Near Real-time Pointcloud Smoothing for 3D Imaging Systems

Colin Lea
CS420 - Parallel Programming - Fall 2011

Abstract **--** In this project a GPU-based implementation of Moving Least Squares is presented for smoothing 3D pointclouds. We used an Xbox Kinect to generate spatial data and coded our algorithm using CUDA with the Thrust library. Our implementation uses an organized set of points and can be computed at ~7 Hz on a Nvidia Quadro FX 4800. While perhaps not directly comparable, it has a speedup of between 30-60x when compared to the non-organized serial solution currently in PCL. Additional ways to harness greater performance are also discussed.

I. Introduction

With the recent introduction of low-cost 3D imaging systems, such as the Xbox Kinect, there has become an increased interest in developing methods for using this data in object analysis and activity recognition. Traditional techniques for solving these problems often rely solely on image sequences and could benefit from incorporating spatial data. The Kinect is unique in such that it is capable of generating depth information alongside video. While there have been other systems that do this for more than a decade, including stereo vision and structured light setups, the ease of use and recent introduction of higher quality data tools has made it easier for researchers to work with.

The additional dimensionality is beneficial for developing spatial detectors and other techniques for objection recognition, however, it has increased the amount of computation required for running these algorithms. Even modern computers, like the ones we test on, are insufficient for running basic methods in near real-time. This time requirement is necessary for use in areas such as robotics and surveillance. For reference, a serial implementation of our technique takes between 5 and 20 seconds per iteration on a CPU. Using additional computational resources, such as the graphics card, it is possible to introduce significant speedups. In Section Two we discuss how the uniform, independent structure of this type of data behooves you to run each set of calculations in parallel.

In our application we are interested in employing a smoothing algorithm to reduce the amount of noise in our data. This is necessary for a variety of higher level techniques for feature detection and scene reconstruction. The specific algorithm we implement is Moving Least Squares (MLS). It was our hypothesis that this technique would see a significant performance increase due to the parallel nature of the algorithm. The general idea of this method is that for each point in space we want to find an ideal "true" position based on neighboring locations. Point Cloud Library (PCL) [1] is used to interact with the data and the algorithm is implemented using CUDA with the Thrust library. For efficiency reasons we modify the original, generalized MLS algorithm to suite working with the Kinect. In Section Three we evaluate this algorithm on multiple data sets for both speedup and scaleup using two hardware systems. We find that on average we obtain about 7 frames per second using a Nvidia Quadro FX 4800 graphics card. Section four includes a general discussion of our methods, results, and directions for future research.

II. Parallel Design

The goal of our algorithm is to smooth a pointcloud coming from the Kinect in near real-time. The input is a set of 307,200 points which contain x, y, z positions and rgb color values. This is stored in a 1D array as a set of PointXYZRGB structs using PCL. Our data is essentially an unwrapped 640x480 depth and color image, thus intuitively it can be thought of as a grid. The output should contain the same number of points with new, refined positions using the same data format. It is not necessary to handle the color

information individually - we can simply extract the color-to-point mapping from our input image and map it to our output.

The Moving Least Squares algorithm is implemented in CUDA to find these positions and is based on the mathematical techniques in [2]. As depicted in the following figure, there are three key steps that must be performed on every point: find the nearest neighbors, compute the surface normals, and reposition the original points. The general approach we take is based on data decomposition, which is implicit in our usage of CUDA. The overall pipeline is as follows. When we receive a pointcloud from the Kinect we first convert it to a data type compatible with CUDA (the default cloud type does not compile). It is then copied to the GPU as a global variable due to its large size of approximately 4 Mb. The GPU runs a single kernel on each pixel using the steps described below and finally outputs it back to the CPU.

