

APPLIED GPU PROGRAMMING

ASSIGNMENT 2

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: Hello World!

a) Explain how the program is compiled and the environment that you used.

We are using the 11.0 version of CUDA and the Visual Studio 2019 framework to compile the program. This means that we are not using a command prompt to write the lines to run our program, because we can do it by using the Visual Studio interface. We use the NVIDIA Nsight Visual Studio Edition included in the NVIDIA toolkit to be able to debug our code easily.

b) Explain what are CUDA threads and thread blocks.

A CUDA thread is the smallest execution unit in a CUDA program. The CUDA function calls are designed to run a function on different threads simultaneously and working in parallel.

A CUDA block is an abstract concept that represents a group of threads. Each block inside a GPU has its own memory which is shared between all the threads of the block. They also run their own stream processor. CUDA blocks were created to get a better performance while processing in the GPU.

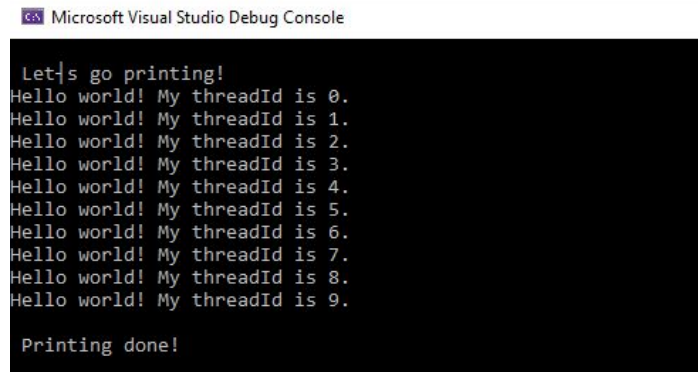
c) How are they related to GPU execution?

Changing the configuration of a kernel function, the programmer can specify the number of blocks and threads per block. The GPU will distribute the work depending on that configuration. For example, it is not the same to run a function with 1 block and 32 threads per block as running it with 32 blocks and 1 thread per block.

We have also learned that each CUDA block is divided into warps. Each one has a number of threads selected by the SM. All of the threads in the same warp execute the same instruction. We can see how it works by running our program. All the threads


allocated in the same warp are shown in order, but if the threads are in different warps, it is impossible to know the output order.

Execution with 1 block and 10 threads per block:

A screenshot of the Microsoft Visual Studio Debug Console window. The title bar reads "Microsoft Visual Studio Debug Console". The console output is as follows:

```
Let's go printing!  
Hello world! My threadId is 0.  
Hello world! My threadId is 1.  
Hello world! My threadId is 2.  
Hello world! My threadId is 3.  
Hello world! My threadId is 4.  
Hello world! My threadId is 5.  
Hello world! My threadId is 6.  
Hello world! My threadId is 7.  
Hello world! My threadId is 8.  
Hello world! My threadId is 9.  
  
Printing done!
```

Execution with 10 blocks and 1 thread per block: (forcing them to be in separated warps):

A screenshot of the Microsoft Visual Studio Debug Console window. The title bar reads "Microsoft Visual Studio Debug Console". The console output is as follows:

```
Let's go printing!  
Hello world! My threadId is 1.  
Hello world! My threadId is 7.  
Hello world! My threadId is 0.  
Hello world! My threadId is 3.  
Hello world! My threadId is 6.  
Hello world! My threadId is 9.  
Hello world! My threadId is 2.  
Hello world! My threadId is 5.  
Hello world! My threadId is 8.  
Hello world! My threadId is 4.  
  
Printing done!
```

Exercise 2: Performing SAXPY on the GPU

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

We want to make sure the GPU uses the correct amount of blocks. We use the formula: $(\text{ARRAY_SIZE} + \text{TPB} - 1) / \text{TPB}$

