GPU Optimization of the ICP Algorithm

How to increase the performances of an algorithm using a GPU with CUDA

Rayane Amrouche
SCIA 2021
EPITA
rayane.amrouche@epita.fr

Mamoune El Habbari
SCIA 2021
EPITA
mamoune.el-habbari@epita.fr

Amine Heffar
SCIA 2021
EPITA
amine.heffar@epita.fr

Yassir Ramdani
SCIA 2021
EPITA
yassir.ramdani@epita.fr

***Abstract*—In this work, we first began by implementing the Iterative Closest Point algorithm (later on referred to as ICP) on CPU. This implementation later served as a baseline where we identified the bottlenecks that needed to be optimized. This gave birth to another version that uses the GPU's capabilities to vastly improve the performances of the algorithm. A final version pushes the GPU to its limits to get the very best solving speeds.**

***Index Terms*—ICP, algorithm, GPU, optimization, performances**

I. Description of the Problem

Given two sets of points, a *model* and a *scene*, in each iteration the ICP algorithm tries to minimize the difference between the two of them so that the scene matches the model. Indeed, the model stays fixed while the algorithm *transforms* the scene using scaling, rotation and translation operations. It is particularly useful for aligning point clouds of different views of an object.

ICP is very resource hungry, especially when the input is comprised of a large number of points and when the operations that we talked about earlier are executed in sequential, the performances are heavily impacted. Fortunately, the operations used in ICP can easily be subject to parallelization which makes the algorithm a good candidate for a GPU based implementation.

II. Baseline CPU Implementation

To implement our CPU baseline, we used a paper [1] written by Shireen Elhabian detailing the ICP algorithm. In this section, we are going to go briefly over the different steps of the algorithm so that we can highlight which ones need to be ported on GPU. This version does not use any kind of parallelization.

Basically, the algorithm can be divided in four distinct steps:

1) we start off by finding the corresponding closest point in the model for each point of the scene;
2) we then compute the scaling (s), rotation (R(.)) and translation (t) factors;
3) we apply the transformation ($y = s * R(x) + t$) on the whole set of points;
4) finally we compute the error between the model and the resulting set of points.

These four steps are repeated until the stop condition is reached (which is an error value lower than a certain predetermined threshold) or if the algorithm reaches a maximum number of iterations.

A. Finding the Corresponding Closest Point

The first step of finding the corresponding closest point is very expensive in terms of processing power. Indeed, for *each* point of the scene, it computes the closest point to it in the model using the euclidean distance. This is by far the most time consuming task of our algorithm (it will be highlighted later in the benchmarks. The complexity of this loop is in $O(n^2)$ which is not viable under real conditions. The good news is these operations are independant from one another which is ideal for parallelization.

B. Computing the Transformation Factors

The algorithm starts by computing the rotation factor. To this end, the quaternion method is applied to extract a rotation matrix. Then by comparing the scene and the matrix of corresponding closest points the scaling factor is computed. Finally, the difference between the scaled and rotated scene centroid point and and the centroid of the corresponding closest points gives the translation vector.

In terms of optimization, we can look at the centroid computation and the computation of the scaling factor that requires a matrix operation between each point of the scene and the corresponding points, and finally the computation of the error which looks at the difference between two matrices.

C. Applying the Transformation

The matter of applying the transformation is simple: we transform each point of our scene using the the previously computed factors following the formula:

$$s * r * p + t$$

where s is the scaling factor (a scalar), r the 3x3 rotation matrix, p a point of our scene and t a 3x1 translation vector. Once again, every operation is independent so it can easily be parallelized.

D. Computing the Error

Finally to compute the error we compute the difference between each point of our newly transformed scene and the set of corresponding closest points.
For the same reasons as the last step, this can also be parallelized.

III. Baseline GPU Implementation

There is not much to say about the baseline GPU implementation. A matrix class has been added in which all the operations necessary to perform the algorithm are written. There is no discernable difference between the two algorithm other than the fact that many operations on the matrices were parallelized to take full advantage of the GPU.

However, this version was not as efficient as the CPU version, Indeed, even if each operation was performed faster thanks to the power of parallelization, it took too much time to move the data from host to device and vice-versa. Thus we decided to not keep this version and instead we started making real improvements by focusing on the real bottlenecks. To do so, we used a profiler and a benchmarking library.

