Accelerating Robust Normal Estimation

A comparison of CPU vs. GPU performance

Rohit Gupta & Ian Endres

HERE Technologies

rohit.gupta@here.com — +31 (40) 744 1368

Abstract

We present an implementation for calculating surface normals on the CPU and GPU.The GPU implementation can be up to an order of magnitude faster than the CPU implementation for the setup on a Desktop workstation with a 6-core (12 thread) Intel Xeon processor and a Quadro M4000 from NVIDIA with 8GB memory on board.

Introduction

In order to understand the contents of a scene from point clouds (Figure 1), it is often required to calculate normals of the surface the point cloud represents. Furthermore, this calculation is highly time consuming and recurring so optimizing it can have a significant impact on the total runtime.

Algorithm

We use the RANSAC approach for calculating surface normals on our point clouds. In Algorithm 1 we list the computational steps in our implementation.

Algorithm 1 Robust L1 normals estimation

1: Inputs
1. Point cloud: $\mathcal{X} = x_i$, N number of points
2. No. of neighbors: K_{neigh},
3. No. of normal hypotheses/point: $K_{samp} = \frac{K_{neigh}}{2}$.

Outputs
1. Surface normals/point: $\mathcal{V} = v_i$
2. Goodness of fit score/point:$\mathcal{S} = s_i$

2: Calculate nearest neighbors K_{Neigh} for each point in $\mathcal{X}$.
3: **for** i:=0,...,N **do**
4: $y_{ij} = x_i - x_j$, where $j = 0, ..., K_{neigh}$.{Compute relative vectors and store in $\mathcal{Y} = y_{ij}$}
5: $\hat{\mathcal{Y}} = \frac{y_{ij}}{|y_{ij}|}$ {Normalize $\mathcal{Y}$}
6: **end for**
7: $\mathcal{S} = 0, m, n$ Uniform_random_distribution in K_{neigh}
8: **for** k=0,...,K_{samp} **do**
9: $u = \hat{y_m} \times \hat{y_n}$, $\hat{\mathcal{U}} = \frac{u}{|u|}$ {Normalize U}
10: $s' = \sum_j |\hat{y}_j^T \hat{u}|$ {Compute hypothesis quality score (minimum value)}
11: **if** ($s' < s_i$) **then**
12: $v_i = \hat{u}$, $s_i = s'$
13: **end if**
14: **end for**

We chose to implement our normal estimation using the RANSAC method since it is robust to some edge effects which PCA is prone to. The RANSAC method shows a marked improvement (Figure 3) as compared to the PCA scheme (Figure 2) at discontinuities or corners in the point cloud (Figure 1). It exhibits a change in direction of the normals which mimics the physical surface.

The PCA method relies on using calculation of eigenvectors and eingenvalues of the covariance matrices generated using neighbors of a query point. Query points can be all the points in the given point cloud.

The RANSAC (Random sample consensus) on the other hand chooses a random subset of the relative distances from the neighbors of the query points. We further create a set of hypothesis which are the outer products of these (randomly chosen) relative distances amongst each other. Using the hypothesis we generated we calculate the inner product of each of the hypothesis for each point with the relative vectors to arrive at a set of scores per point. The lowest score gives us our candidate normal.

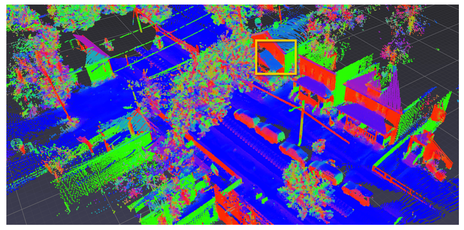

Figure 1: Point Cloud with region of interest

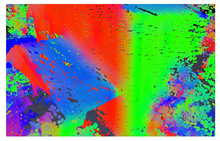

Figure 2: PCA

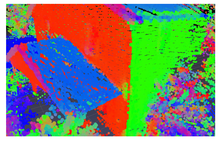

Figure 3: Robust

Division into Kernels

The code was broken down into kernels. Brief description of the kernels used follows:

- FLANN NN search – This kernel is a call to the nearest neighbor search as implemented in FLANN. Line 2 in Algorithm 1.
- Populate_Y – This kernel generates a matrix whose rows contain relative vectors of the current query point to all its neighbors. So, the matrix size is $N \times K_{neigh}$. It also normalizes each of these vectors. Lines 3 through 6 in Algorithm 1.
- Calculate_normals – This kernel calculates the cross product of two randomly selected vectors from $\mathcal{Y}$ matrix for all points. The number of generated cross products is K_{samp}. They are also normalized. Line 9 in Algorithm 1.
- Dot_products – This kernel calculates dot products of each query point with each of its candidate cross products to generate K_{samp} scores per point in the point cloud. Line 10 in Algorithm 1.
- Best_normal_search – The normal vector (cross product from calculate normals kernel) with the minimum of these scores is chosen as the normal for a query point in this kernel. Lines 11 through 13 in Algorithm 1.

Results

In the adjoining plots we show how the execution times vary for both robust normal calculation and the preceding calculation of nearest neighbors on the GPU. In Figure 4 we show the time taken by the FLANN implementation to calculate nearest neighbors. This is common to both CPU and GPU implementations. In Figures 5 nd 6 we have the plots for the execution times for CPU and GPU implementations of normal estimation.

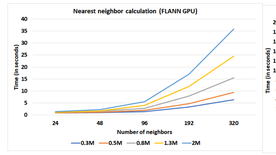

Figure 4: Nearest neighbor calc. using FLANN

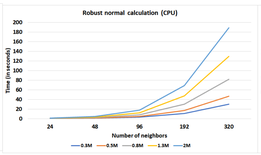

Figure 5: Robust normal calculation on the CPU

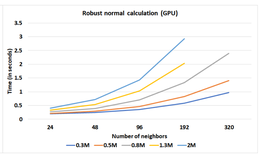

Figure 6: Robust normal calculation on the GPU

Discussion

In Figure 7 we can notice that the speedup can be more than an order of magnitude when comparing CPU and GPU implementations. However, for larger sizes of point clouds and larger nieghborhoods our method requires more memory on the GPU.

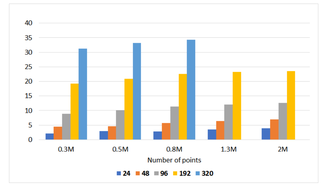

Figure 7: Speedup for robust normal calculation

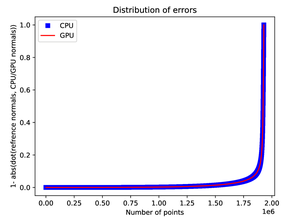

Figure 8: Comparing error distribution with respect to reference

In Figure 8 we plot the distribution of errors of our method versus the reference which comes from a PCA based normal estimation implementation. We did this for many different runs of our method and we saw the same kind of error distribution.

Conclusion

In conclusion we would like to state that using the GPU for calculating surface normals can be beneficial. If we take a look at the profiling information for the two implementations

Kernel Name	Percentage execution time	
	CPU	GPU
FLANN NN search	54	78
Populate_Y	13.2	11.7
Calculate_normals	5.9	7.4
Dot_products	46	2.5
Best_normal_search	0.7	0.2

Table 1: Dataset with 2M points (63 neighbors). Execution on CPU and GPU

We can notice how most of the time on the GPU is spent in calculating the nearest neighbors. The normal calculation subroutines on the other hand are dominated by the data intensive steps in the beginning of the calculation. On the CPU the calculation of dot products looses heavily to the GPU since on the GPU all dot products are calculated in parallel whereas the CPU only has 12 threads for parallel execution.