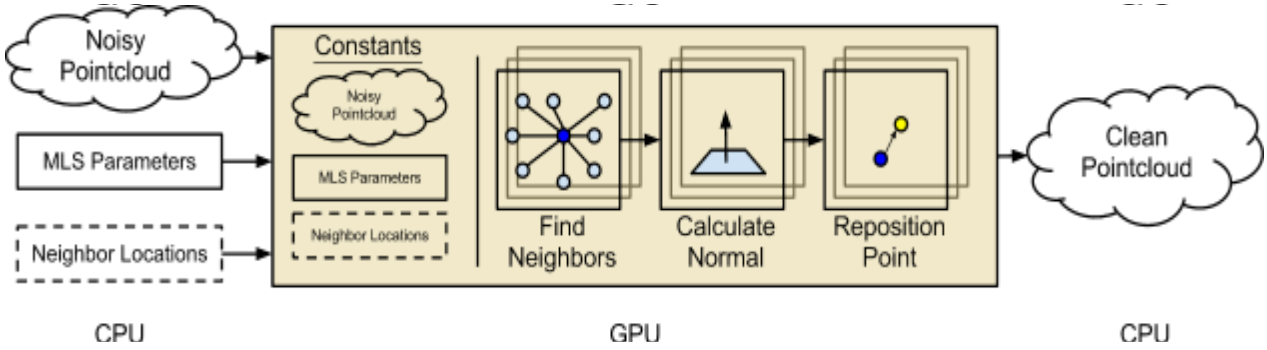

Figure 1. Pipeline from preprocessed data to final smoothed pointcloud.

Nearest Neighbors

The original goal was to generalize the algorithm to any pointcloud, not just one generated from an Xbox Kinect, however, for time and efficiency reasons we decided against. In our implementation the neighbors are simply defined as the set of k-connected points. If you think about this as being a grid then in the one-connected case the neighbors are the set of 8 surrounding points. In the general unorganized[1] case, to find a set of closest points using a naive approach requires N*N sets of calculations where N is the number of points in our data set. A more efficient method is to implement an oct- or kd-tree which greatly decreases the time to find neighboring points, but still requires much more computation than our approach. This general method is what the currently available serial implementation in PCL takes. While it is capable of working on a larger variety of data, including unorganized sets of points, it is unnecessary when working with pointclouds coming from systems like the Kinect.

The pointcloud can be interpreted as a grid, thus we know where the potential nearest neighbors are located. The relative locations are constant for all points in the dataset but vary depending on the connectivity, which is a user-defined input. To speedup computation an array of relative indicies can be generated on the CPU and sent to shared memory on the GPU. This prevents memory bank collisions while reading shared memory and decreases the processing time. A check is done on each of the neighbors to ensure they are within a specified distance to the current point. Our model requires at least three neighbors to fit correctly, so if there are too few points then the function returns and the selected point is set to a null value. The set of close neighbors is stored in the current thread in registers and thus can be accessed very fast.

Surface Normals

[1] In this context the points in an "organized" pointcloud have a known and systematic sequence, such as a raster scan of an image. In an "unorganized" pointcloud the voxels can be randomly placed in the data array.

Assuming there is a sufficient number of points, the surface normal is calculated using eigen analysis on the neighbor set. A 3x3 covariance matrix showing positional variation is determined by first finding the centroid of the set and then calculating covariance based on the zero-centered neighbor locations using the equation below. This matrix is symmetric so for efficiency reasons only the upper triangular region is calculated. The lower triangular elements are simply copied from the upper section.

$$q_{j,k} = \frac{1}{N-1} \sum_{i=1}^{N} (x_{i,j} - \overline{x_j})(x_{i,k} - \overline{x}_k)$$

Equation 1. Covariance calculation for each element

The three eigenvectors of the covariance matrix are orthogonal and define the current point's coordinate system. The smallest eigenvalue/vector pair corresponds to the surface normal. Intuitively, this makes sense because the smallest eigenvalue represents the direction with least variance and the neighbors are constrained on the surface of our model. This means there is little variation in the z-axis of the local coordinate system.

All surface normal calculations are done in a the same CUDA kernel as the neighborhood selection. The function to calculate the eigenvectors of a 3x3 matrix is taken from PCL code based on the Eigen3 library.

Repositioning Points

The repositioning algorithm is a set of equations that projects the selected point onto the plane defined by the surface normal and then extrudes it based on weightings from the other neighboring positions. The projection is simply performed by finding the distance to the plane and subtracting that value in the direction of the normal. A three degree polynomial is used to fit a surface to the neighbors. The center point is then re-projected outwards based on the correlation between the neighbors and this surface. See [2] for more details.

This repositioning is mathematically dense and hence should provide good performance. The parallel implementation is done in similar fashion to the serial method except that CUDA data structures, like float3, are used instead of the original arrays.

