EE-5351 Project Report
Localization using GPU based Iterative Closest Point (ICP) algorithm

Amol Vagad
vagad001@umn.edu
Aniket Pendse
pends005@umn.edu

I. Introduction

The autonomous vehicle (AV) industry has made progress at an exponential rate in the past few years. Though this technology seems very interesting it is equally complex due to the combination of algorithms and sensors. To facilitate smoother operation of the AV's it is important that they are able to detect and follow a desired path on highways and other roads. Though GPS with inertial measurement units (IMU) provide information about the position of the vehicle they are not accurate enough to help it drive itself. Thus to overcome this flaw ,high resolution cameras, LIDARs, Radars and High Definition (HD) maps are used by the AV's for localization. The accuracy of these sensors and data-sets is very high and the error is reduced to merely few centimetres. To use the outputs from these sensors and the data of HD maps it is necessary to have a robust algorithm. The data obtained from the LIDAR as well as the HD map tile is 3 dimesnsional (3D) in nature. The HD map data consists of various localization assets like barriers, poles,sign boards and lanes. The LIDAR also detects such assets depending upon the number of channels. Now for the LIDAR to validate the detection with the assets of the HD maps it is necessary to have a robust algorithm.

One of the most widely used algorithms for localization is called the Iterative Closest Point (ICP) algorithm.Iterative Closest Point (ICP) is an algorithm used to minimize the difference between 2 sets of 3D point cloud data. It gives the rotation and translation required to be applied to one point cloud data-set(source) in order to match it with the other(reference). The matching parameter used is nothing but a mean square error between the 2 point cloud data-sets. So the general process involved in ICP is you apply the transformation R and t to the source point cloud data-set P's and calculate the mean square error between the transformed point cloud data-set and the reference point cloud data-set. The objective is to minimize the mean square error between the 2 point cloud data-sets for a certain Rotation R and translation t.

Implementing this algorithm is not a complex task. However, in real-time conditions for an AV it is highly important to have the algorithm to produce quick and efficient results. Despite great advancements in the CPU architectures over the years it is still not possible to implement such an algorithm to handle huge amount of data (approximately 20,000 3D points in one HD map tile) using just the CPU. Thus we need to use the processing power of a GPU to implement parallelism on every point on a data-set thereby reducing the total operation overhead. The development AV's are equipped with powerful GPU's to carry out such a task. However, their highly limited synchronization and restricted memory model means that performance rapidly decreases with branching threads,random memory access and data-access synchronization [1].This is particularly relevant for occupancy grid based ICP algorithms, as there is a need for hundreds of threads to add scan points and modify the grid in parallel without causing data corruption [2].

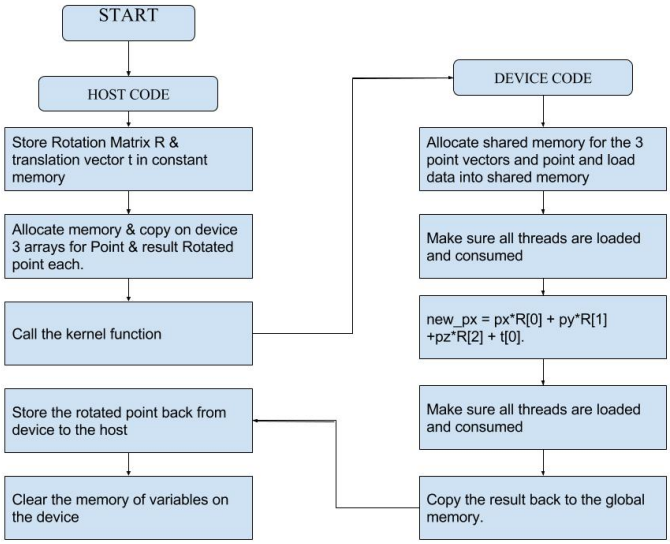
For this study, we first developed the ICP algorithm with just a CPU. The results obtained were satisfactory as the error was low. However, the time taken by this implementation was very high(approximately 40 seconds/iteration). Thus the target was to develop a GPU based version that provides at least 15x improvement to manage a realistic use for a AV development vehicle. The development was carried out on a NVIDIA 1080 GPU at VSI LAbs , Saint Louis Park facility. For reference point cloud data set, HD map tiles from HERE were used. Since, we could not implement reading point cloud data from LIDAR due to time constraint we have used the same HD map tile data but transformed it by a scaling factor. Finally, the objective is to match the transformed data back with the original data. CUDA programming techniques, optimization methods and concepts learned during the course were applied during development of this project.

