Universitat Autònoma de Barcelona

Parallel Programming: GPU

Assignment 4

Alfredo Hernández Alejandro Jiménez

28th January

Contents

1 Introduction

The backbone of many computer systems is usually centred in its CPU, a general purpose processor. But then, we have the GPU, a more specialized processor contanied in many computers alongside the CPU. These GPU processors were created to handle very complicated calculations mainly related to 2D and 3D graphics, for which the CPU is not so efficient. Thus, CPU and GPU are two quite different processing units, designed for two different purposes, with different trade-offs, and so they have different performance characteristics. Certain tasks are faster in a CPU while other tasks are faster computed in a GPU.

In order to understand their different procedures simplistically, one could think as that the CPU excels at doing complex manipulations to a small set of data, whereas the GPU excels at doing simple manipulations to a large set of data.

Lately, some GPUs have got so specialised that they are now capable of rendering certain calculations even better than central processors [1]. Because of this, there is now a movement that is taking advantage of a computer's GPU to supplement a CPU and speed up various tasks [2], or even combining both into a single unit.

As technology continues to advance, we might see an increasing degree of convergence of these once-separate parts. Actually, AMD envisages a future where the CPU and GPU are one single unit, capable of seamlessly working together on the same task [3].

In this assignment we aim to improve the execution time of the now-common Laplace 2D code and measure the impact of using GPU-oriented parallelisation, which is usually done by CUDA. For this project, CUDA shall be employed for visualisation purposes regarding performance, but not for the parallelisation itself. For this, we are going to make use of *OpenAcc*; a programming standard for parallel programming of heterogeneous CPU/GPU systems [4]. The advantage of using OpenACC is that - like OpenMP - we can annotate C code that should be accelerated and/or parallelised. Moreover, it now intends to be used both with NVIDIA's and AMD's GPUs, since its 2.0 version presented important steps towards more general implementations [5].

In order to make the biggest difference on the code and fully appreciate the power of parallel computing, we chose the most computational demanding loop and created many threads using OpenACC, which would split the iterations into different, parallel, more efficient processes.

2 Implementation of OpenACC

2.1 Getting to know OpenACC

The first thing to do is to identify the GPUs in our computer. We do this doing the following:

```
module load pgi64/17.4
pgaccelinfo | grep "Device␣Name" && pgaccelinfo | grep "PGI␣Compiler"
```

If we use the host `aolin21`, we see that the server has two GPUs:

```
Device Name:                    GeForce GTX 1080 Ti
Device Name:                    GeForce GTX 680
PGI Compiler Option:            -ta=tesla:cc60
PGI Compiler Option:            -ta=tesla:cc30
```

However, on the first Laboratory session we will be running our tests on the local Laboratory machines to be able to use the Visual Profiler and familiarise ourselves with OpenACC. The GPU on the Lab's machines is:

```
Device Name:                    GeForce GT 430
PGI Compiler Option:            -ta=tesla:cc20
```

If we need the full information about the GPUs we would just need to run `pgaccelinfo` without grepping the output.

In case we have more than one GPU on the machine, we need to select which one to use exporting the following environment variable:

```
# Use first GPU
export CUDA_VISIBLE_DEVICE=1,0
# Use second GPU
export CUDA_VISIBLE_DEVICE=0,1
```

To compile our code using the PGI compiler, whether it is for CPU or GPU, we need run the following CLI command:

```
# Compile for CPU
pgcc -lm -fast -Minfo=all code-cpu.c -o run-cpu
# Compile for GPU
pgcc -lm -fast -acc -ta=tesla:cc20 -Minfo=accel code-gpu.c -o run-gpu
```

Note that PGI Compiler to performs a series of optimisations on the code generated using **`#pragma`** directives; these optimisations can be listed using the `-Minfo` flag.

For profiling we have several options:

- `perf stat`, which should not be used for code compiled for GPU.
- `pgprof`, the profiler provided by the PGI Compiler.

- `nvprof` (or `nvvp` for the GUI version), the NVIDIA Profiler.

If we want to profile our code compiled for GPU, we should use either the PGI or NVIDIA profilers.

To use the NVIDIA Profiler (either `nvvp` for the visual profiler, or `nvprof` for the CLI version) we need to load `cuda` to make NVIDIA drivers available:

```
1 module add cuda/7.5
```

As a general rule, based on the findings of [6], we have decided to use `#pragma acc kernels` as much as possible, as they found that explicitly putting all the directives shown by `-Minfo` flag whilst compiling sometimes has a bad impact on the execution time. We will only use manual directives when we want to force the compiler to do something it is unable to do automatically.