Additional Details

The current state of parallel computing in PCL as of release 1.3 is fragmented. There are two directions: coding directly with CUDA and using CUDA with the Thrust Library. Thrust is a higher-level templated interface for programming on the GPU which obtains similar performance to raw CUDA code [3]. No GPU code has been released in the Binary build of PCL and there is currently frustration due to the lack of documentation for either of them. For example, the only place to find system requirements is by sifting through posts on the web forum. It was there where we found the straight CUDA implementation requires a Fermi or later graphics card. For three reasons we chose to implement our algorithm in line with previous Thrust-based methods: (1) we didn't have access to a Fermi card, (2) because of PCL dependencies we were not able to compile code using traditional CUDA techniques, and (3) re-implementing the fundamental point cloud data structures to work with non-Fermi cards would have required too much time for the scope of this project.

While we could have chosen to parallelize Moving Least Squares using other architectures, such as using MPI or OpenMP, the potential speed increases using CUDA were far greater. Our approach relies only on a neighborhood region, thus it naturally works as a data decomposition technique and is suited to the GPU.

III. Evaluation

Three experiments were designed to test for speedup and scaleup. The speedup test evaluates a typical use of our software on two hardware setups. The second test experiments with a varying neighborhood size to evaluate different real-world run times. In the final test we artificially increase the size of the data to see how well our algorithm scales. We also show a sample output and discuss results from profiling the code using the CUDA Visual Profiler.

For practical reasons, our experiments were tested on two hardware setups. Evaluation is performed with both a serial and parallel implementation of Moving Least Squares. While we would like to show more data points for additional computers, there is a long setup time to get the whole system up and running. Depending on the machine, installing the Kinect software, Point Cloud Library, CUDA, as well as all of their associated libraries can take multiple hours. We have seen problems installing these on different operating systems and even with different CUDA graphics cards which makes the operation non-trivial. We felt that it was more worthwhile to develop better documentation and experiments than spend hours trying to get our code running on an additional computer.

The first computer in our setup is a 2007 Macbook Pro with a Nvidia GeForce 8600M GT, Intel Core 2 Duo clocked at 2.16 Ghz, and 4 Gb of RAM. The graphics card has 128 Mb of memory and 32 processing cores. This will be referred to as MBP.

The second setup is a server in the Laboratory for Computational Sensing and Robotics with six Nvidia Quadro FX 4800s, three quadcore Intel Xeon E5645s clocked at 2.4 Ghz, and 12 Gb of RAM. The graphics cards have 1.5 Gb of memory and 192 processors each, however, only one of these units is being used at a time. This will be referred to as Balaur.

Time per iteration is computed based on the total time from the start to end of the whole pointcloud function but not including visualization. Our program is event based; when a new pointcloud is received from the Kinect a callback function is performed. A Boost mutex (lock) is used to prevent multiple pointclouds from interrupting each other. The timer starts at the beginning of this function before any data is sent to the GPU and ends after the MLS algorithm completes and the data is back on the CPU.

Experiment 1: Kinect Stream
Our first experiment evaluates speedup and represents how our system is intended to be used. Data is streamed from the Kinect at a resolution of 640x480. The neighborhood is generated based on 3-connectivity which includes 48 neighbors plus the selected point. All trials are computed using the Kinect with a similar scene. There are approximately the same number of expected MLS computations for all setups.

Note that while we use both serial and parallel implementations, it is not completely fair to directly compare the run times. The serial implemention is a general method that works on an unorganized set of points. Our parallel method is meant for an organized set of points, such as that generated by the Kinect. The serial implementation constructs a kd-tree for finding nearest neighbors which is unnecessary for our goal of exploring real-time 3D feature detection.

The serial implementation finds the nearest neighbors based on the radial distance from each point. Our connectivity-based approach has an implicit non-uniform distance between points due to the Kinect's camera model; the farther from the device the bigger the distance between pixels. The depth granularity is calculated by $dz = (D/2)^2$ so at out point of interest, 2 meters away, it is approximately 1 cm. The total radial search distance is defined by k*dz where k is the connectivity. Thus for 3-connectivity we set distance to 3 cm.

