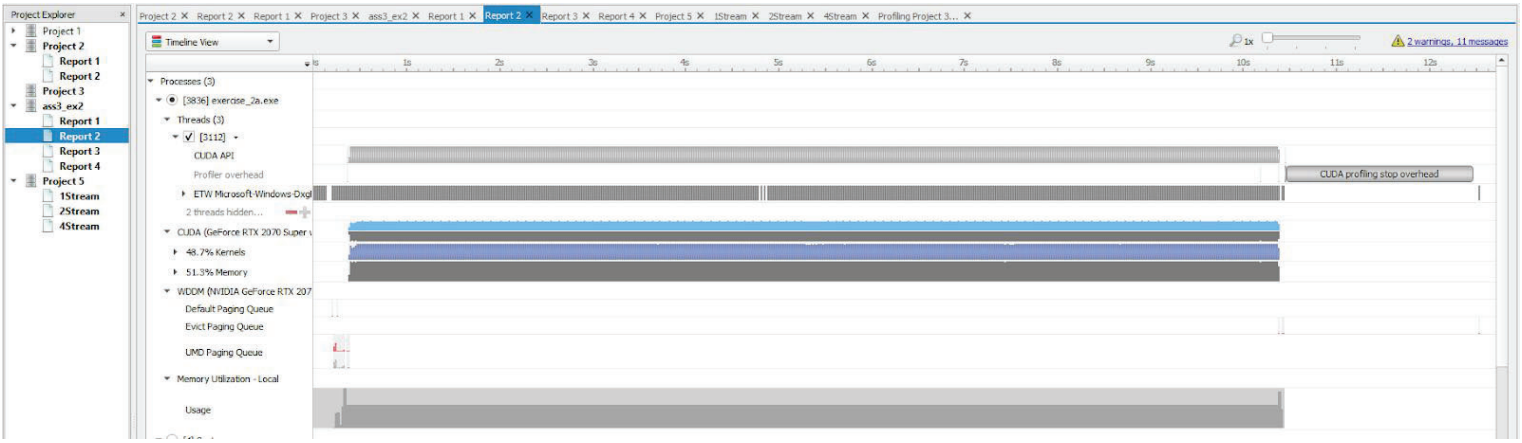
Pinned memory is useful because we avoid the information that is constantly being accessed to be paged off. However, it is not possible to pin all the memory. This is why pageable memory exists. Pageable memory allows us to access more information and load the pages when we need them and page them off when there are others to be accessed.
To sum up, it is handy to have some important memory pinned that is of constant use (it is necessary to use streams in CUDA), but for the rest of the data that is not that important, it is better to have it pageable.

b) Do you see any difference in terms of break down of execution time after changing to pinned memory from pageable memory?

As it can be seen in the next following pictures, when we used only pageable memory, 96.5% of the CUDA execution time was due to memory transferences back and forth and just a small 3.5% was due to the kernel processes. This changed when we also used pinned memory for particle data with a 51.3% of the time spent in copying memory against a 48.9% of kernel processes.



With only pageable memory



Using pinned memory too.

c) What is a managed memory? What are the implications of using managed memory?

It is a single memory space that can be accessed by both GPU and GPU. The main change is that there is no need of moving the memory from GPU and CPU and vice-versa. Either GPU or CPU can read and write from the same memory.

This is possible thanks to CUDA manager software, because it takes care of migrating the necessary pages from one to another.

In the code, we replace malloc() to use cudaMallocManaged() instead.There is no need of using cudaMemcpy().

Finally, when the kernel is launched, it is necessary to synchronize using cudaDeviceSynchronize() before using the output in the host because it works asynchronously.

d) If you are using Tegner or lab computers, the use of managed memory will result in an implicit memory copy before CUDA kernel launch. Why is that?

We are not using Tegner nor lab computers, but we think these supercomputers are using multiple GPU's. The Cuda software manager migrates pages from device to host and vice-versa but we guess that it is not possible to do it between GPU's. This is why, in order to "share" space, the memory has to be copied first to all the GPUs's memories.