This grants blocks for every necessary thread. However, it is also true that some extra threads may be running but we can exit those processes using a condition with the thread id:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
if(i < ARRAY_SIZE)
...
```

- b) **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.**

This table shows the results of our CPU and GPU running the program with 64 threads per block.

The time is measured in milliseconds (note that some times are 0,00000 because we don't measure with enough precision, but it works for the experiment)

ARRAY_SIZE	CPU	GPU
100.000	0.000 ms	0.000 ms
1.000.000	0.997 ms	0.000 ms
10.000.000	8.975 ms	0.985 ms
100.000.000	81.818 ms	7.978 ms

As we can appreciate in the table, the CPU is always far slower than the GPU. For example, at the third entry in the table, it can be seen that the CPU takes 9 times more than the GPU to process the same data. As the array size increments exponentially, GPU and CPU timing do the same.

EXERCISE 3: CUDA simulation and GPU Profiling

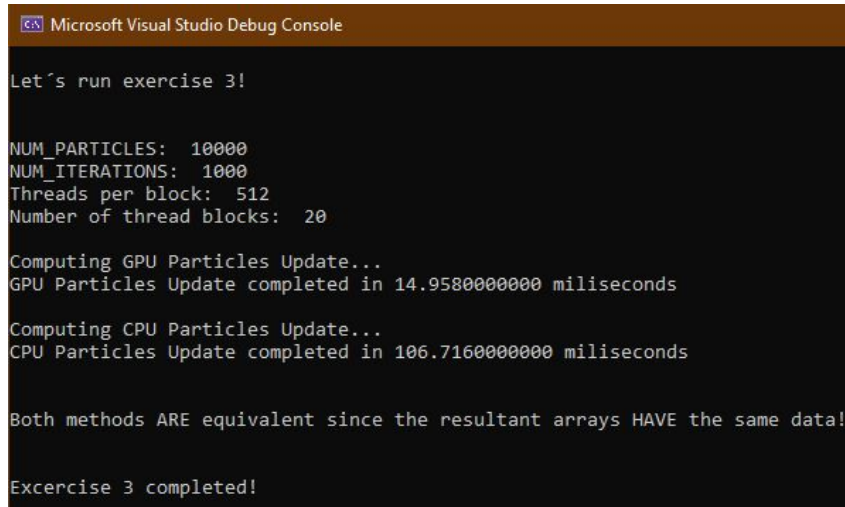
- a) **Measure the execution time of the CPU and GPU version, varying the number of particles.**

This table shows the time for Threads per Block = 512

NUM_PARTICLES	CPU	GPU
10.000	106.716 ms	14.958 ms
100.000	1259.324 ms	43.883 ms
1.000.000	11994.341 ms	259.317 ms
10.000.000	116362.859 ms !!!	2056.501 ms

As it is shown in the table, CPU's time increases exponentially. The increase of particles does not bother the GPU as it only spends 2 seconds processing 10.000.000 of particles. It is clear that parallelism is a technique that can be used to improve the performance of many problems such as graphics or neural network systems.

- b) Generate one or more performance and include it in the report with a description of the experimental setup, 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?



```
Microsoft Visual Studio Debug Console

Let's run exercise 3!

NUM_PARTICLES: 10000
NUM_ITERATIONS: 1000
Threads per block: 512
Number of thread blocks: 20

Computing GPU Particles Update...
GPU Particles Update completed in 14.958000000 milliseconds

Computing CPU Particles Update...
CPU Particles Update completed in 106.716000000 milliseconds

Both methods ARE equivalent since the resultant arrays HAVE the same data!

