EPITA
Graduate School of Computer Science

GPGPU Fast implementation

Iterative Closest Point algorithm in CUDA

Yossra Antari - Axen Georget - Romain Schloesing - Clément Réau

yossra.antari@epita.fr - axen.georget@epita.fr - romain.schloesing@epita.fr - clement.reau@epita.fr

supervised by

Edwin Carlinet - Joseph Chalazon

November 2020

Table of Contents

1 Introduction

This report presents the work of the implementation and performance analysis of an Iterative Closest Point (ICP) in CUDA. ICP is an algorithm employed to minimize the difference between two clouds of points. ICP is often used to reconstruct 2D or 3D surfaces. In other words, one point cloud, the *reference* is kept fixed, while the other one, the *source*, is transformed to best match the reference. Below, we will describe you our train of thought for the implementation of the ICP in Cuda.

2 Division of tasks

Our workflow can be summarized to two words : pair programming.
Everyone in the team was involved in the CPU implementation and the GPU implementation at some point (debugging, implementations, optimisations). Axen took care of the benchmark and plot all our graphs for the analysis.

3 CPU Implementation

3.1 Find Correspondences

First, for each point in the actual cloud, we compute the corresponding closest point in the reference cloud. To implement this, we had two options, either use a 'brute force' algorithm or use a KNN (k nearest neighbor) algorithm. Even though KNN has better performances, it is much more suited for finding large number of neighbors. In our case, we just need to find the closest point, so we chose to implement the brute force method. The idea behind that is that for every points of our actual cloud, we loop through the entire reference cloud, compute the euclidean distance at each point and keep the closest one. At the end, we store the result in a matrix that will be used in the further steps.

3.2 Find and apply alignment

In this step we can summarize the algorithm of finding the rotation, scaling factor and translation offset given the previous pair of points as follows:

1. Find the centroids and compute the points relative to their centroids
2. For each pair of points, compute the nine possible products of the two vectors.
3. Compute the ten independent elements of the 4x4 symmetric matrix N
4. Find the eigenvalues and eigenvector of N.
   For this calculation, we did not use Eigen library. Indeed we implemented the *Power Iteration* algorithm that works well for our use case
5. Compute scaling factor
6. Compute translation offset

3.3 Compute residual error

For the last step, we compute the residual error and compare it to a predefined threshold. The residual error is defined as the mean square error between the transformed scene points and the model points. So it basically tell us how far we are from the desired shape.

4 GPU Implementation

4.1 Profiling

Once our CPU implementation was correctly done without the use of any library, we started to search the slowing parts of our algorithm. In order to do this, we profiled the CPU implementation. The goal here was to determine what to optimize first in the GPU implementation. Surprisingly, the slower parts were by far the memory allocation and de-allocation. The other slow parts were the nearest neighbours algorithm but it was nothing when compared to the memory management.

The issue at that point was that we could have try to optimize some basic functions like the matrix dot product or even the nearest neighbor but we were afraid that doing the memory copies in addition to the memory management will not help us in any way to gain time.

Our idea was thus to change the way we are handling memory by using full GPU memory. At that time we thought that this would be our best option but we later realized that it was not a very good idea.

4.2 Structure change

To achieve our plan (switching from host to device memory for the whole algorithm) we first had to change our structure. By using 'CudaMallocPitch' we knew that we would lost the size information contained into std::vector that we were using. The first step was thus to implement our own structure to represent a matrix containing the needed information like the size but also the pitch.

After this big refactor we changed all the memory allocation which were made on the host to the device using CUDA functions. At first we let the algorithm as is and we were calling them using 1 block of 1 thread.

4.3 First Implementation (GPU_1)

The only difference with the CPU implementation in our first GPU implementation is the location of the matrix in the memory. In our GPU implementation everything is stored in the device instead of in the host. The goal of this was not to loose time making copy between the host and the device. In addition to that the nearest neighbour algorithm was partially paralellized so the algorithm was not taking too much time.

After implementing this we realized that doing a loop and doing some conditions in the GPU was way slower than doing them in the CPU, this first implementation was thus way slower than the CPU implementation.

4.4 First Optimization (GPU_2)

Our first baseline was, as described, very slow and were not using the GPU strength at all except for the memory and the nearest neighbour algorithm. This first optimization was to use different threads and blocks to parallelize the functions that allow it. To do so we took every functions that were parallelizable easily which basically concerns all the matrix operations.

4.5 Second Optimization (GPU_3)

Now that we had an algorithm that uses the power of parallelization GPUs offers we used the tool 'nvprof' to find what we needed to optimize. It told us that in fact, half of the computation time was taken by the nearest neighbour algorithm. This can easily be explained considering that it consists in a brute-force loop to compute all the distances between the source and destination. One of those two loops were indeed parallelized but this was clearly not enough.