After studying the bottlenecks of the algorithm, it was easier to identify the parts that needed to be improved. The most obvious ones were the numerous loops that were iterating on the points of the cloud that can be performed simultaneously.

Essentially, the baseline GPU implementation uses the same algorithm as in the CPU version with the exception of the parallelization made possible thanks to CUDA.

IV. Analysis of performance bottlenecks

The algorithm covered is a very naive one: it is largely optimizable through parallelization. Theoretically, the complexity of this algorithm is very dependent on the size of the clouds of points that are to be studied. The bottlenecks are then easily detectable: they are the loops that run through all these points. Parallelization is thus perfect to greatly improve the computation time and CUDA is very useful for this task.

The study of execution times and thus the profiling of the CPU program can tell us concretely what are the bottlenecks of the project. We used the Valgrind's profiler, Callgrind, to study the exact execution time of each method of the baseline CPU implementation. The result (figure 1) is very clear-cut: The method that determines the corresponding points represents 99% of the execution time of the program.

Specifically, we can observe on the figure 2 is the fact that it is the determination of the distance between points that takes more than 50% of the execution time of the program. this part of the program performs several sums and selects columns of the matrix of points in a recurrent way.

Consequently, computing all distances between each point at the same time, thanks to parallelization, would allow us to gain a lot of execution time.

V. Improvements over the CPU baseline

For our benchmarking purposes, we used Google's benchmarking library. The best indicator of performance for an algorithm is first and foremost its execution time. This library allows to measure just that with a separation between the total time and the CPU time. A cloud of approximately 3000 points was tested.

TABLE I
Comparison between the CPU and GPU versions of the main algorithm

Method	Time	CPU Time	Improvement Factor
cpu_icp	63934 ms	63924 ms	/
gpu_icp	19025 ms	19012 ms	3.36

As it has been highlighted in table VII, the execution time has been divided by 3.27 by porting some parts of the code to GPU. In the next section we will go in detail on every part that has been optimized.

A. Finding the Corresponding Closest Point

Instead of a sequential computing of each closest point to the scene using the CPU, a kernel function lets each thread in the GPU perform one calculation at the same time. The result is a very significant drop in execution time.

TABLE II
Comparison between the CPU and GPU versions of the method that finds the corresponding closest points

Method	Time	CPU Time	Improvement Factor
cpu_closest	9100 ms	9099 ms	/
gpu_closest	3063 ms	2968 ms	2.97

As we can see in table VIII, porting this method on GPU is very much worth it. For a single iteration of our algorithm, it takes almost 9 seconds for the CPU version to compute the corresponding closest points matrix while it takes nearly 3 times less time for the GPU to perform the same task.

We can also note that in the GPU version, most of the time is spent on CPU.

B. Finding the Alignment

Finding Alignment is the algorithm that computes the transformation factors, applies them to the scene matrix and returns the overall error in the computation. In the same fashion as the first step, we optimize by getting rid of the loops and replacing it with the parallelization.

Two specific parts of the algorithm have been ported on GPU. First the centroid computation.

TABLE III
COMPARISON BETWEEN THE CPU AND GPU VERSION OF THE CENTROID COMPUTATION

Method	Time	CPU Time	Improvement Factor
cpu_centroid	1.37 ms	1.37 ms	/
gpu_centroid	2.46 ms	2.46 ms	0.56

The results (cf. table III) turned out to be unexpected. Indeed, the GPU version takes almost twice as much time to be executed than the CPU version. This is probably due to the fact that one of the operations that is performed is a sum which is not as efficient on GPU when the number of points in the set is not very large (2903 in our benchmarks). In this case the version that has been selected is the CPU one.

The other part that has been ported to GPU is the error computation between the new matrix after applying the transformation and the corresponding closest points matrix.

TABLE IV
COMPARISON BETWEEN THE CPU AND GPU VERSIONS OF THE ERROR COMPUTATION METHOD IN *find_alignment*

method	Time	CPU Time	Improvement Factor
cpu_error_alignment	8.84 ms	8.84 ms	/
gpu_error_alignment	1.80 ms	1.80 ms	4.91

The error computation in *find_alignment* is 5 times faster (cf. table IV), meanwhile the time spent on the GPU is negligible compared to the CPU.