Excercise 3 completed!
```

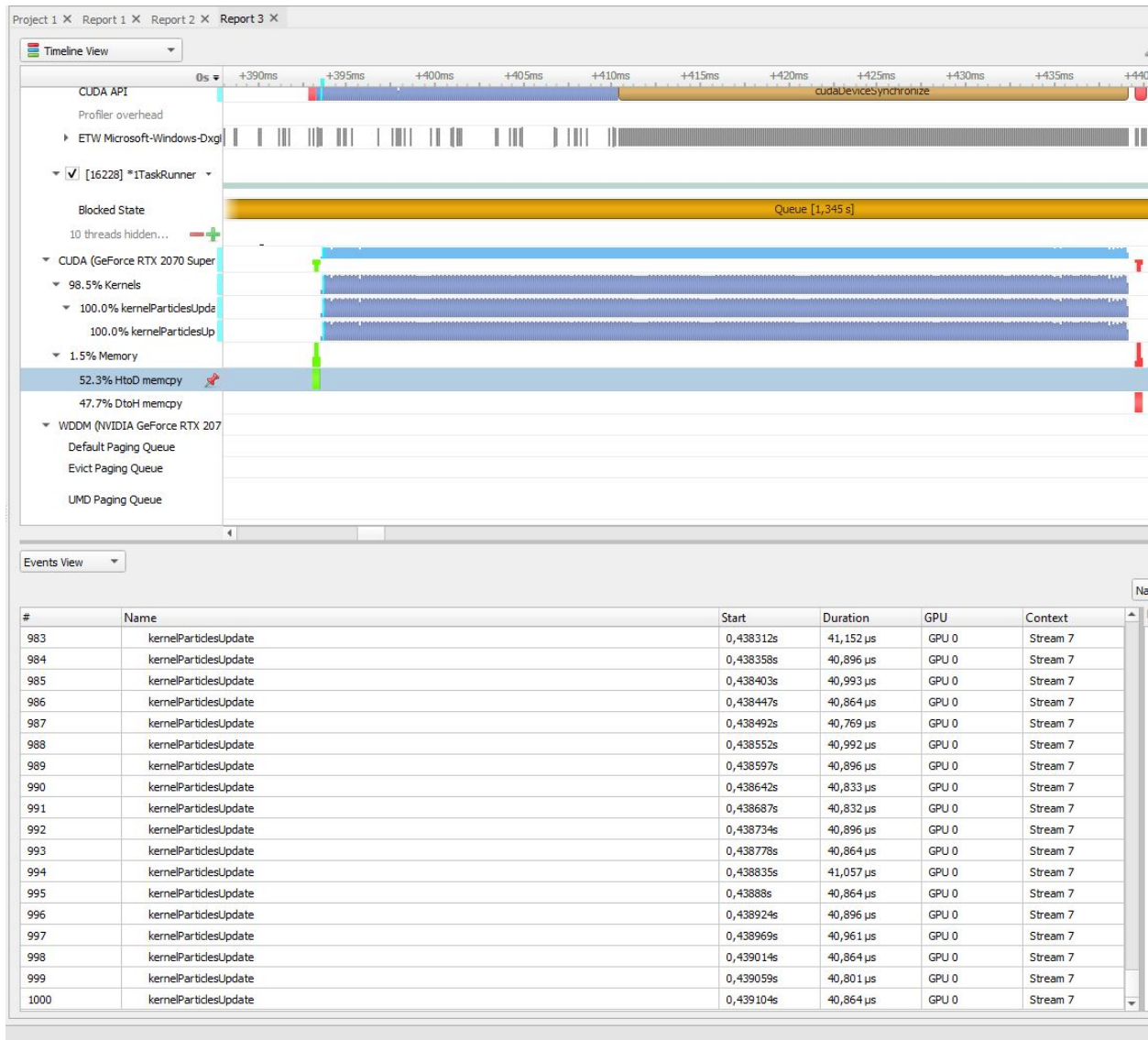
This figure shows the data used to calculate the particles and the time used for each CPU and GPU. It is compiled in Visual Studio using a Intel i9 10th generation as a CPU and a NVIDIA Geforce GTX 2070 as a GPU. We summarized these program runs in the tables.

This table shows the time for NUM_PARTICLES = 10.000.000

Threads per block	GPU
32	2669.863 ms
64	1948.769 ms
128	1950.829 ms
256	2012.617 ms
512	2056.501 ms
1024	2200.118 ms

The optimal configuration is 64 threads per block although 128 is a good configuration too. With only 32, the GPU might have had problems allocating the threads into the different blocks, slowing the process considerably.

c) If the simulation involves CPU dependent functions still holds? How would you expect the performance will change in terms of GPU execution?



Using the command `nvprof` and the NVIDIA Nsight Systems, we have been able to get the graphics of the whole process.

As you can see in the picture, thread process (called `kernelPartidesUpdate`) have taken around 40 ms (40 μ s / table entry) all together.