II. Design Overview

The basic idea of the design was to first read the 3D point cloud data from a CSV (Comma Separated Variables) file into our structure called point-cloud-data which consists of attributes like the x,y, and z coordinates,size and index for every point. This becomes our model data-set or the reference data-set. To obtain the measurement data-set we just apply transformation on the model data to simulate the measurement data. To perform this transformation we developed a cuda kernel function that performs rotation and translation on the GPU. Thus, we can perform transformation on a large chunk

of data at a time thereby reducing the time required to generate the data. This operation is not needed if we were taking inputs directly from a LIDAR. Method of tiling is applied to perform the multiplication of Rotation matrix with the input 3D point. The rotation matrix R and translation vector t are stored in the constant memory to avoid multiple global memory accesses since they have constant values throughout the operation. The 3 coordinate vectors of input data are loaded into the shared memory to avoid global memory access overhead.

Figure 1. Flow chart of transformation kernel



Once the measurement data is generated, it is then used against the model data to calculate the closest point iteratively. Firstly we declare a 4x1 vector called rt which stores the rotation angle as its first element and the x,y and z coordinates for translation as the succeeding 3 elements. This vector is input to the closest point calculation function. Inside the closest point calculation function, using the elements of the vector rt we then generate the rotation matrix R.

III. Implementation

A. Transformation Kernel

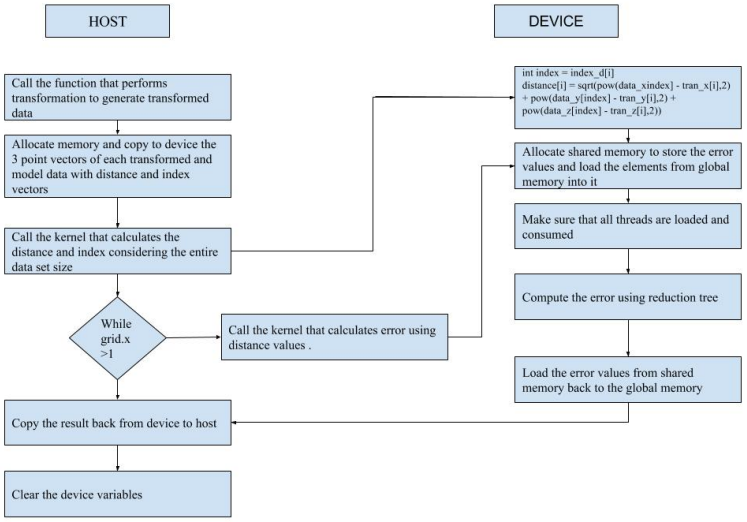
The input to this kernel are the x, y and z coordinates of the input point cloud and the output is stored in the x,y and z coordinates of the transformed point cloud. Once the kernel is called first all the x, y and z coordinates of the input data in a block are stored in shared memory. This is because each of the x,y and z coordinates of a point are used 3 times to calculate the x,y and z coordinates of the transformed point. Further, the elements of rotational matrix and the translation vector are stored in constant memory because throughout the transformation kernel they are constant and are called repeatedly inside the kernel. The transformation equation is given as follows

$$x_{out} = x_{in} * R_{1,1} + y_{in} * R_{1,2} + z_{in} R_{1,3} + t_1 \quad (1)$$

$$y_{out} = x_{in} * R_{2,1} + y_{in} * R_{2,2} + z_{in} R_{2,3} + t_2 \quad (2)$$

$$z_{out} = x_{in} * R_{3,1} + y_{in} * R_{3,2} + z_{in} R_{3,3} + t_3 \quad (3)$$

Figure 2. Flow chart of the error calculation function



Once the above operation is performed the values are stored in the transformed x, y and z coordinate arrays using the thread indices. Care is taken to make sure that the memory accesses are coalesced while storing data in shared memory and the divergence is as low as possible.

B. Distance and Index Calculation Kernel

This kernel is called in the function used to find closest point. In this kernel we calculate the distance of all the individual points in the model data with a single point in the measurement data point cloud and also store the index of each model point in the index array. This operation is a prerequisite to find the closest point in the model data point cloud. The inputs to this kernel are x,y and z coordinates of all the model data points and the x, y, z coordinates of the point of concern from the measurement data. We use the Euclidean distance formula to calculate the distance

$$dist = \sqrt{(x^i_{mod} - x_{meas})^2 + (y^i_{mod} - y_{meas})^2 + (z^i_{mod} - z_{meas})^2} \quad (4)$$

The distance corresponding to each model data point is stored in a distance array. Also at the same time the thread index for each point are stored in the index array. Since we need to perform only a single operation there is no need of using shared or constant memory. Also the memory accesses are coalesced since each thread in a warp is made to access adjacent memory and there is no divergence since all the threads in a wrap perform the distance calculation operation.