When we combine everything we come to the following results for *find_alignment*:

TABLE V
COMPARISON BETWEEN THE CPU AND GPU VERSIONS OF *find_alignment*

method	Time	CPU Time	Improvement Factor
cpu_find_alignment	15.4 ms	15.4 ms	/
gpu_find_alignment	5.69 ms	5.68 ms	2.71

The results of our benchmarks on *find_alignment* (cf. table V) are indisputable: the algorithm is 3 times faster and that is for a single iteration. Thus given a number n of iterations, this version allows to gain n times 4.74 milliseconds in total.

C. Computing the Error

The computation of the error is another loop that has been replaced by a kernel function.

TABLE VI
COMPARISON BETWEEN THE CPU AND THE GPU VERSIONS OF THE ERROR COMPUTATION METHOD

method	Time	CPU Time	Improvement Factor
cpu_error_compute	8.69 ms	8.69 ms	/
gpu_error_compute	1.19 ms	1.19 ms	7.30

Here again in table VI it is highlighted that the GPU version is faster (more than 7 times).

VI. ANALYSIS OF PERFORMANCE BOTTLENECKS ON GPU

The analysis of the baseline GPU implementation demonstrates exactly what we expected. Merely switching from the CPU to the GPU results in a 3-fold improvement on the tests performed on sample data with 3,000 points each in the scene and model. However, there are several features that can be improved on in the basic GPU implementation.

First of all, the main loop computing the closest points could be better parallelized. If done successfully we could theoretically improve the execution time by a consequent factor. Indeed, currently only the inner loop is parallelized, which is a considerable shortfall in terms of computing time. If we engage in a parallelization of the surrounding loop, however, if would mean that we would have a memory limit on large point clouds since this operation would be very memory intensive.

We could also implement a KD tree, but our research shows that its implementation would not guarantee significant improvements in terms of algorithm convergence since its results are less accurate.

We have improved the function that computes the best alignment for each point in the scene. However, this improvement is almost negligible compared to the nearest neighbor search. Thus, improving it again in the optimized version would be a burden without any consequent improvement.

Bottlenecks detected on CPU have been corrected so we analyzed the baseline GPU implementation to detect new bottlenecks and unsurprisingly the most time-consuming function is once again the one computing the distances between each points. To be precise, for the nearest neighbor search, the biggest bottleneck seems to be the memory copying of the points in device memory and the device synchronization. Besides, table table X show that in terms of CUDA API calls, more than 70% of the execution time is dedicated to cudaDeviceSynchronize and 12% are dedicated to MallocPitch. These observations reveal that the main axis of optimization revolves around matrix allocation and threads synchronization. Thus, parallelization optimization and batching would consequently improve the performance

of the program.

An additional performance test was performed to determine the optimal block size for GPU implementations, and it turns out that blocks around 128 to 512 worked best. We chose to carry on with 256 (16x16) threads per block because this is on average the optimal block size for the basic implementation.

VII. Optimized GPU Implementation using Batching

A. The 2D Optimization concept

The optimized version of our GPU implementation will, as previously stated, try to make the dual search loop of the closest point between the model and the scene in the *find-corresponding* function as much parallel as possible. Instead of using a single thread to find the match with the closest model point to the scene points cloud, here we will parallelize along the lines for a given point of the scene and all the points in the model and following the columns for a given point of the model and all the points of the scene.

$$distance(i, j) = Model(j) - Scene(i)$$

B. Limits

By applying this optimization, a clear improvement in performance has been achieved, especially on the example "Cow" or models of similar size. However, the calculation of ICP distorted the results for models with more points such as "Horse". Indeed this method has a limitation since it requires to keep in the GPU memory all the distances between all the points of the model and all the points of the scene. This allows to extract the closest point to a given point needed to compute the transformations (rotation, translation and scale) that will be applied to the scene for the next iteration.

Given a model of N points and a scene of N points, the peak of simultaneous memory storage is $O(N^2)$.

C. Batching

The memory constraint led us to parallelize our calculations independently, using batches of several lines in order to maintain the optimization even on large models. This batching was very simple to set up, and the computation of the optimal size of the batch was found empirically after a multitude of tests combining various values of block and batch sizes. The optimal value based on our tests is 1280 $(32 * 40)$ and a block dimensions of $(32, 32)$.

The batching implementation used, is not fully naive because it allows to cover the maximum GPU memory: while there is still some available GPU memory, the GPU will then run additional batches until saturation, which makes the optimization insensitive to the GPU memory available on the hardware (as long as the storage size is a $O(N)$). Afterwards, it will process all the batches in parallel according to the initial idea of this optimization.