The sum of Host to Device memory transfer and Device to Host memory transfer is about 600 μ s \rightarrow 0.6ms.

If that transfer would have to be made as many times as the number of iterations, (NUM_ITERATIONS was 1000 in this execution), the data would have to be moved in both ways in every iteration (0,6ms), and that would take $0.6\text{ms} * 1000 \rightarrow 0.6$ seconds!!

Thread process time would remain the same since the data to be processed and the way to process has not changed. This means that transferring data (0.6 seconds) would take much longer than real processing (40 ms) in comparison.

To reduce the times that memory has to be transferred, it could be better to use bigger blocks. As blocks have shared memory, the data transference of each block could be made altogether, and there wouldn't be such an amount of transferences.

To sum up, if the program requires a lot of copying from GPU to CPU and vice versa, those operations would become the slowest ones. We guess that moving data is done block by block, which would mean that the more threads per block the GPU has, the less operations the system would have to do. Hence, bigger blocks would be the best configuration for that problem.

EXERCISE Bonus (+1) - Calculate PI with CUDA

a) Measure the execution time of the GPU version, varying NUM_ITER.

This table shows the time for Threads per block = 32

NUM_ITER	GPU	Pi approximation
1024 (2^{10})	17.952 ms	3.183594
4096 (2^{12})	19.978 ms	3.137695
16384 (2^{14})	21.930 ms	3.129150
65536 (2^{16})	23.918 ms	3.138611
262144 (2^{18})	25.931 ms	3.141754

b) Measure the execution time, varying the block size in the GPU version from 16, 32, ..., up to 256 threads per block.

This table shows the time for NUM_ITER = 262144 (2^{18})

Threads per block	GPU
32	25.931 ms
64	25.930 ms
128	25.896 ms
256	24.223 ms
512	11.363 ms
1024	10.483 ms

When we changed the number of threads per block, we realized that the program stops calculating if the points are inside the circle. We couldn't figure out why this is happening since the thread code is the same, and we don't know how to debug it. We just know that the threads counts are always 0, so they are not counting the successful events.