Exercise 3 - CUDA Streams / Asynchronous Copy - Particle Batching

a) What are the advantages of using CUDA streams and asynchronous memory copies?

The first advantage is that it is possible to divide the data into segments, which often improves the performance because it allows us to use the shared memory of the block.

Streams also let us overlap computation in the device and CUDA memory copies. This is a technique very useful to use the time when the host or device is idle. For example, while the first stream is copying data, the second stream can launch the kernel. Streams can go in parallel.

b) What is the performance improvement (if any) in using more than one CUDA stream?

We have tested the GPU with 1 steam, 2 streams and with 4 streams. As we can see in the following pictures the GPU is faster with more streams.

	1 stream	2 streams	4 streams
Approach_1	860.61 ms	620.30 ms	315.12 ms
Approach_2	872.76 ms	665.18 ms	337.06 ms

The code of both approaches can be checked in the exercise_3.cu file.

Both approaches reach the same result. This happens because of the new Hyper-Q GPU feature, which eliminates the need to tailor the launch order.

c) What is the impact of batch size on the performance?

We have tested out a program with 8 streams, which implies a smaller batch size. The results show that the GPU spent more time than computing with only 4 streams. Our approach is that the optimum program is the one that finds a balance between the batch size and the number of threads. The GPU has a kernel engine and copying unit which can run in parallel using streams but they have a limited capability so they can only overlap a limited amount of kernel and copying operations. Otherwise, we could launch one single operation in each thread and get the result immediately.

Exercise 4: Bonus Exercise - CUDA Libraries - cuBLAS

a) Explain why the matrix size has to be a multiple of 16?

Because we are working with a TILE_SIZE = 16 so our matrices must be a multiple of TILE_SIZE. In the code, tiles represent the submatrices. Therefore, the matrices should be splittable in the smallest units that will be computed, submatrices (tiles), and for this reason, the size of the matrices must be a multiple of the tiles size (16).

b) Refer to shared_sgemm_kernel(). There are two __syncthreads() in the loop. What are they used for, in the context of this code?

The first __syncthreads() makes sure that every thread inside its block has its shared memory complete before starting calculating the multiplication.

The second __syncthreads() makes sure that the submatrix in the result is complete (C'). This is necessary because C' = A'B'. If C' is not complete, the threads will start overwriting the shared memory with the following submatrices (A'' and B'') which are used to calculate the rest of the result submatrices (C'',C'''...).

1. **What is the directive that can potentially improve performance in the actual multiplication? What does it do?**

   After conducting a lot of research, our first guess was that the directive #pragma unroll could improve the multiplication of the matrices.It serializes the instructions before the execution causing an improvement in the internal calculations.

   However, then we found the directive #pragma acc parallel loop. After having read NVIDIA documentation, it seems that the library to include “openacc.h” and the SDK that contains that file, will be released for Windows soon. Therefore, we couldn't test it but we are almost sure that it can improve the performance.This directive papalleizes multiple operations such as the ones in a for loop.

   Some of the documents that helped us reaching this conclusions are the next ones:

   https://developer.download.nvidia.com/CUDA/training/OpenACC_1_0_intro_jan2012.pdf

   https://www.hpc2n.umu.se/sites/default/files/conferences-courses/2017/Gpu-course/slides-2017-12-12.pdf

   https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

2. **There is a large speedup after switching from using global memory to shared memory, compared to the Edge Detector in Exercise 1. What might be the reason?**

   The reason could be that the complexity of matrix multiplication is $O(n^3)$ and the algorithm for the edge detector is not that demanding. Hence, the improvement is much more noticeable when more calculations are made.

c) Refer to cublas_sgemm(). We asked that you compute C = B A instead of C = A B . It has to do with an important property of cuBLAS. What is that, and why do we do C = B A ?