In addition, our batching algorithm also provides a slight performance improvement compared to the bare optimization, which takes all the points and processes them at once.

VIII. Improvement of GPU Optimization

A. Improvements

The observed improvements are clearly more remarkable than the naive GPU and therefore CPU version.

If we look more in depth, the optimized feature seems to be the major bottleneck still present on the naive GPU version, hence the significant performance improvements introduced by the optimized version. As you may see in the Table XXII.

B. Results

TABLE VII
Comparison between the GPU Naive and GPU Optimized versions of the main algorithm

Method	Time	GPU Time	Improvement Factor
gpu_naive_icp	19025 ms	19012 ms	/
gpu_optimized_icp	109 ms	109 ms	174.4

TABLE VIII
Comparison between the GPU Naive and GPU Optimized versions of the method that finds the corresponding closest points

Method	Time	CPU Time	Improvement Factor
gpu_naive_closest	3063 ms	2968 ms	/
gpu_optimised_closest	7.67 ms	7.67 ms	386.96

The improvement factors shows us that we are more than 386 times faster compared to the naive GPU version on the part of the algorithm that computes the closest corresponding points.

And in a general way on all the algorithm we reach an improvement factor on the whole algorithm of more than 174 times faster than the naive version, which proves that the bottleneck that needed to be improved was the function that computes the closest corresponding points.

IX. Distribution of work in the team

TABLE IX
Team work repartition

/	Amrouche	Heffar	El Habbari	Ramdani
ICP and utils	40%	20%	20%	20%
CPU baseline	20%	25%	35%	20%
GPU baseline	20%	30%	15%	35%
GPU Parallelization	10%	40%	10%	40%
Benchmark	30%	5%	60%	5%

References

[1] S. Elhabian, Amal Farag, Aly Farag, "A Tutorial on Rigid Registration: Iterative Closest Point (ICP)," Louisville, KY: University of Louisville, March 2009

X. Annexes

A. Tables

TABLE X
PROFILING THE NAIVE GPU VERSION OF THE ALGORITHM ON COW FILE (2904 POINTS)

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	97.06%	9.29333s	20321	457.3352μs	453.8052μs	587.4052μs	find_min_distance_naive
	2.24%	214.64ms	40733	5.269052μs	896ns	15.29652μs	[CUDA memcpy HtoD]
	0.40%	37.964ms	20321	1.868052μs	1.824052μs	7.552052μs	compute_distance_naive
	0.30%	28.630ms	20356	1.406052μs	608ns	7.104052μs	[CUDA memcpy DtoH]
	0.00%	25.28152μs	7	3.611052μs	3.520052μs	3.968052μs	compute_err
	0.00%	22.59252μs	14	1.613052μs	1.568052μs	1.952052μs	substract_col
	0.00%	12.99252μs	7	1.856052μs	1.792052μs	2.176052μs	y_p_norm
API calls:	70.21%	9.49441s	40663	233.4952μs	1.895052μs	725.0752μs	cudaDeviceSynchronize
	12.97%	1.75373s	40733	43.05452μs	1.910052μs	133.89ms	cudaMallocPitch
	8.87%	1.19894s	81375	14.73352μs	1.540052μs	2.4696ms	cudaFree
	3.65%	492.98ms	40768	12.09252μs	4.195052μs	284.2052μs	cudaMemcpy2D
	1.88%	254.48ms	20321	12.52252μs	7.863052μs	284.4052μs	cudaMemcpy
	1.64%	221.42ms	40670	5.444052μs	3.535052μs	274.8452μs	cudaLaunchKernel
	0.79%	106.60ms	40642	2.622052μs	1.981052μs	288.8552μs	cudaMalloc
	0.00%	189.1852μs	101	1.873052μs	253ns	76.41452μs	cuDeviceGetAttribute
	0.00%	85.22752μs	1	85.22752μs	85.22752μs	85.22752μs	cuDeviceTotalMem
	0.00%	23.81352μs	1	23.81352μs	23.81352μs	23.81352μs	cuDeviceGetName
	0.00%	7.429052μs	1	7.429052μs	7.429052μs	7.429052μs	cuDeviceGetPCIBusId
	0.00%	1.507052μs	3	502ns	267ns	965ns	cuDeviceGetCount
	0.00%	904ns	2	452ns	249ns	655ns	cuDeviceGet
	0.00%	353ns	1	353ns	353ns	353ns	cuDeviceGetUuid