C. Closest Point Index Kernel

As the name suggests this kernel is used to determine the closest point index for each point in the measurement data. It uses the distance array and the index formed after calling the previous kernel. The input to this kernel are the distance array and the index array and the output is the smallest distance and the corresponding index. It uses a reduction tree algorithm to

find the smallest distance and save the index. The distances and indices are stored in a shared memory twice the size of the block size. So each thread in a block loads 2 elements. Once all the elements are stored in the shared memory each thread compares 2 elements at positions t and t + stride and stores the smaller distance at position t. Here t represents the thread index and stride is the offset which starts with a value equal to block size. This comparison is run in a loop and the stride value is halved every iteration until it becomes 1. As the smaller distance values are stored the corresponding index values are also stored along with it. So at the end only the smallest distance and corresponding index remain at the zeroth position for each block. This value is stored back at the block index position in the distance and index array. Coalesced memory accesses take place since adjacent threads saves adjacent elements in the shared memory. And divergence is avoided by using stride.

D. Distance Between Model and Measurement Data Points

This kernel is called in the function to find the distance error between the 2 point clouds. This kernel is used to calculate distance between each measurement data point with the corresponding closest point from the model data point cloud referenced by the index array. So input for this kernel are the x, y and z coordinates of all the measurement data points and the x,y and z coordinates of all the model data points. In addition to this we also pass the index array which stores the indices of the closest point in model data point cloud for each point in measurement data. The distance formula is similar to the one given in Eq.(4).

$$dist = \sqrt{(x_{mod}^{ind_i} - x_{meas}^{i})^2 + (y_{mod}^{ind_i} - y_{meas}^{i})^2 + (z_{mod}^{ind_i} - z_{meas}^{i}} \quad (5)$$

Since, the model data is referenced by the index array the accesses to the model data point cloud are not coalesced however, rest accesses are coalesced and the divergence is also very low.

E. Calculate Total Error

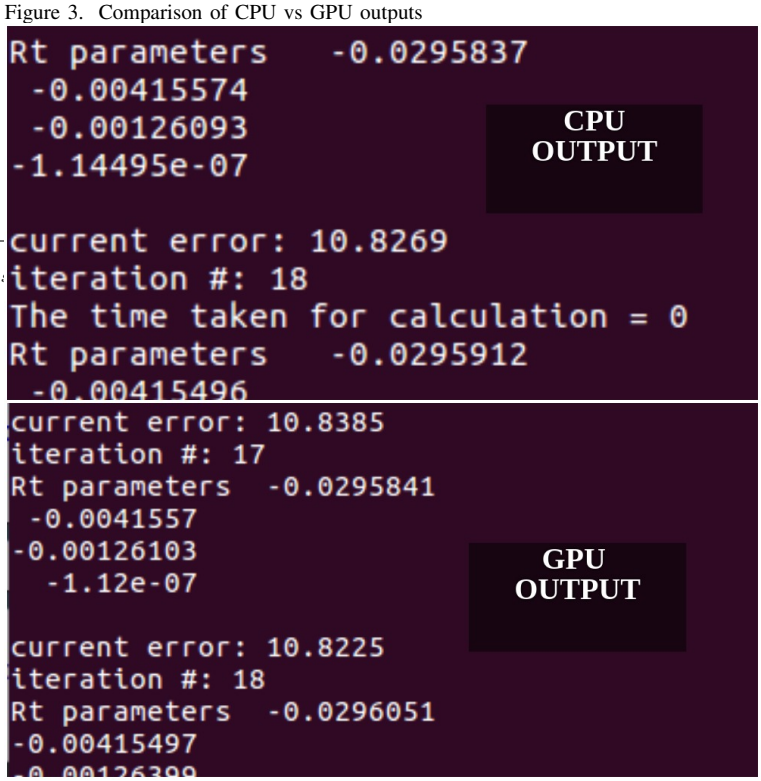
This kernel takes the distance array formed by the previous kernel call and calculates the sum of all the elements in this array. For this purpose we first store twice the block size number of elements in the shared memory. Each thread loading 2 elements in shared memory blockDim.x distances apart. Once the distances are stored we apply a reduction tree algorithm to calculate the sum. We take 2 elements each at a time stride distance apart (that is with index t and t + stride) and add them and store them at lower indices. The stride value is halved every iteration and this process is run until stride is equal to one. This kernel stores one summed value per block and need to be called repeatedly until we get total sum of all the distances. The output of this kernel is passed to the optimizer for optimization.