Having all this in mind, we can proceed to work with versions of the code same `laplace2d.c` code. The performance results of these will be analysed with more detail in § 3.

3 Analysis of 2D Laplace program

3.1 CPU versions of the program

To ease the work with the different versions of the code, we will use a systematic naming structure for the source codes, the binaries, and the logs:

- `lapCPU[#number]-description.c`
- `lapCPU[#number]`
- `lapCPU[#number]-perf.log`

This systematic naming convention will allow us to analyse the results easily using bash, AWK (using `extract_perf.awk`), and R.

3.1.1 Baseline `laplace2d.c` (`lapCPU1`)

First of all, we use the baseline `laplace2d.c` program as provided by the professors:

```
1 pgcc -lm -fast -Minfo=all lapCPU1-baseline.c -o lapCPU1
2 perf stat ./lapCPU1 250 2> lapCPU1-perf.log
```

The PGI compiler performs a series of optimisations on the code generated for CPU. The most relevant performance metrics are the elapsed execution time (135.33 s), the total number of executed machine instructions (40.26×10^9), and the IPC rate (0.09).

3.1.2 Baseline `lapFusion` (`lapCPU2` & `lapCPU3`)

The second analysed version of the code is using the baseline `lapFusion.c` program as provided by the professors:

```
1 pgcc -lm -fast -openmp -Minfo=all lapCPU2-Fusion-baseline.c -o
    lapCPU2
2 perf stat ./lapCPU2 4096 250 2> lapCPU2-perf.log
```

The execution time is significantly reduced from 135.33 s to 17.06 s, but the total number of executed instructions is multiplied by 2.75 (109.06×10^9). This is something that we must check. A detailed look at the output from the PGI compiler reveals that the optimised code is not vectorized, even with the OpenMP **`#pragma`** that confirms that the loop in line 23 is vectorisable.

In order to improve the task of the compiler, we include the `inline` keyword before the declaration of functions `stencil()` and `max_error()` in the `lapFusion.c` code, and it is compiled and executed again:

```
1 pgcc -lm -fast -openmp -Minfo=all lapCPU3-Fusion-good.c -o lapCPU3
2 perf stat ./lapCPU3 4096 250 2> lapCPU3-perf.log
```

Again, the execution time is reduced from 17.06 s to 9.43 s, and the total number of executed instructions is almost halved (56.66×10^9), but still the optimised code is not vectorised. The compiler is not able to make a good analysis of the data dependencies when the matrices are represented as one-dimensional vectors and the elements are addressed using complex expressions.

Instead of trying to improve the output from the PGI compiler we opt by using the GCC compiler.

3.1.3 Parallelised lapFusion using OpenMP (lapCPU4)

First of all, we need to load the GCC compiler into the system:

```
1 module load gcc/7.2.0
```

Now, we create a parallelised version of the `lapFusion` code using the following **`#pragma`** directives:

```
24   #pragma omp parallel for num_threads(3)
25   for ( j=1; j < n-1; j++ )
26   #pragma omp simd reduction(max:error)
27   for ( i=1; i < n-1; i++ )
```

Then we compile using the GCC compiler and execute as usual:

```
1 gcc -Ofast -lm -fopenmp lapCPU4-Fusion-parallel.c -o lapCPU4
2 perf stat ./lapCPU4 4096 250 2> lapCPU4-perf.log
```

Notice that using the `inline` keyword has a very small influence on the execution time when vectorising the code using OpenMP.

The execution with three threads for 250 convergence loop iterations is about 2.2 times faster than using a single thread and the operations have indeed been divided by 3 (19.24×10^9).

The fact that the performance does not scale linearly with the number of threads indicates that there is a performance problem. Since the total number of instructions executed is almost the same, the problem is an unexpected reduction of the IPC rate: some instructions are executed slower when using three processing cores than when using only one core. The explanation is that the DRAM memory bandwidth is almost exhausted with a single thread, and therefore the performance is bounded by the peak bandwidth of the DRAM memory.

3.1.4 Comparison of the CPU versions

In figure 1 below we can see a visual comparison of the execution times of the CPU versions of the Laplace program. The execution time is computed for 10 000 iterations of the convergence loop, which is extrapolated by multiplying by 40 the time for executing 250 iterations of the convergence loop.