TABLE XI
PROFILING THE OPTIMIZED GPU VERSION OF THE ALGORITHM ON COW FILE (2904 POINTS)

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	61.76%	29.304ms	21	1.3954ms	1.1357ms	1.6427ms	find_Y
	35.61%	16.895ms	21	804.52μs	287.69μs	1.0637ms	compute_distance
	1.72%	814.68μs	154	5.2900μs	2.4320μs	8.5440μs	[CUDA memcpy HtoD]
	0.69%	325.25μs	70	4.6460μs	2.4960μs	7.6160μs	[CUDA memcpy DtoH]
	0.13%	63.744μs	14	4.5530μs	4.4480μs	5.0240μs	compute_err
	0.06%	28.448μs	14	2.0320μs	1.9840μs	2.4320μs	substract_col
	0.03%	16.480μs	7	2.3540μs	2.2720μs	2.6560μs	y_p_norm
API calls:	72.53%	135.48ms	196	691.22μs	1.8760μs	129.40ms	cudaMallocPitch
	17.01%	31.776ms	56	567.43μs	6.8590μs	1.6468ms	cudaDeviceSynchronize
	6.61%	12.350ms	224	55.133μs	5.0530μs	700.42μs	cudaMemcpy2D
	3.39%	6.3309ms	196	32.300μs	1.5660μs	309.87μs	cudaFree
	0.32%	602.90μs	77	7.8290μs	4.2000μs	37.340μs	cudaLaunchKernel
	0.07%	125.71μs	101	1.2440μs	112ns	54.243μs	cuDeviceGetAttribute
	0.04%	76.023μs	1	76.023μs	76.023μs	76.023μs	cuDeviceTotalMem
	0.02%	29.095μs	1	29.095μs	29.095μs	29.095μs	cuDeviceGetName
	0.01%	14.565μs	2	7.2820μs	138ns	14.427μs	cuDeviceGet
	0.00%	7.6520μs	1	7.6520μs	7.6520μs	7.6520μs	cuDeviceGetPCIBusId
	0.00%	1.0530μs	3	351ns	150ns	726ns	cuDeviceGetCount
	0.00%	231ns	1	231ns	231ns	231ns	cuDeviceGetUuid

TABLE XII
EXECUTION TIME SUMMARY ON COW FILE (2904 POINTS)

Method	Execution time				
	CPU TO GPU OPTIMISATION			GPU OPTIMISATION	
	CPU	*GPU*	*Improvement Factor*	*GPU OPTIMIZED*	*Improvement Factor over CPU*
Loop	63934 ms	19025 ms	3.36	109	586.55
Closest	9100 ms	3063 ms	2.97	7.67	1186.44
Centroid	1.37 ms	2.46 ms	0.55	/	/
Error Alignment	8.76 ms	1.76 ms	4.97	/	/
Find Alignment	15.2 ms	4.74 ms	3.21	/	/
Error Compute	8.44 ms	1.16 ms	7.26	/	/