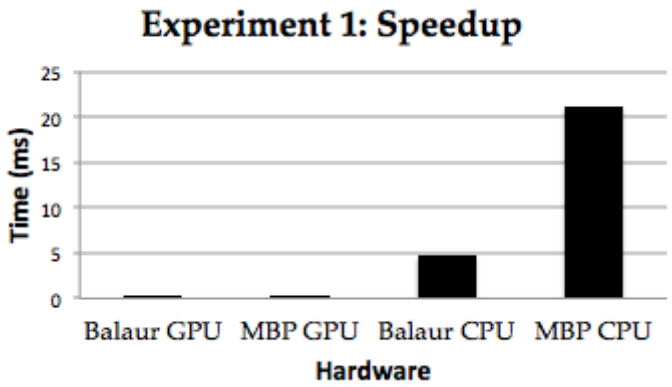

Figure 2. Comparison of calculation times for our implementation versus the serial method.

Figure 2 clearly depicts the large difference between serial and parallel implementations. Using the equation for speedup, $S = \frac{T_{Small}}{T_{Large}}$, Balaur GPU is 2.57x faster than the MBP GPU and 32.89x faster than the Balaur CPU implementation. The MBP GPU version is 57.61x faster than the serial version. The average calculation times for all four systems are listed in Table 1.

Hardware:	Balaur GPU	MBP GPU	Balaur CPU	MBP CPU
Time:	142.89 ms	368.17 ms	4.76 s	21.21 s

Table 1. Calculation times for serial and parallel implementations of Moving Least Squares

While it is not appropriate to claim the large gains are solely due to the parallelization, this experiment shows that using a a GPU and a simpler search method can provide tremendous speed improvements over serial, tree-based implementations.

Experiment 2: k-Connectivity

In this experiment we look at how varying the number of neighboring elements in our set increases the overall GPU time. Tests are performed using connectivity ranging from one through seven. Note that the number of elements in the set is quadratic. In our notation we include the center value as well as the neighbors, thus the count is defined by $N = (2k+1)^2$ and contains the connectivity-to-element mapping {1:9, 2:25, 3:49, 4:81, 5:121, 6:169, 7:225}. To further clarify, for a 7-connected run there are 225 points that must be operated on to find the true neighborhood set and calculate the covariance.

The experiment is performed on Balaur running only the GPU code. Results are shown in Figure 3. The solid line indicates the averaged GPU computation time at each connectivity over 25 iterations. The dashed line depicts the number of neighbors in the set and is used show how the rise in time correlates with the neighbor size.

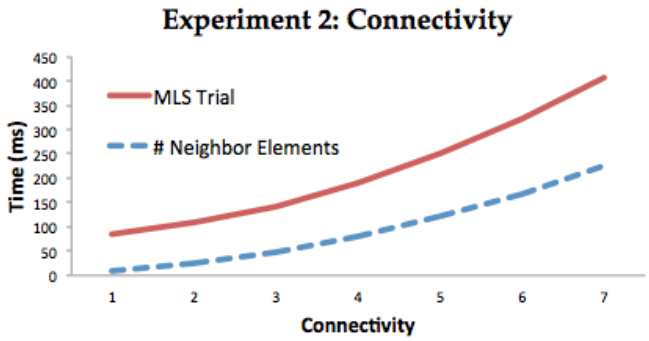

Figure 3.Comparison of GPU time for varies levels of neighbor connectivity

The results are congruent with our expectations. The neighborhood component in the kernel is the only part that varies with size so we assumed the total time would vary linearly with the size of the set. When plotted against the total neighborhood count the line is linear with a correlation of R^2=0.96. When considering the extremes of the neighborhood size, it is not excepted that it continues linearly. Once the registers and shared memory fill up then the neighborhood data will have to be stored on global memory which is much slower. However, for small values, as shown here, we see the necessary framerate for applications requiring near real-time calculations performance.

Experiment 3: Scaleup
Scaleup was evaluated by artificially increasing the amount of data processed by the GPU on Balaur. The general equation for scaleup is defined by the ratio of time it takes for one processor to compute and the time for N processors to compute. It was established that the GPU on the MBP has 32 cores and Balaur has 192, thus there is a difference of a factor of six. This means for scaleup comparison there should be six times as many sets of operations being performed on the larger computer. In our code this is accomplished by setting the pointcloud size to 6*307,200 (1,843,200 points) and replicating the scene six times. Connectivity was set at 3 for a total of 49 neighbors.