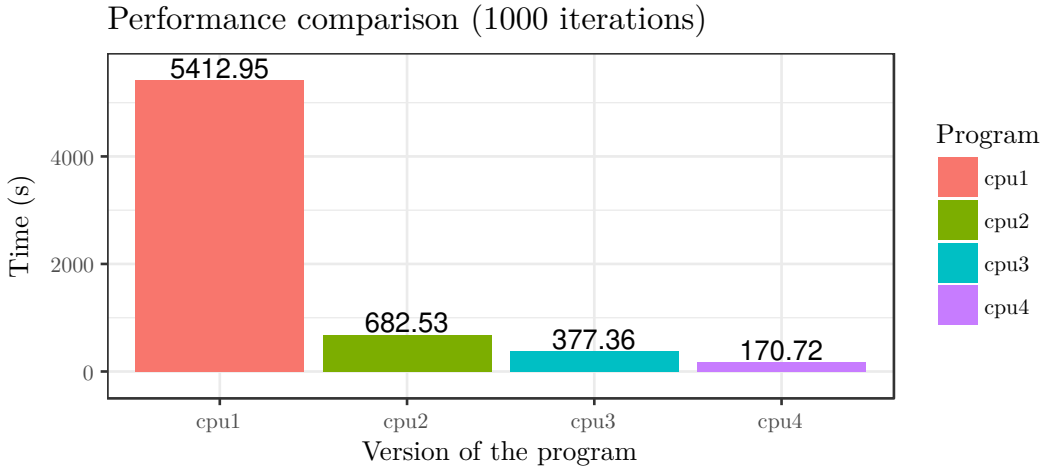

Figure 1: Execution times of the CPU versions of the program

In the figure we can clearly see how each of the improvements has resulted in a reduction of the compilation time, specially the change from the sequential `laplace2d.c` version to the `lapFusion.c` version.

3.2 GPU versions of the program

To ease the work with the different versions of the code, we will use a systematic naming structure for the source codes, the binaries, and the compiler and profiler logs:

- `lapGPU[#number]-description.c`
- `lapGPU[#number]`
- `lapGPU[#number]-comp.log`
- `lapGPU[#number]-nvp.log`
- `lapGPU[#number]-metrics.log`

This systematic naming convention will allow us to analyse the results easily using bash, AWK (using `extract_nvp.awk`), and R.

3.2.1 Baseline (`lapGPU1`)

For this part, we will make our modifications using **`#pragma`** directives on the `laplace2d.c` code.

```
77 while ( error > tol && iter < iter_max )
78 {
79   #pragma acc kernels
80   for( i=1; i < m-1; i++ )
81     for( j=1; j < n-1; j++)
82       Anew[j][i] = ( A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i]) / 4;
```

```
83
84    error = 0.f;
85    #pragma acc kernels
86    for( i=1; i < m-1; i++ )
87      for( j=1; j < n-1; j++)
88        error = fmaxf( error, sqrtf( fabsf( Anew[j][i]-A[j][i] ) ) );
89
90    #pragma acc kernels
91    for( i=1; i < m-1; i++ )
92      for( j=1; j < n-1; j++)
93        A[j][i] = Anew[j][i];
94
95    iter++;
96    if(iter % (iter_max/10) == 0) printf("%5d, %0.6f\n", iter, error);
97  }
```

Since we will use the GPU for the calculations, we must use the PGI compiler:

```
pgcc -fast -acc -ta=tesla:cc30 -Minfo=accel lapGPU1-baseline.c -o
    lapGPU1 &> lapGPU1-comp.log
```

Then we can use `nvprof` to extract performance stats and some useful metrics:

```
nvprof ./lapGPU1 10000 2> lapGPU1-nvp.log
nvprof --kernels "main_82_gpu" --metrics
    sm_activity,inst_executed,l2_utilization,
    dram_utilization,dram_read_throughput,dram_write_throughput,ipc
    ./lapGPU1 100 2> lapGPU1-metrics.log
```

The first thing to notice is that if we only use **`#pragma acc kernels`** directives before each of the loops, most of the computation time is wasted on copying memory from the CPU to the GPU (and the other way around) for each iteration, as seen in figure 2 below.