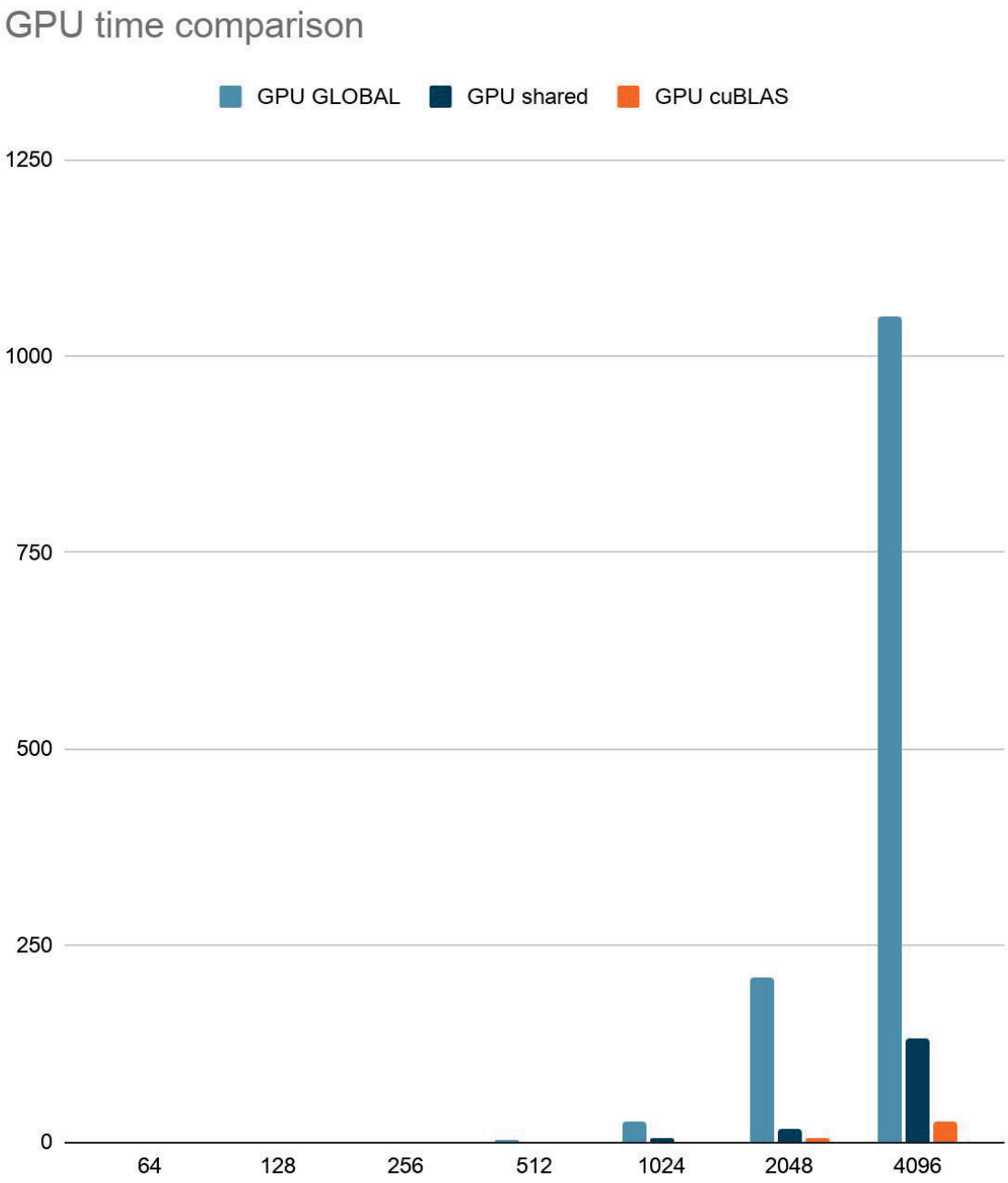
The property is that cuBLAS “understands” the matrices in column-major order which is equivalent to say that it reads the transposed matrix (because we are used to write matrices in row-major order. Moreover, cuBLAS also returns the final matrix in column-major order, so:

If the input is C = AB, it will read the matrices as $C=A^t B^t = (BA)^t$, which is not the result that we want.