Hardware:	Balaur GPU	MBP GPU
Time:	809.35 ms	368.17 ms

Table 2. GPU computation times to measure scaleup. Balaur processed six times as many points as MBP.

Table 2 shows the average GPU computation time over 25 iterations. The scaleup is calculated by the ratio of the two numbers and in our case is 0.45. This is lower than expected but can be explained by the problem found in the profiler, discussed later. To preface, a significant number of threads are not active because of the large amount of data being stored in the registers. This effect is magnified when six times as much data is being processed. We are interested to see the scaleup when this problem is fixed. It is anticipated that the scaleup should increase to more accurately reflect the processor increase and more importantly architecture changes between the MBP and Balaur GPUs. Based on the increases in memory and other significant hardware changes we thought that scaleup would reach closer to one.

Sample Output
Figure 4 depicts a raw and smoothed pointcloud with a person sitting at a desk on the left with a table with several monitors on the right. Note that the images are from slightly different vantage points so, particularly on the right side, the smoothed pointcloud looks less dense. Although the output is not perfectly smooth it does decrease the overall noise. This is especially apparent on flat surfaces like the ground plane which appears to have a variance of about half of the original. While we do not empirically evaluate the Moving Least Squares algorithm, previous comparisons show that the surface normals on the

smoothed pointcloud have significantly less noise than the original data [4].

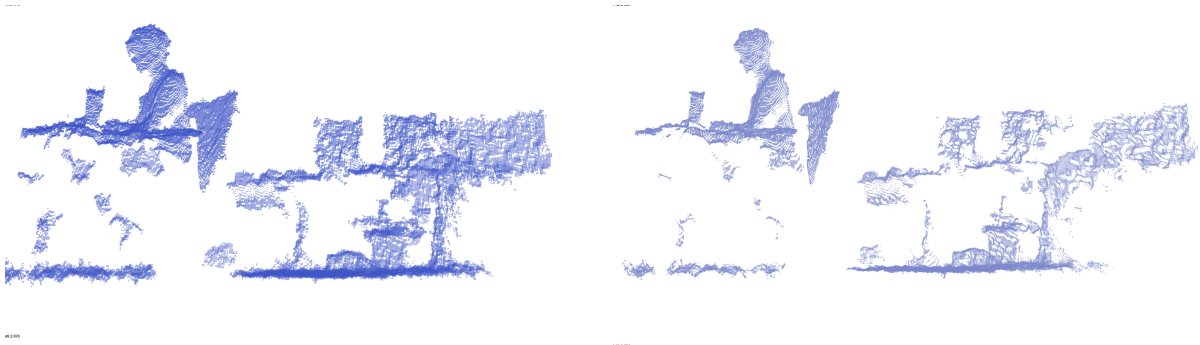

Figure 4. (left) Original Pointcloud (right) Smoothed pointcloud

CUDA Visual Profiler

The CUDA profiler has revealed that there is room for improvement. In the current state, too many registers are being used per thread which is limiting the number of active blocks in each streaming multiprocessor (SM). Less than half of the threads in each SM are being used for the same reason. By decreasing the amount of data held in the registers a larger number of threads can become active. Profiling highlighted a problem in our code were each thread is storing its own set of relative neighbors. The relative neighbors is constant between threads and there is no reason to store this data in so many locations. Eliminating this memory hog should dramatically increase the amount of space available for storing local data such as covariance and neighborhood weights.

While using Thrust made programming in CUDA slightly easier, and prevented us from having to re-implement the PCL data structures, it also removed the necessity to define grid and block sizes in the kernel. If we were to program in CUDA directly then we could further optimize these parameters using the provided profiler.

Figure 5 depicts the time spent doing each type of operation on the GPU. As expected, most of the time is in the main function, "launch_closure_by_value-3," which is our MLS kernel. We also find that the memory transfers are negligible - each is less than 3% of the total.