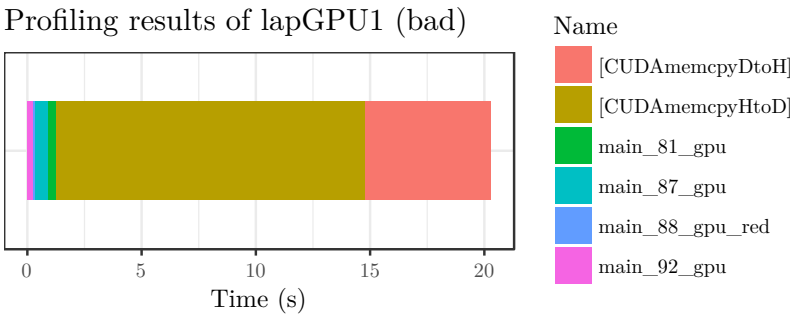

Figure 2: Timeline of the execution time of the GPU1 version (bad), for 250 iterations

To fix this, we also need to include the following **`#pragma`** directive before the main **`while`** loop to tell the program that the GPU already has all the variables; this way the memory is only copied once from the CPU to the GPU at the beginning, and from the GPU to the CPU when the program finishes:

```
77 #pragma acc data copyin(A,Anew)
78 while ( error > tol && iter < iter_max )
```

In this way, we reduce the copying time to a bare minimum, as seen in figure 3 below. Even though we do 40 times the amount of iterations, the time is only doubled. This is a clear indication that we are using the GPU almost purely for calculations.

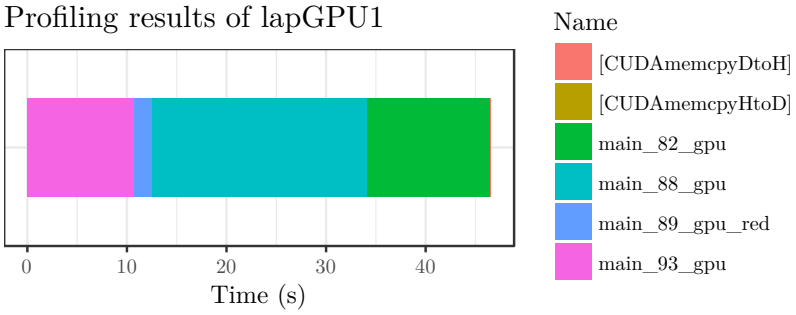

Figure 3: Timeline of the execution time of the GPU1 version, for 10 000 iterations

3.2.2 Optimisation: loop fusion (lapGPU2)

For this version of the program, we fuse the loop for the calculation of the next step and the error into one single loop:

```
81 #pragma acc kernels
82     for( i=1; i < m-1; i++ )
83       for( j=1; j < n-1; j++){
84         Anew[j][i] = ( A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i]) / 4;
85         error = fmaxf( error, sqrtf( fabsf( Anew[j][i]-A[j][i] ) )
              );}
```

Then we compile with the PGI compiler as usual, and extract performance and metrics information:

```
1 pgcc -fast -acc -ta=tesla:cc30 -Minfo=accel lapGPU2-loopfusion.c -o
    lapGPU2 &> lapGPU2-comp.log
2 nvprof ./lapGPU2 10000 2> lapGPU2-nvp.log
3 nvprof --kernels "main_82_gpu" --metrics
    sm_activity,inst_executed,l2_utilization,
    dram_utilization,dram_read_throughput,dram_write_throughput,ipc
    ./lapGPU2 100 2> lapGPU2-metrics.log
```

In figure 4 below we can clearly see how the two loops have been fused into one single operation, and how this fused operation is faster than the two operations by themselves in figure 3. Notice that the reduction operation and the one done in `main_93_gpu` have essentially the same duration in both implementations.

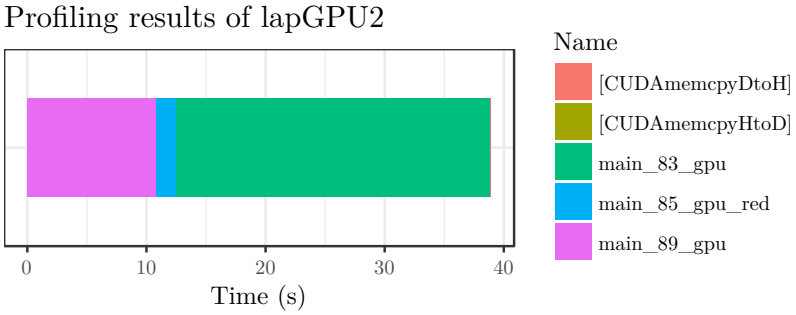

Figure 4: Timeline of the execution time of the GPU2 version, for 10 000 iterations

3.2.3 Optimisation: loop interchange (lapGPU3)

For this version, we just exchange the order of the nested loops in lines 82–83 and 88–89:

```
81 for( i=1; i < m-1; i++ )
82   for( j=1; j < n-1; j++){
```