B. *Figures*

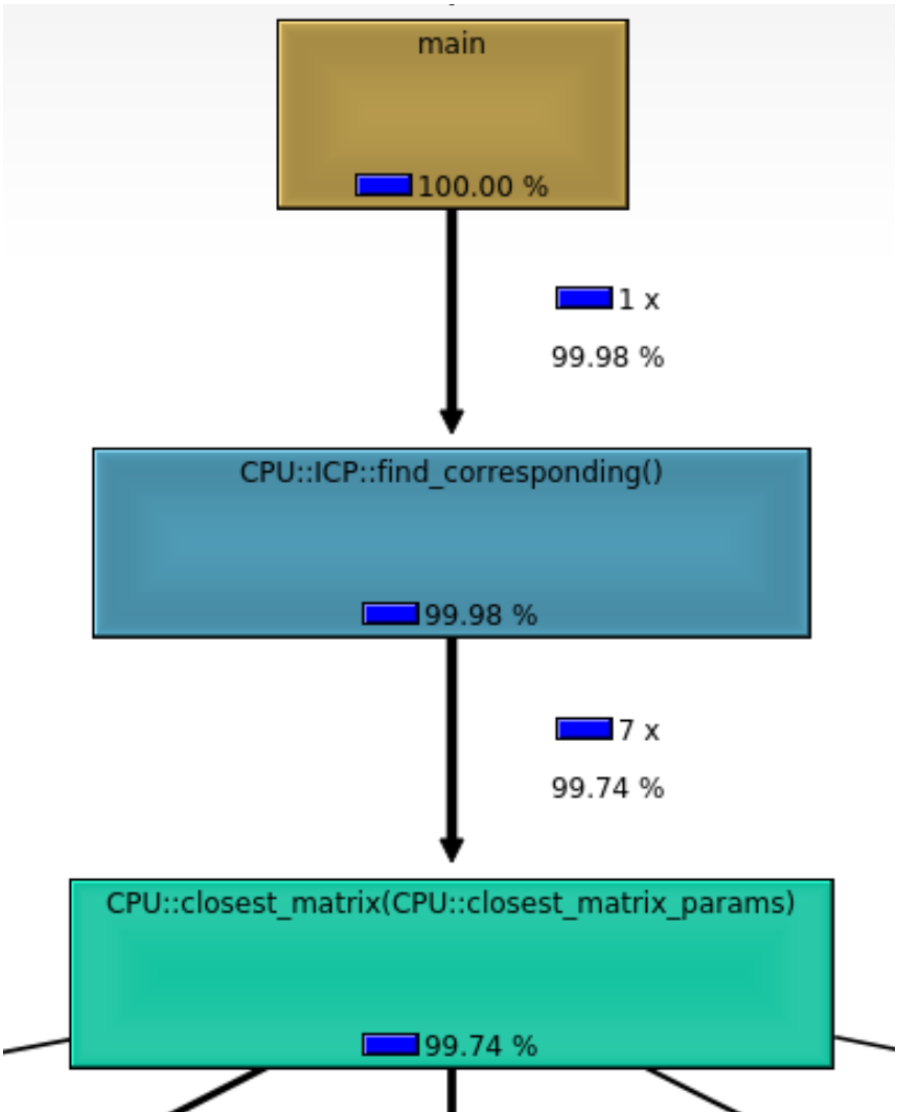

Fig. 1. Profiler's graph of CPU program (part 1).

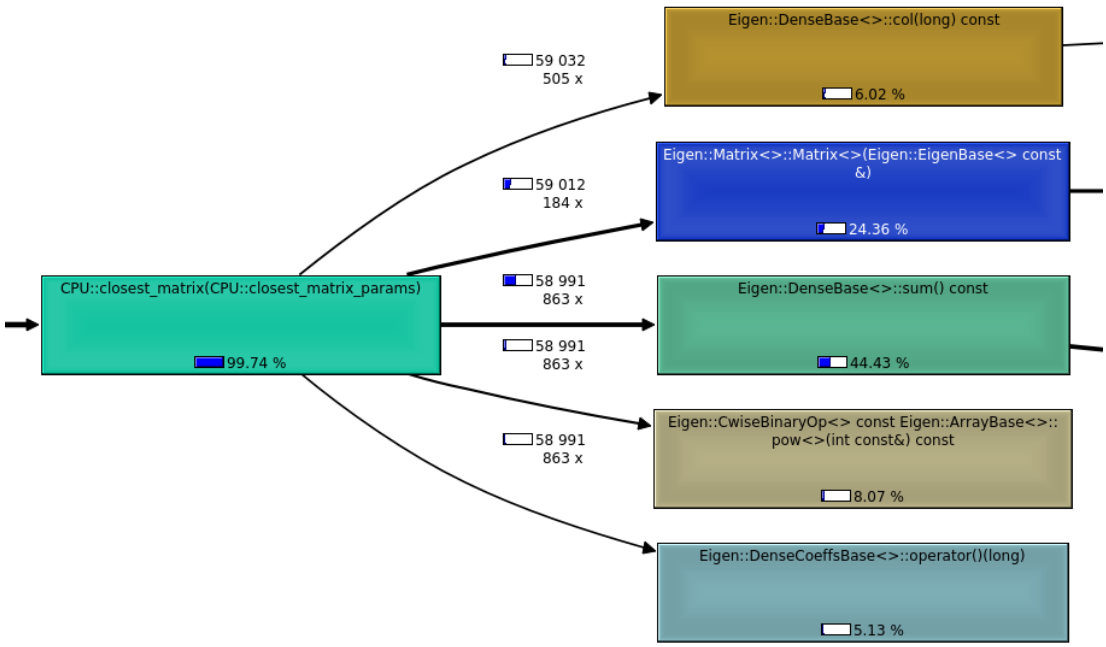

Fig. 2. Profiler's graph of CPU program (part 2).