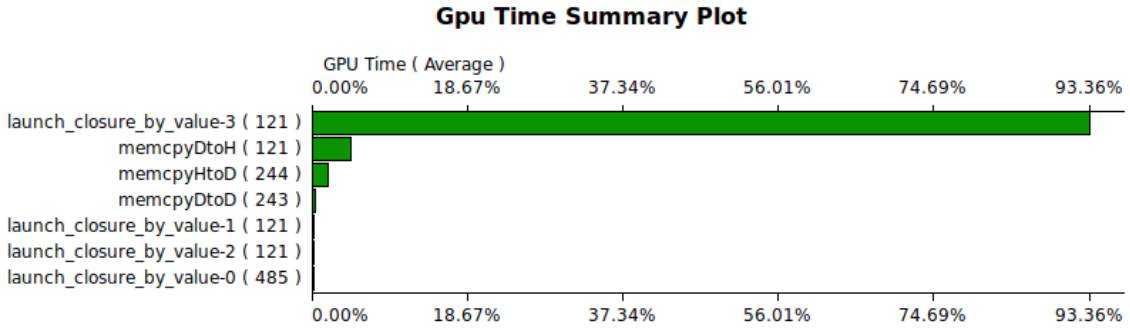

Figure 5. Summary of GPU time by function running experiment 1 on Balaur.

IV. Discussion

Although we were not able to demonstrate real-time (30 FPS) performance, our GPU implementation is adequate for many robotics and computer vision applications that suffice with near real-time. While our current method is not the most efficient method of performing a parallelized Moving Least Squares, it has provided insight into how we can develop a better performing implementation. The CUDA profiler in particular showed the critical weak point in the system: there is too much information being put in registers. By moving that data elsewhere it may be possible to achieve notably better performance. Additionally, there are other techniques that could be investigated to improve performance. For example,

the depth data could be split into smaller chunks that are sent to specific multiprocessors. Feasibility of this depends on the specific GPU used and other implementation details to know whether or not there is sufficient space in shared memory. A addition technique to consider is operating on unsigned integer values instead of on floating point depth. This would significantly reduce the amount of space needed to store the pointcloud on the card.

It would be interesting to compare a GPU version of the generalized MLS technique to ours. There is a GPU-based octree implementation in PCL which could be used to develop one, however it requires a CUDA 2.0+ capable graphics card which was not available to us. Given that we were not able to get full re-time performance it is unlikely that a generalized version would perform with adequate speed. There is an interesting idea we considered to generalize MLS without implementing a time-consuming tree structure. The idea is to evaluate our algorithm at multiple viewpoints. This would be particularly effective if there are multiple 3D imaging systems that are being merged together. Either this algorithm could be applied to each stream independently or the streams could be merged and then smoothed. The latter could be done by generating a range image at multiple viewpoints and the performing MLS on each. This could easily work for any convex set of points. This could be especially useful for extracting objects from a scene or doing feature detection. This is something to explore in future work.

Because of the increased size of pointclouds compared to RGB images (in part due to the floating point numbers) it is difficult to write algorithms that run in real time. Our results are congruent with recent trends in robotics that demonstrate efficient perception techniques using the GPU. While MPI or other paradigms could potentially be used in real-time in some computer vision applications where a large server rack can be stored on-site, the small space on most robots limits high performance on-board computing. Additionally, cloud computing resources are ill-suited because of the large communication overhead coming from the large volume of sensor data.

V. Conclusion

In this project we demonstrate that 3D pointcloud smoothing can be performed in near real-time by running code on the GPU. The code is currently available online on GitHub [5].

VI. References

[1] R Rusu and S Cousins, "3D is here: Point Cloud Library (PCL)." IEEE International Conference on Robotics and Automation (ICRA), May 2011.
[2] S. Fleishman, D. Cohen-Or and C. T. Silva. "Robust moving least-squares fitting with sharp features" ACM SIGGRAPH Papers, 2005.
[3] "A Short Test on the Code Efficiency of CUDA and Thrust." May 22, 2010. url: http://code-saturne.blogspot.com/2010/05/simple-test-on-efficiency-of-thrust.html
[4] "Smoothing and normal estimation based on polynomial reconstruction." url: http://pointclouds.org/documentation/tutorials/resampling.php#moving-least-squares
[5] Colin Lea's Kinect Github repository. url: github.com/colincsl/Kinect-Projects/tree/master/MovingLeastSquares_PCL