↦

```
81 for( j=1; j < n-1; j++ )
82   for( i=1; i < m-1; i++){
```

Then we compile with the PGI compiler as usual, and extract performance and metrics information:

```
1 pgcc -fast -acc -ta=tesla:cc30 -Minfo=accel lapGPU3-loopinter.c -o
    lapGPU3 &> lapGPU3-comp.log
2 nvprof ./lapGPU3 10000 2> lapGPU3-nvp.log
3 nvprof --kernels "main_83_gpu" --metrics
    sm_activity,inst_executed,l2_utilization,
    dram_utilization,dram_read_throughput,dram_write_throughput,ipc
    ./lapGPU3 100 2> lapGPU3-metrics.log
```

Having a look at figure 5 below and comparing it to figure 4, we see no major improvements; one could even say that the performance is a bit worse.

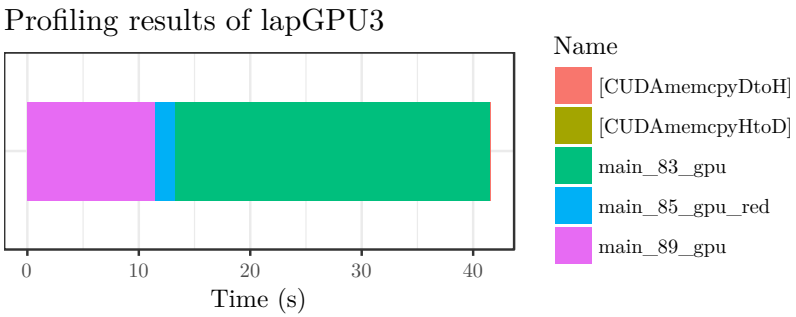

Figure 5: Timeline of the execution time of the GPU3 version, for 10 000 iterations

The reason for not being any major improvement is that the PGI compiler already knows the “good” order of the loops when it compiles the program.

3.2.4 Optimisation: `sqrt` out of the loop (`lapGPU4`)

For this version, we move the $\sqrt{\texttt{error}}$ outside of the loop, since we do not need to know the square root of the intermediate errors, just the final error:

```
82 for( i=1; i < m-1; i++ )
83   for( j=1; j < n-1; j++){
84     Anew[j][i] = ( A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i]) / 4;
85     error = fmaxf( error, fabsf( Anew[j][i]-A[j][i] ) ) ;}
86 error = sqrtf(error);
```

Then we compile with the PGI compiler as usual, and extract performance and metrics information:

```
1 pgcc -fast -acc -ta=tesla:cc30 -Minfo=accel lapGPU4-sqrt.c -o
    lapGPU4 &> lapGPU4-comp.log
2 nvprof ./lapGPU4 10000 2> lapGPU4-nvp.log
3 nvprof --kernels "main_83_gpu" --metrics
    sm_activity,inst_executed,l2_utilization,
    dram_utilization,dram_read_throughput,dram_write_throughput,ipc
    ./lapGPU4 100 2> lapGPU4-metrics.log
```

This improvement just has a small impact on the execution time, as seen in figure 6.

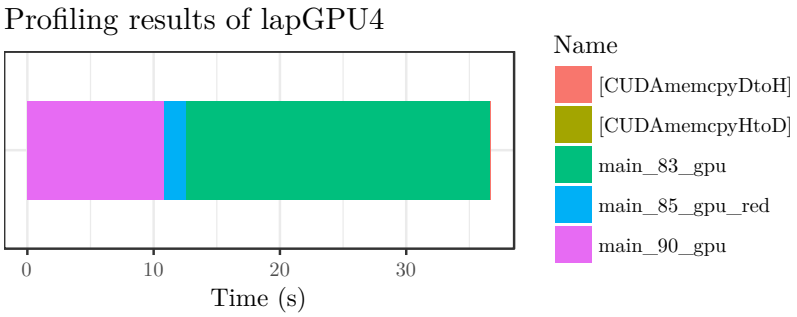

Figure 6: Timeline of the execution time of the GPU4 version, for 10 000 iterations

As a summary of these last versions, we can say that versions 2 and 3 are essentially the same, because the compiler already manages to optimise the order of the loops. and Version 4 does not add much to the performance, because of the limitation of the DRAM bandwidth.