IV. Verification

As mentioned before, since the measurement data was generated by simulation i.e transforming the model data by fixed values, the accuracy of the CPU code was easy to find out. The accuracy was measured by checking the difference between the estimated values from the optimizer and the true values. To verify the correctness of the GPU code, the outputs of both the codes were simply compared. Upon comparison it was observed that the difference between the two values was of the order of 1e-5. This percentage of accuracy is acceptable for the required application. The pattern was repeated throughout the iterations thereby confirming that the accuracy stays constant throughout the implementation. Additionally when different data sets of similar type were used the accuracy was found to be consistently high among all the data-sets.

Majority of the variables in our code are double precision values due to that fact that the 3D points in the data have small values (order of 0.01). As these small values are later used to find distances using the euclidean distance formula thereafter more precision is needed. Upon switching to single point precision using floating values to reduce the memory consumption of our code we found that the results were unsatisfactory as the accuracy was heavily compromised. Thus, we need to give up on achieving more speed in order to maintain the accuracy of our implementation.

Figure 3. Comparison of CPU vs GPU outputs



V. Performance

The initial goal for obtaining the speed up was 15 times the CPU version of the code. However, it was observed that the speed up was almost 22 times on the GPU version. The time taken to complete all the 20 iterations was 746 seconds on the CPU compared to just 33 seconds on the GPU. One of the most important reason for the spike in the speed up is due to the parallel handing of the large data-set. This helps

to carry multiple operations in parallel thereby reducing the total overhead.This speed up was found to be constant on a number of data-sets of sizes of the order 20k.

Figure 4. Performance of the kernels

```
ptxas info    : 0 bytes gmem, 96 bytes cmem[3]
ptxas info    : Compiling entry function '_Z19CalculateTotalErrorPdi' for 'sm_30'
ptxas info    : Function properties for _Z19CalculateTotalErrorPdi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 15 registers, 4096 bytes smem, 332 bytes cmem[0]
ptxas info    : Compiling entry function '_Z26CalculateDistanceAllPointsPdS_S_S_S_PiS_i' for 'sm_30'
ptxas info    : Function properties for _Z26CalculateDistanceAllPointsPdS_S_S_S_PiS_i
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 30 registers, 388 bytes cmem[0], 236 bytes cmem[2]
ptxas info    : Function properties for __internal_accurate_pow
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Compiling entry function '_Z18CalculateBestIndexPdPii' for 'sm_30'
ptxas info    : Function properties for _Z18CalculateBestIndexPdPii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 22 registers, 6144 bytes smem, 340 bytes cmem[0]
ptxas info    : Compiling entry function '_Z31CalculateDistanceIndexEachPointdddPdS_S_PiS_i' for 'sm_30'
ptxas info    : Function properties for _Z31CalculateDistanceIndexEachPointdddPdS_S_PiS_i
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 32 registers, 388 bytes cmem[0], 236 bytes cmem[2]
ptxas info    : Function properties for __internal_accurate_pow
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Compiling entry function '_Z21PerformRotationKernelPdS_S_S_S_i' for 'sm_30'
ptxas info    : Function properties for _Z21PerformRotationKernelPdS_S_S_S_i
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 17 registers, 6144 bytes smem, 372 bytes cmem[0]
```

The figure 4 shows the performance of all the kernels with respect to the resource usage. When these numbers were fed to the CUDA Occupancy Calculator, we came to the conclusion that the maximum occupancy was achieved for all the kernels however the main limiting factor for all the kernels was the number of threads. Fig. 5 shows the kernels and their respective limiting factors.

Figure 5. Results from CUDA Occupancy Calculator

Kernel Name	Occupancy Percentage	Limiting Factor
Calculate Total Error	100%	Number of Threads
Calculate Distance All Points	100%	Number of Threads and Number of Registers
Calculate Best Index	100%	Number of Threads
Calculate Distance Index Each Point	100%	Number of Threads and Number of Registers
Perform Rotation Kernel	100%	Number of Threads

As we can observe for the table , all the kernels share the common limiting factor being the number of threads with the only exceptions being the calclateDistanceToAllPoints and CalculateDistanceIndexEachPoint kernels which are also limited by the number of registers. Thus, we can arrive at a conclusion that the speed up would definitely increase with increase in number of threads. So if we port to multiple GPU's or use a better GPU then we might achieve more parallelism than we have right now. Fig.6 gives us a general idea about the speed up achieved for different sizes of data. It was observed that for smaller data sizes of order 100, the accuracy of both GPU and CPU is off. The reason for this is that the optimizer convergence to a local minimum and gives incorrect values of parameters R and t. However, for data of order 1000 or greater the convergence of CPU and GPU codes are correct and as you can see the performance of GPU is much better as compared to CPU.