- c) Change the code to single precision and compare the result and performance with the code using double precision. Do you obtain what you expected in terms of accuracy and performance? Motivate your answer.**

After having changed the randomised numbers' precision from double to float, we have obtained quite similar GPU execution times and the same precision for the pi approximation. We guess that it is a negligible difference for the general time and performance.

We are not really sure if we did not understand the question or if we were supposed to get another result,

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 block has $16 \times 16 = 256$ 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 (14×14) than before (16×16), 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

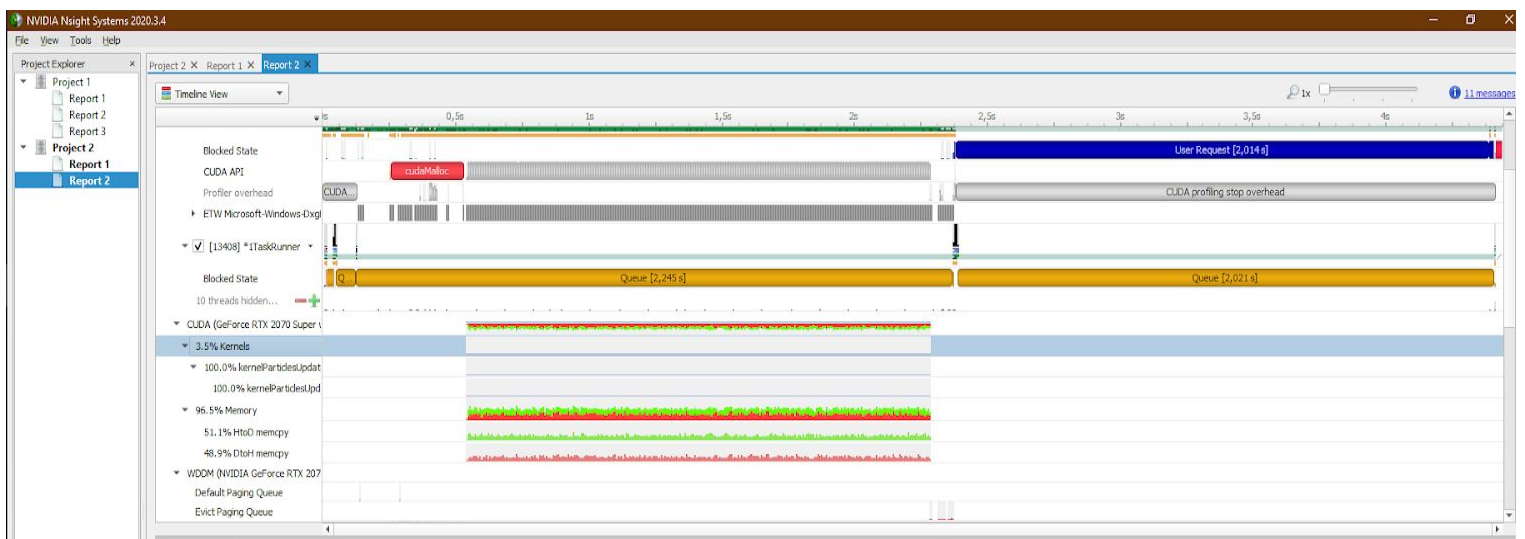
- a) **What are the differences between pageable memory and pinned memory, what are the tradeoffs?**

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.

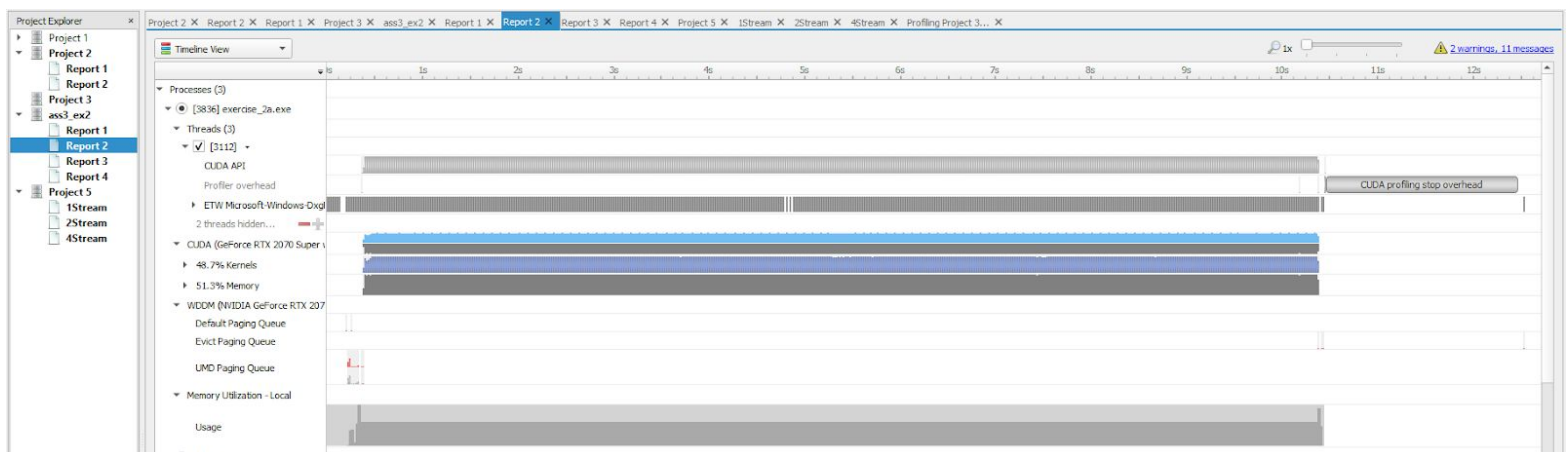
To sum up, it is handy to have some important memory pinned that is of constant use (it is necessary to use streams in CUDA), but for the rest of the data that is not that important, it is better to have it pageable.