3.2.5 Optimisation: double buffer (`lapGPU5`)

For this version, is to use a double buffer. This allows us to avoid copying from `Anew` to `A` by reversing roles of `A` and `Anew` on every iteration:

```
81 if(iter % 2 == 0){
82   #pragma acc kernels
```

```
83    for( j=1; j < n-1; j++){
84      for( i=1; i < m-1; i++ ){
85        Anew[j][i] = ( A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])*0.25f;
86        error = fmaxf( error, fabsf( Anew[j][i]-A[j][i] ) );}}
87  } else{
88    #pragma acc kernels
89    for( j=1; j < n-1; j++){
90      for( i=1; i < m-1; i++ ){
91        A[j][i] = (
             Anew[j][i+1]+Anew[j][i-1]+Anew[j-1][i]+Anew[j+1][i])*0.25f;
92        error = fmaxf( error, fabsf( A[j][i]-Anew[j][i] ) );}}
93  }
```

Then we compile with the PGI compiler as usual, and extract performance and metrics information:

```
1 pgcc -fast -acc -ta=tesla:cc30 -Minfo=accel lapGPU5-double.c -o
    lapGPU5 &> lapGPU5-comp.log
2 nvprof ./lapGPU5 10000 2> lapGPU5-nvp.log
3 nvprof --kernels "main_84_gpu" --metrics
    sm_activity,inst_executed,l2_utilization,
    dram_utilization,dram_read_throughput,dram_write_throughput,ipc
    ./lapGPU5 100 2> lapGPU5-metrics.log
```

This is the next major improvement next to the loop fusion. In figure 7 we can clearly see how the execution is divided into two processes of computation and reduction. This is a clear indication that the double buffer works as intended.

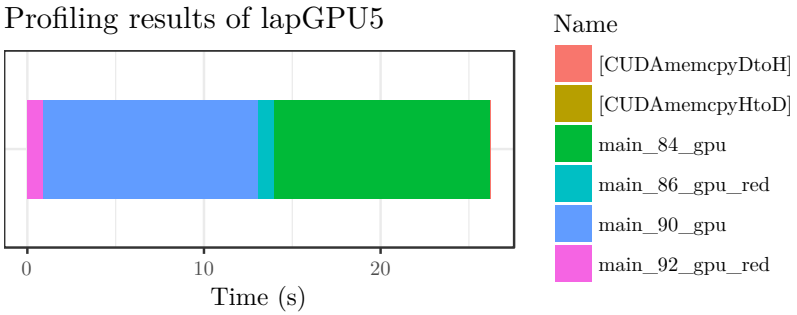

Figure 7: Timeline of the execution time of the GPU5 version, for 10 000 iterations

3.2.6 Optimisation: parallel loop (vector 128) (lapGPU6)

Looking at `lapGPU5-comp.log` we see we can still make a small optimisation to reduce the execution time: adding `vector_length(128)` to the `#pragma acc kernels` directives in lines 82 and 88 to force the size of the parallel loop:

```
82 #pragma acc kernels vector_length(128)
```

With this we make sure to divide the GPU into threads of the appropriate size to fully take advantage of our hardware.

Then we compile with the PGI compiler as usual, and extract performance and metrics information:

```
pgcc -fast -acc -ta=tesla:cc30 -Minfo=accel lapGPU6-vec128.c -o
    lapGPU6 &> lapGPU6-comp.log
nvprof ./lapGPU6 10000 &> lapGPU6-nvp.log
nvprof --kernels "main_84_gpu" --metrics
    sm_activity,inst_executed,l2_utilization,
    dram_utilization,dram_read_throughput,dram_write_throughput,ipc
    ./lapGPU6 100 2> lapGPU6-metrics.log
```

In figure 8 we can see how the functioning to figure 7 is essentially the same. We just get a small time improvement.

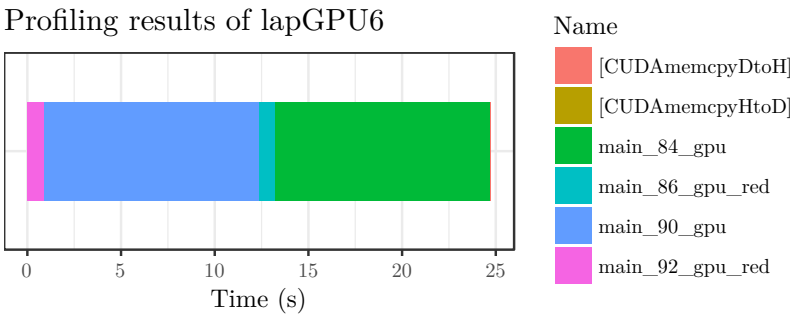

Figure 8: Timeline of the execution time of the GPU6 version, for 10 000 iterations