As the profiling were stating, this function was only called once per iteration, we thus decided to move this function back to the CPU by doing memory copy at the beginning and at the end of it, to go from device to host and execute the computation on the host. This has a hudge performances effect on our benchmark but still was slower than the CPU version.

4.6 Final Optimization (GPU_FINAL)

By doing a last profiling we saw that the function 'cudaDeviceSynchronize' was taking a lot of time and was called a lot. This final optimization thus consists in calling this function only when needed. This did not impact our performances a lot but we saw a slight gain in some cases.

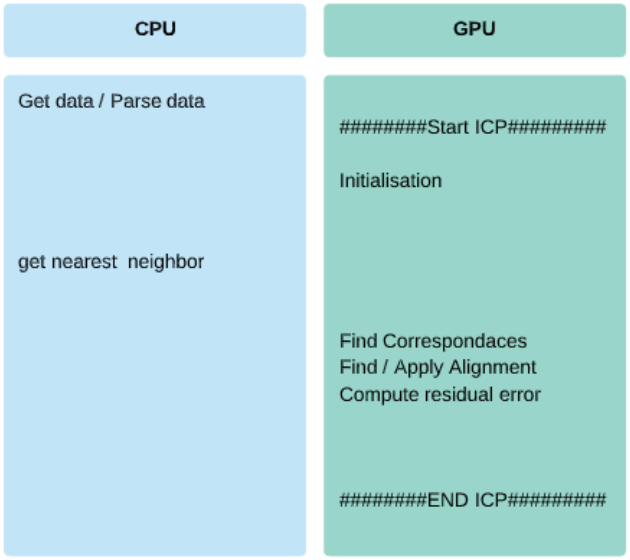


Figure 1: Program functions distribution between CPU and GPU

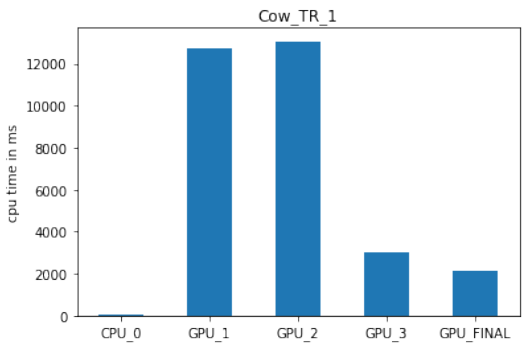
5 Performance's indicators

In order to analyse the performance performances of our different implementations, we used multiple performance's indicator such as :

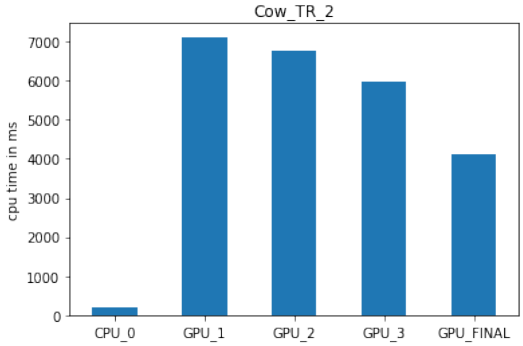
- Execution time taken by ICP to reach the threshold of 1e-5 with a maximum of 20 iterations
- Profiling in CPU and GPU in order to enlighten the pain points in our program

6 Analysis of performance

The graphs below show how much time is taken to execute an ICP in different set of points.

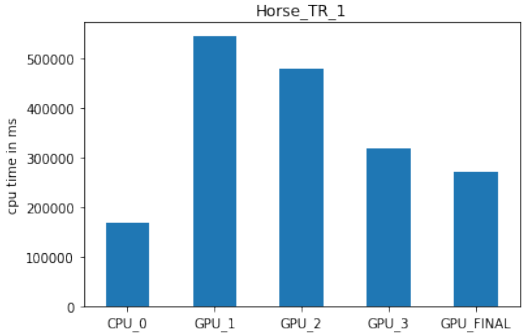


(a) Execution time on Cow_TR_1 as source with Cow_Ref as destination

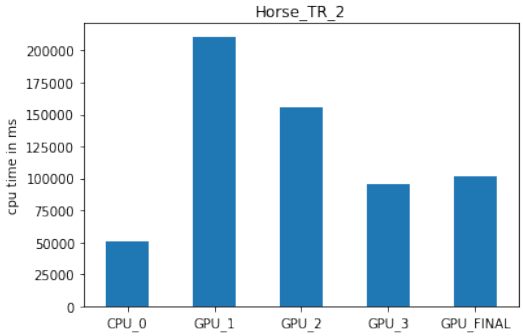


(b) Execution time on Cow_TR_2 as source with Cow_Ref as destination

One can see the improvement between our different GPU implementation but our CPU implementation is still faster.