Figure 6. Performance of CPU and GPU for varying sizes of data

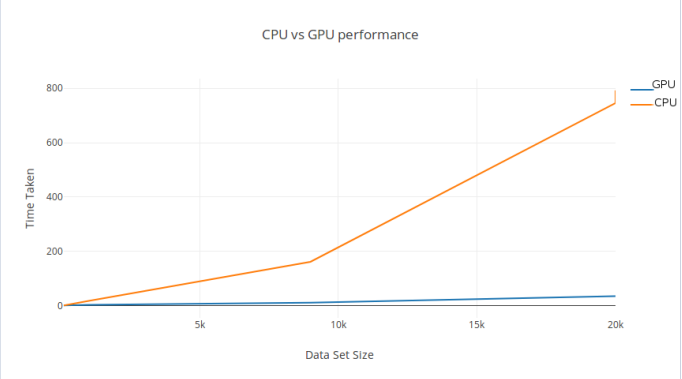


Figure 7. Throughput and Bandwidth Performance of Kernels

Kernel Name	FLOPS	Memory Bandwidth	Limiting Factor
Calculate Total Error	156.25 MFLOPS	10Gbps	Bandwidth
Calculate Distance All Points	234.3 MFLOPS	10Gbps	Bandwidth
Calculate Best Index	78.125 MFLOPS	10Gbps	Bandwidth
Calculate Distance Index Each Point	187.5MFLOPS	10Gbps	Bandwidth
Perform Rotation Kernel	4.68MFLOPS	10 Gbps	Bandwidth

Fig. 7 Shows the throughput performance of the kernels. It can be observed that the limiting factor for the kernels is the memory bandwidth since all the data transfers are being done in double precision for higher accuracy.(The test were carried out on GTX 1080 whose memory bandwidth is 10 Gbps and maximum FLOPS allowed are 5816 GFLOPS)

The system performance can also improved with better handling of data. Upon the execution of the GPU code it was found that almost 85% of the implementation was consumed by the Calculation of closest point function. A solution to this problem would be implementing a binning algorithm using octree data structure which will help to sort data into bins so we don't need to access all the data points for each data point to find the closest point to it. Using a octrees we could just look for in the current bin and the neighbouring adjacent bins for the closest point instead of the entire data-set. This algorithm was implemented on CPU and the speed up was found to be 17 times more than our regular CPU code. However, there were many obstacles while porting the algorithm on the GPU. Firstly, the octree structure was difficult to implement for the kernel code. Secondly, since we were generating bins dynamically on the CPU code we could not do it on the GPU code as we need to specify the memory in advance before transferring the data on the GPU. Supplying large memory

would again make us allocate memory unnecessarily. However, a successful implementation of GPU would certainly give an additional 15-20 times more speed up.

VI. Conclusion

The implementation was successful as we achieved the objective of 15 times speed up with 22 times speed up in practice. We were successful in using the tools of parallelism learned in the class like usage of shared and constant memory, memory coalescing, divergence reduction using strides and the implementation of reduction trees. The parallelism helped to handle the large number of 3D data points in the data-set which helped to speed up the process. However, despite the successful implementation there are still more improvements required to make the code applicable in real-time operations. Improvements can be achieved by using better data management algorithms like binning and using multiple GPU's to increase the number of threads for a kernel as these were found to be the parameters hindering the performance of the code.

The various versions throughout the development of this project can be found on our Github repository here : https://github.com/AmolVagad/Parallel-Programming-Project

master branch : CPU implementation CUDA branch : GPU implementation

VII. Acknowledgement

We are very thankful to Professor John Sartori and Teaching Assistant Raghunathan for their constant guidance and support. We are also thankful to the ECE-IT department for the access to GPU LAB computers throughout the course to run our codes.

VIII. References

[1] R. Shams and N. Barnes, "Speeding up mutual information computation using nvidia cuda hardware," in Digital Image Computing Techniques and Applications. IEEE, 2007, pp. 555–560.

[2] A. Ratter, C. Sammut and M. McGill, "GPU accelerated graph SLAM and occupancy voxel based ICP for encoder-free mobile robots," 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, 2013, pp. 540-547.

[3] CUDA Toolkit Documentation : http://docs.nvidia.com/cuda/