3.2.7 Comparison of the GPU versions

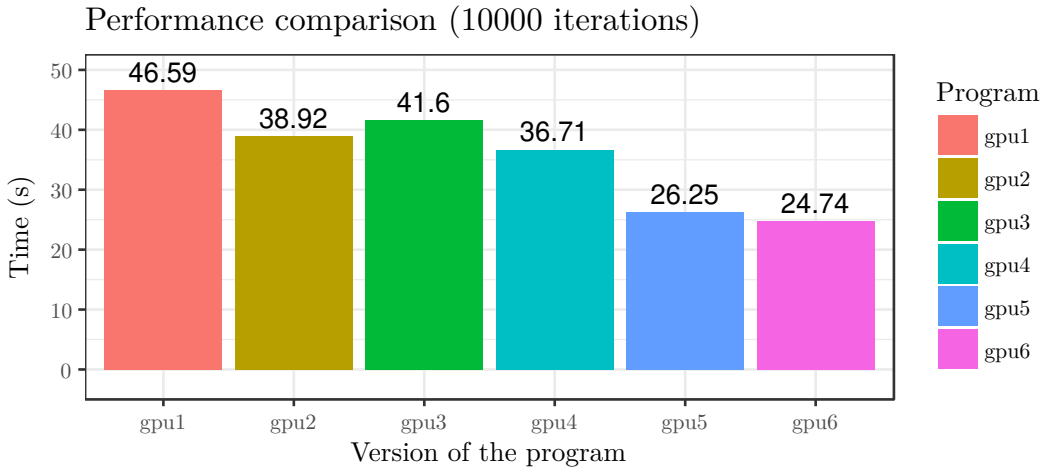

Figure 9: Execution times of the GPU versions of the program

If we take a look at figure 9, we can compare the different implementations of GPU-oriented parallelisation and address its different results. It is relevant to note that not all implementations have been equally useful; as a matter of fact, some of them have been even counter-productive. Therefore, we must consider the characteristics of the best implementations individually and highlight its importance.

The great improvements in the performance when using these different implementations are observed in `lapGPU2` and `lapGPU5`. The former fuses different loops into a single loops, considerably reducing the operations needed to perform the same results; whilst the latter uses a double buffer in order to avoid carrying big matrices more times that needed, releasing a great deal of memory and hence easing the overall computation.

3.2.8 Summary of the metrics

In table 1 below we can see a few conclusions can be extracted:

- The biggest change is when implementing the loop fusion (`GPU2`). The executed instructions are almost doubled, whilst the DRAM read/write throughput is halved. The IPC rate is a bit worse, but that does not have an important impact on the performance.
- For some reason, the loop inversion (`GPU3`) has a pretty important, and bad, result on the performance as well as the metrics. The executed instructions were increased when compared to `GPU2`, being this probably the reason of the increase in the execution time.
- Moving the square root out of the loop (`GPU5`) improves a bit the DRAM read/write throughput and reduces the amount of executed instructions, as expected.
- Using the double buffer (`GPU5`), even though makes a great performance improvement, does not make a big change on the metrics.
- Forcing the size of the parallel loop (`GPU6`) to 128 not only reduces the execution time, but the total executed instructions whilst improving the IPC rate.

Version of the Program	`GPU1`	`GPU2`	`GPU3`	`GPU4`	`GPU5`	`GPU6`
Instructions Executed	24 105 472	46 363 722	50 047 882	43 117 312	43 117 312	41 005 504
L2 Cache Utilization	Mid (6)	Mid (4)	Mid (4)	Mid (4)	Mid (4)	Mid (4)
DRAM Utilization	Mid (6)	Low (3)	Low (3)	Low (3)	Low (3)	Mid (4)
DRAM Read (GB/s)	50.161	23.281	23.149	24.653	24.658	26.781
DRAM Write (GB/s)	51.078	24.181	23.979	26.454	26.455	27.929
Executed IPC	2.253 070	2.001 180	1.983 949	1.972 142	1.972 746	2.037 747
Multiprocessor Activity	99.72%	99.86%	99.87%	99.86%	99.86%	99.85%

Table 1: Summary of the average metrics obtained with `nvprof` for the different implementations of the program for GPU

3.3 Comparison of CPU vs GPU versions

Until now we have discussed the different implementations used within a CPU-oriented parallelisation and their compared performances; nextly, we have stutdied the different implementations employed within a GPU-oriented parallelisation. Now it would be interesting to contrast the performances acquired between all different CPU and GPU oriented parallelisation implementations.