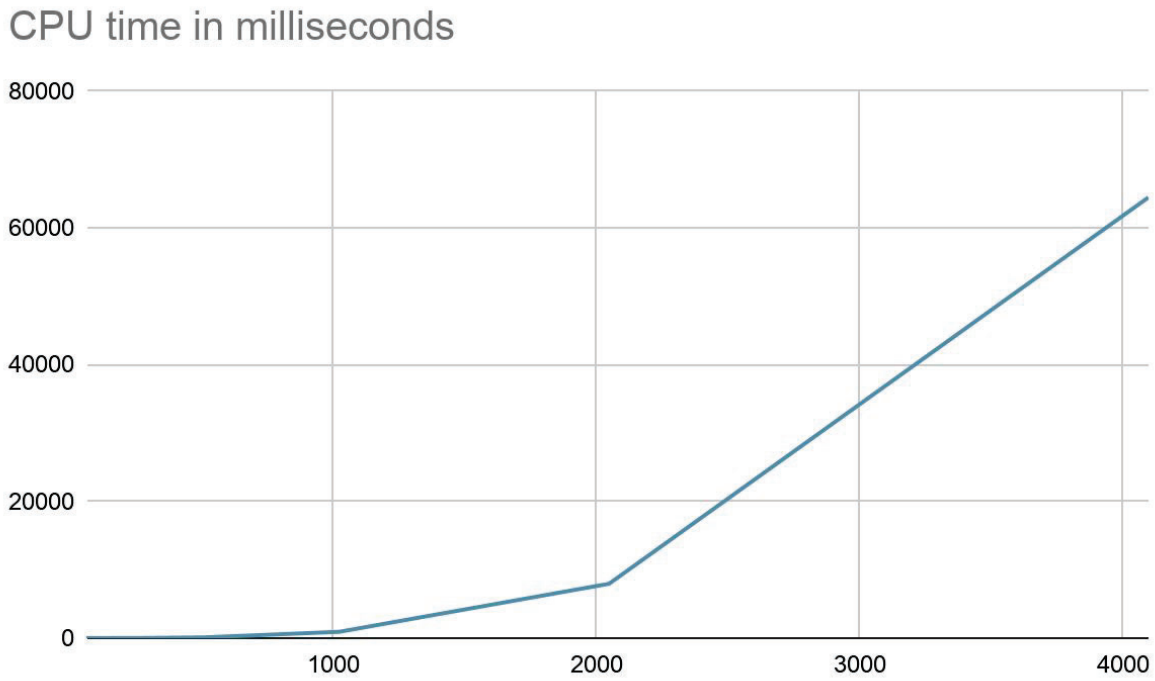
To avoid extra calculations it is easier to change the input to C = BA,
because it will be “read” as $C=B^t A^t = (AB)^t$

Because the result is given in column-major order: $(AB)^t$ is AB in row-major order, which is the expected result.

d) Run the program with different input sizes, for example from 64, 128, ... , to 4096. Make a grouped bar plot of the execution times of the different versions (CPU, GPU Global, GPU Shared, GPU cuBLAS). You can plot CPU results in a separate figure if the execution time goes out of the scale compared to the rest.



MATRIX_SIZE	GPU global	GPU shared	GPU cuBLAS
64*64	0ms	0ms	0ms
128	0ms	0ms	0ms
256	0ms	0ms	0ms
512	2.991ms	0.995ms	0ms
1024	25.965ms	3.995ms	1.014ms
2048	208.491ms	15.963ms	4.001ms
4096	1049.19ms5	131.647ms	25.929ms



MATRIX_SIZE	CPU
64	0 ms
128	0 ms
256	0 ms
512	2.991 ms
1024	25.965 ms
2048	208.491ms
4096	1049.195 ms

e) The way the execution time benchmark that is implemented in the code is good enough for this exercise, but in general it is not a good way to do a benchmark. Why?

We guess that it is not good to do a benchmark because each GPU can have a different internal architecture. This could mean that one algorithm might be optimal in one GPU because it maximizes its resources. However, a different GPU might be optimized with a different implementation.