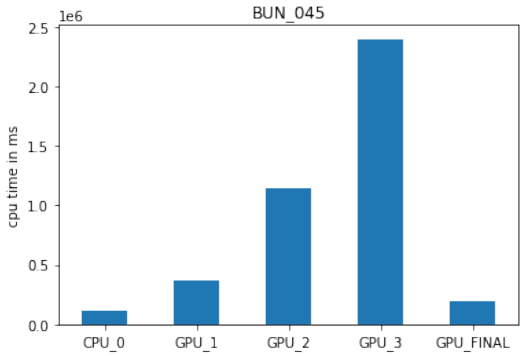


(c) Execution time on Horse_TR_1 as source with Horse_Ref as destination

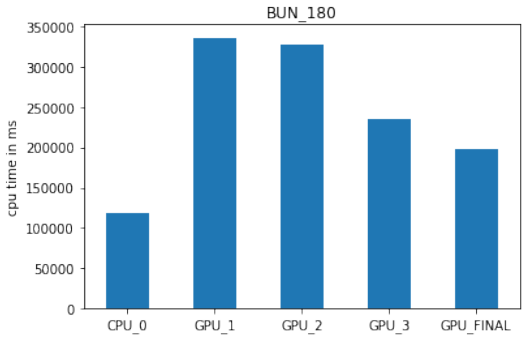


(d) Execution time on Horse_TR_2 as source with Horse_Ref as destination

Here, we can have the same observation on our implementation, but the test are on the Horse points sets.

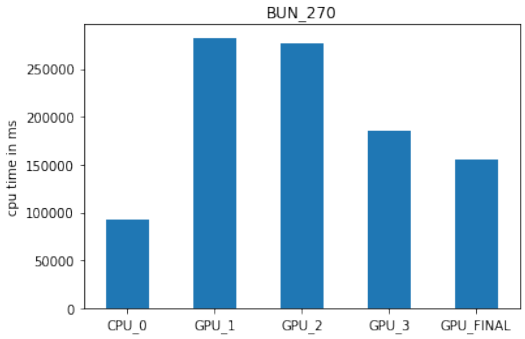


(e) Execution time on Bun_045 as source with Bun_000 as destination

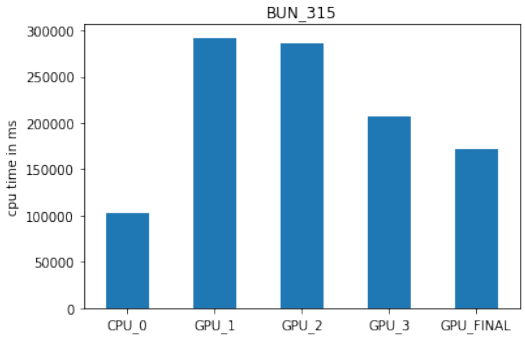


(f) Execution time on Bun_180 with Bun_000 as destination

For the bun180 points set, it is the same as before, but for the bun045 points set we have strange result on our GPU implementation with the final GPU version not far away behind the CPU. It is probably due to a particularity in the points set.



(g) Execution time on BUN_270 with Bun_000 as destination



(h) Execution time on BUN_315 with Bun_000 as destination

For those 2 last points sets, bun270 and bun315 we have the same result as the ones for cow and horse points sets.

7 Possible improvements

7.1 What we should have done differently?

7.1.1 Cuda implementation choices

When we did the Cuda version of our ICP algorithm, we were not really sure on how to do it. So after a little team brainstorming, we naively chose to switch a lot of our functions from host to device, thinking that it will greatly improve our performances. After seeing and analysing our GPU performances, one can say that it was a mistake. As an improvement, we should try to switch back some device functions to host and only let the one that performed great on the device.

7.2 Algorithm improvement

7.2.1 Nearest Neighbor implementation

One of the possible way to improve our project is to have a better get nearest neighbor function. The one we have is using a loop to go through every points of the desired shape. Another possibility would have been to use another algorithm, like KNN (k nearest neighbors). KNN doesn't perform way better than our version on 1 neighbor, but if we look for the 3 nearest neighbors and then compute the orthogonal projection of our actual points, maybe we can have a faster convergence.