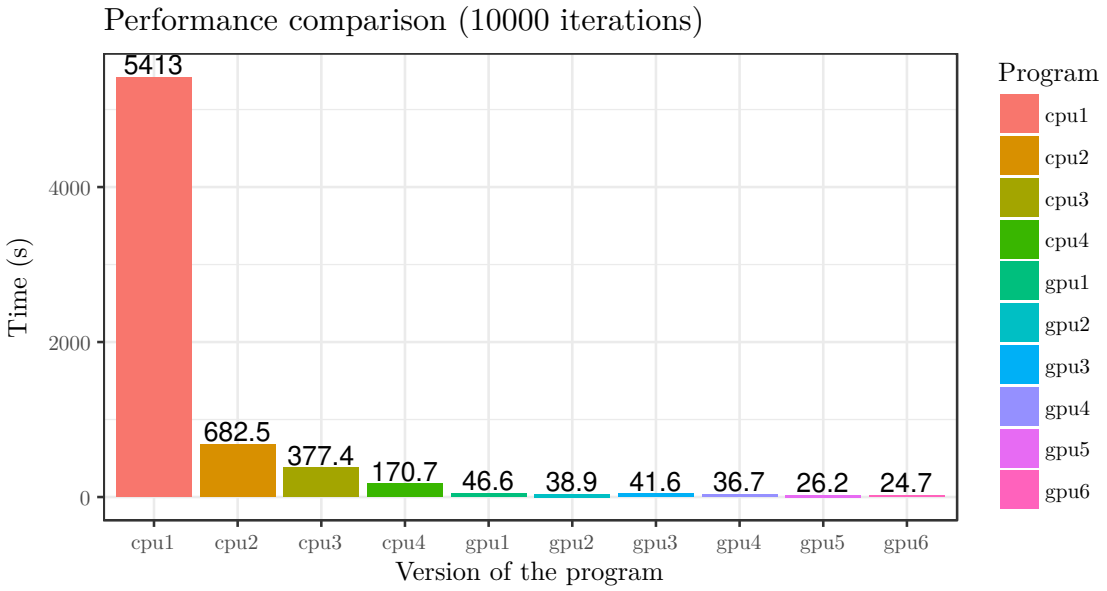

Figure 10: Execution times of the CPU and GPU versions of the program

On figure 10 above it is shown the execution times of all the different versions of the program. If we look closely at the figure, it is quite clear how dwarfed are the differences between the particular versions, when taking into account whether the version is CPU-oriented or GPU-oriented. The general conclusion extracted from this graph is that no matter how well implemented is a CPU parallelisation, the worst implementation using GPU is far better and it consumes considerably less computation time.

Despite that, some great improvements can be – and should be – acknowledged when well-implemented CPU parallelisation is used. This highlights the importance of a thorough parallelisation of the code, specially when no GPU is available.

4 Conclusions

During the resolution of this exercise, we have had the opportunity to manually create a parallel code, in contrast with the one we had optimised in the OpenMP assignment, that was itself an improvement from the very first code of the course. This time we have used a different methodology to paralellise the code, we have learned to take advantage of the GPU of our system, whose usage has increased dramatically over the past few years.

All of this has let us comprehend the importance of parallel programming when aiming for high performance computation, for we have experienced an incremental increase of the performance in the code we have been improving.

One of the most important things learn in this assignment has been the fact that one should not use OpenACC as a magic formula for improving the execution time of the code; one has to be aware of how the machine is working and optimise the code to really take advantage of the hardware capabilities of the system.

References

[1] Shuai Che and Michael Boyer and Jiayuan Meng and David Tarjan and Jeremy W. Sheaffer and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Elsevier*, 2008.

[2] J. Fung and S. Mann. Using Multiple Graphics Cards as a General Purpose Parallel Computer : Applications to Computer Vision. *University of Toronto*, 2004.

[3] What is Heterogeneous System Architecture (HSA)? URL: https://developer.amd.com/heterogeneous-computing-2/what-is-heterogeneous-system-architecture-hsa/.

[4] Nvidia, Cray, PGI, and CAPS launch 'OpenACC' programming standard for parallel computing. URL: https://www.theinquirer.net/inquirer/news/2124878/nvidia-cray-pgi-caps-launch-openacc-programming-standard-parallel-computing.

[5] PGI 2014 Adds OpenACC 2.0 Features for NVIDIA and AMD GPU Accelerators. URL: https://www.hpcwire.com/off-the-wire/pgi-2014-adds-openacc-2-0-features-nvidia-amd-gpu-accelerators/.

[6] OpenACC Programming and Best Practices Guide. (June), 2015. URL: http://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf.