Iterative Closest Point (ICP) with CUDA Parallelization

Long Qian (lqian8)

December 15, 2016

1 Objective

Iterative Closest Point (ICP) algorithm [1] is a well-known procedure for finding the registration between 3D point clouds without correspondance information. It is routinely applied in the domain of medical imaging and object tracking. As a PhD student in Laboratory of Computational Sensing and Robotics (LCSR) [1], I am often faced with the problems that involve ICP. However, the traditional ICP algorithm is not fast enough to be realtime when the point cloud is large, which restricts the usage of ICP in many realtime-critical applications, e.g. tracking. Therefore, it is beneficial to implemented a paralleled ICP that takes advantage of the efficiency of GPGPU. In section 2, the implementation of normal and paralleled ICP is described. In section 3, the speedup and scaleup property of paralleled ICP is demonstrated and discussed. Section 4 concludes the report.

2 Implementation and Observation

Traditional ICP

A traditional ICP algorithm is first implemented via Python [2] programming language with the help of Numpy [3] package. A brief algorithm diagram is listed in Alg. 1.

Data: Point Clouds dst and src
Result: Point Cloud res: a transformation of src, close to dst
begin
  Initialize $res = src$;
  while *res is close enough to dst* **do**
    For each point p_i in res, find the closest point q_i in dst, stack all q_i to form a new point cloud tgt;
    Compute point cloud registration R, T from res to tgt, update point cloud res with the transformation R, T;
  end
end

Algorithm 1: Iterative Closest Point Algorithm

The traditional ICP with 1024 points and maximum 20 iterations is executed, and the program is profiled with cProfile [4] tool in Python. Profiling data shows that the step of finding correspondence takes 104.684s of the total 104.956s, which is 99.75%. According to Amdahl's law, if this part of the program can be paralleled, the efficiency of the algorithm can be increased to a great extent.

Paralleled ICP

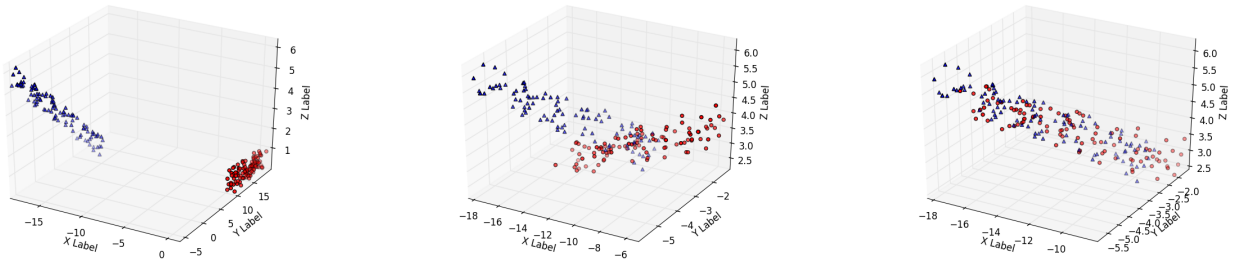
Motivated by the low efficiency of finding correpsondence, I implemented a paralleled function of correspondence finding in CUDA. PyCuda [5] is used as the interface between Python and CUDA C program. By observation, for each point in the point cloud src, all the points in the dst is iterated through, and the distance between them are calculated. Then the point with minimum distance is selected. Therefore, the CUDA part is designed into three parts:

1. Initialization: Load the dst point cloud into the global memory of CUDA environment, so that each thread can access it without unnecessary copying.

[1] LCSR: `https://lcsr.jhu.edu`
[2] Python: `https://www.python.org`
[3] Numpy: `https://www.numpy.org`
[4] cProfile: `https://docs.python.org/2/library/profile`
[5] PyCuda: `https://mathema.tician.de/software/pycuda`

2. Find correspondence: The actual function to find the correspondence point in dst for a single point in src, save the correspondent point in res matrix, and output the minimum distance.
3. Reduction: Sum up total distances computed by each thread. This value is used to determine whether the loop should be terminated.

Similar to traditional ICP, an experimental run of 1024 points, 20 iterations and 1 thread is executed, to validate the program. Fig. 1 demonstrates the result of iteration 0, 2 and 10, using Matplotlib [6]. Note that the total time with 1 thread is 3.578s, which is already significantly better than traditional ICP, because of the better efficiency of C execution than Python. In the next section, the speedup and scaleup property of the paralleled ICP will be investigated.



(a) Iteration 0 (b) Iteration 2 (c) Iteration 10

Figure 1: Paralleled ICP at Several Iterations

3 Results and Discussion

The paralleled ICP is run under multiple configurations, with various number of points in point cloud, and various number of threads.

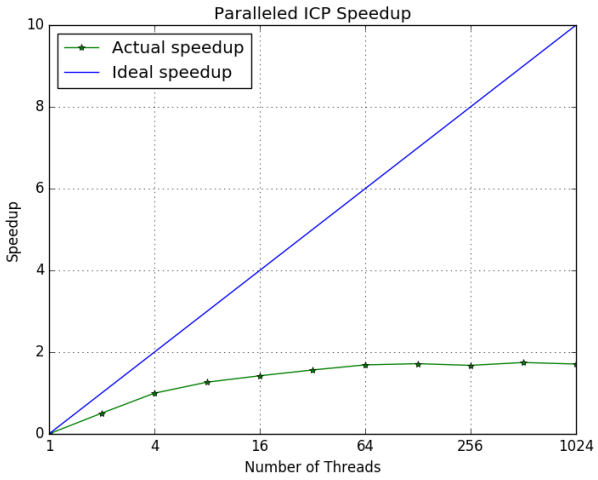


Figure 2: Speedup Diagrams of Paralleled ICP

Speedup Analysis

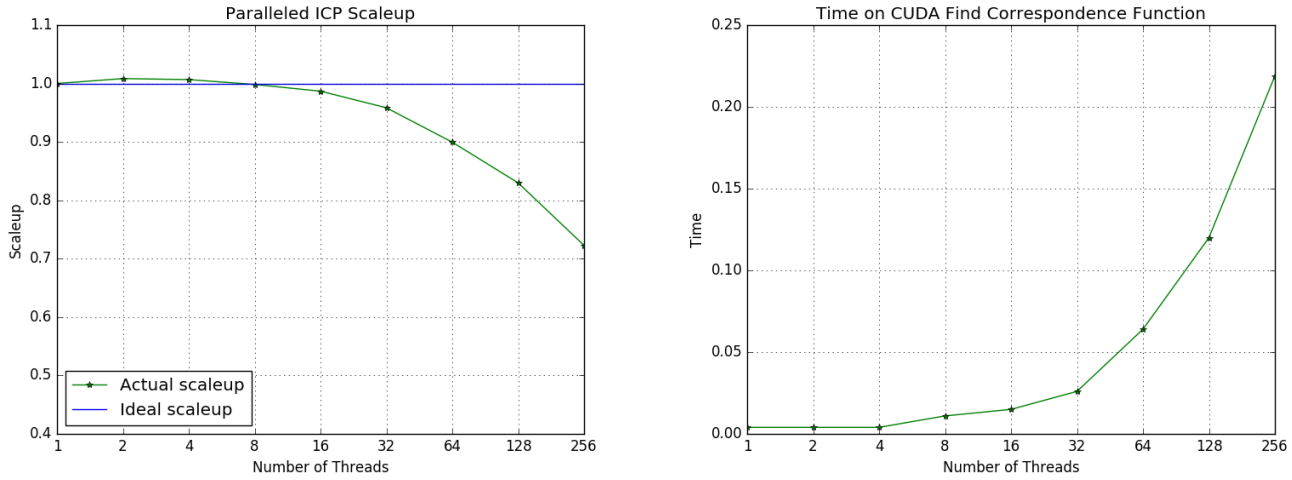
A speedup experiment is run and analyzed, with point cloud of 1024 points running with 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 threads. The speedup diagram is plotted using Matplotlib, and shown in Fig. 2. The ideal speedup (blue line) is a linear function with respect to the number of cores used, and the actual speedup is the green plot. It is obvious that the speedup of paralleled ICP is sublinear. The sublinear property is caused by the startup cost and interference, for several reasons:

[6] Matplotlib: `http://matplotlib.org/`

1. The startup cost is high. Let's take a look at the profiling result of 1024 core implementation. The total time is 0.648s, only 0.245s (37.81%) is spent on the `solve` function, which is the ICP algorithm, all other parts are startup cost, including saving dst matrix to CUDA environment, compile CUDA module, create point cloud object, etc.
2. The interference is high. The process of finding correspondence involves no interference at all, because the dst object is global and constant, therefore, each point in src point cloud is treated separately, without any data dependency among different points. However, the reduction process takes 0.224s of the total 0.648s (35.57%). There is an implicit barrier in this part.
3. The skew is small. Since the finding correspondence procedure is highly paralleled, and involves no dependency except for the reduction. Each thread has almost same amount of work. Thus, the skew is negligible.

Scaleup Analysis

A scaleup experiment is run and analyzed as well, with (number of points, number of threads) in (16, 1) , (32, 2) , (64, 4) , (128, 8) , (256, 16) , (512, 32) , (1024, 64) , (2048, 128) , (4096, 256). The scaleup diagram is shown in Fig. 3a.



(a) Scaleup Diagram

(b) CUDA Finding Correspondence Time

Figure 3: Scaleup Diagrams of Paralleled ICP

It can be observed that, when the number of threads is increased from 1 to 16, the performance is almost linear, but if the numebr of threads and taskload are further increased, the performance is becoming sublinear. This is because, with the increased point number, the finding correspondence procedure takes more time to iterate over all the points in the point cloud, thus each thread takes more time to execute. The result is very clear when we look at the time spent exactly on the CUDA function `find_closest`. It can be accessed via observing the accumulated time on the function `function_call` of `driver.py`. It is plotted in Fig. 3b. In the scaleup experiment, 16 points in src point cloud is assigned to one single thread, and each point in src is compared to all the points in dst. The length of dst is a determinant factor in the time consumed in each thread. The problem has a complexity of $O(n^2)$, but the number of thread is n. This is the main reason for sublinear scaleup property.

4 Conclusion

In this project, a traditional Iterative Closest Point (ICP) algorithm is implemented first. After identifying the main bottleneck of the traditional implementation (find correspondence), PyCuda framework is utilized to parallelize the step of find correspondent point. The speedup and scaleup property of paralleled ICP is explored by multiple experiments. The sublinearity of speedup is because of the startup cost, and the interference at the reduction stage. The sublinearity of scaleup is because of the quadratic complexity of the algorithm itself. Python profiling tool is used a lot to validate the hypothesis and observations.

References

[1] P. J. Besl and N. D. McKay. A method for registration of 3-d shapes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(2):239–256, Feb. 1992.