8 Summary of benchmarks

algorithm	src file	src points	dst file	dst points	cpu time (ms)	frame rate (ms)
CPU_1	Cow_TR_1	2903	Cow_ref	2903	4.882521e+01	5.118811
GPU_1	Cow_TR_1	2903	Cow_ref	2903	1.273949e+04	0.069618
GPU_2	Cow_TR_1	2903	Cow_ref	2903	1.306145e+04	0.076512
GPU_3	Cow_TR_1	2903	Cow_ref	2903	3.031239e+03	0.329808
GPU_FINAL	Cow_TR_1	2903	Cow_ref	2903	2.114469e+03	0.472829
CPU_1	Cow_TR_2	2903	Cow_ref	2903	1.948224e+02	2.565970
GPU_1	Cow_TR_2	2903	Cow_ref	2903	7.114861e+03	0.140523
GPU_2	Cow_TR_2	2903	Cow_ref	2903	6.755896e+03	0.147990
GPU_3	Cow_TR_2	2903	Cow_ref	2903	5.988824e+03	0.166943
GPU_FINAL	Cow_TR_2	2903	Cow_ref	2903	4.111907e+03	0.243147
CPU_1	Horse_TR_1	48485	Horse_ref	48485	1.684183e+05	0.005937
GPU_1	Horse_TR_1	48485	Horse_ref	48485	5.459795e+05	0.001831
GPU_2	Horse_TR_1	48485	Horse_ref	48485	4.801102e+05	0.002083
GPU_3	Horse_TR_1	48485	Horse_ref	48485	3.184038e+05	0.003140
GPU_FINAL	Horse_TR_1	48485	Horse_ref	48485	2.702361e+05	0.003700
CPU_1	Horse_TR_2	48485	Horse_ref	48485	5.043096e+04	0.019824
GPU_1	Horse_TR_2	48485	Horse_ref	48485	2.107963e+05	0.004738
GPU_2	Horse_TR_2	48485	Horse_ref	48485	1.551297e+05	0.006445
GPU_3	Horse_TR_2	48485	Horse_ref	48485	9.537006e+04	0.010484
GPU_FINAL	Horse_TR_2	48485	Horse_ref	48485	1.012687e+05	0.009872
CPU_1	BUN_045	40097	BUN_000	48256	1.182015e+05	0.008459
GPU_1	BUN_045	40097	BUN_000	48256	3.657854e+05	0.002733
GPU_2	BUN_045	40097	BUN_000	48256	1.140975e+06	0.000876
GPU_3	BUN_045	40097	BUN_000	48256	2.396380e+06	0.000417
GPU_FINAL	BUN_045	40097	BUN_000	48256	1.953864e+05	0.005117
CPU_1	BUN_180	40251	BUN_000	48256	1.178384e+05	0.008485
GPU_1	BUN_180	40251	BUN_000	48256	3.363086e+05	0.002973
GPU_2	BUN_180	40251	BUN_000	48256	3.272717e+05	0.003055
GPU_3	BUN_180	40251	BUN_000	48256	2.360457e+05	0.004236
GPU_FINAL	BUN_180	40251	BUN_000	48256	1.971701e+05	0.005071
CPU_1	BUN_270	31701	BUN_000	48256	9.294070e+04	0.010758
GPU_1	BUN_270	31701	BUN_000	48256	2.824420e+05	0.003540
GPU_2	BUN_270	31701	BUN_000	48256	2.771960e+05	0.003607
GPU_3	BUN_270	31701	BUN_000	48256	1.856397e+05	0.005386
GPU_FINAL	BUN_270	31701	BUN_000	48256	1.549719e+05	0.006452
CPU_1	BUN_315	35336	BUN_000	48256	1.032101e+05	0.009687
GPU_1	BUN_315	35336	BUN_000	48256	2.919414e+05	0.003425
GPU_2	BUN_315	35336	BUN_000	48256	2.859656e+05	0.003496
GPU_3	BUN_315	35336	BUN_000	48256	2.068475e+05	0.004834
GPU_FINAL	BUN_315	35336	BUN_000	48256	1.724262e+05	0.005799

9 References

- Iterative Closest Point (ICP) - 5 Minutes with Cyrill by Cyrill Stachniss, YouTube: `https://www.youtube.com/watch?v=QWDM4cFdKrE` (Accessed: October 2020)
- Iterative Closest Point (ICP) Wikipedia page: `https://en.wikipedia.org/wiki/Iterative_closest_point` (Accessed: October 2020)
- A Tutorial on Rigid Registration, Iterative Closed Point (ICP) by Shireen ELHABIAN Amal Farag ALY FARAG, University of Louisville, CVIP Lab: `http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_ICP09.pdf` (Accessed: October 2020)
- Power Iteration Wikipedia page: `https://en.wikipedia.org/wiki/Power_iteration` (Accessed: October 2020)
- Maximizin Unified Memory Performance in CUDA, Nvidia Website: `https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/` (Accessed: October 2020)
- ICP implementation in Python, notebook by Igor Bogoslavskyi, Github: `https://github.com/niosus/notebooks/blob/master/icp.ipynb` (Accessed: October 2020)