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 CPU. 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 GPU'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 stream, 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 ($C'', C''' \dots$).

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 parallelizes 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:

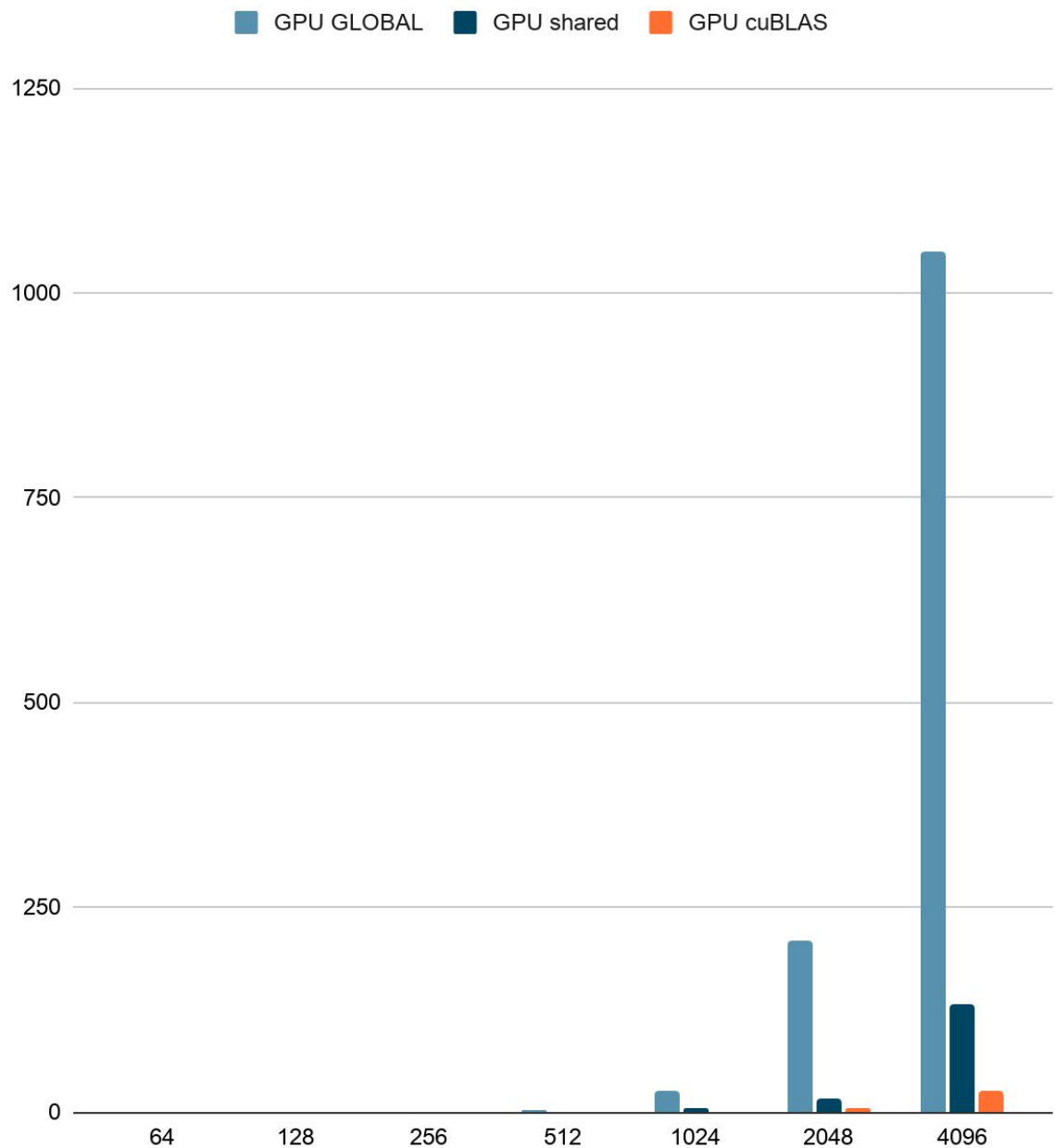
If the input is $C = AB$, it will read the matrices as $C = A^t B^t = (BA)^t$, which is not the result that we want.

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.

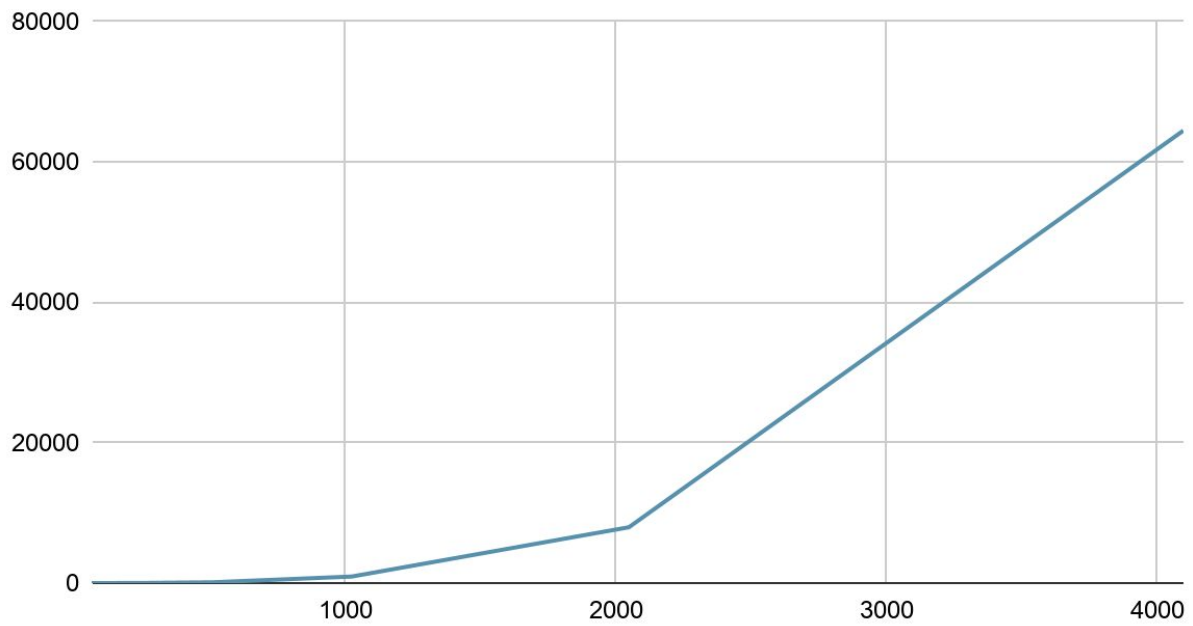
d) Run the program with different input sizes, for example from 64, 128, ... , to 4096. Make a grouped bar plot of the execution times of the different versions (CPU, GPU Global, GPU Shared, GPU cuBLAS). You can plot CPU results in a separate figure if the execution time goes out of the scale compared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

GPU time comparison



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

CPU time in milliseconds



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.