Title:
GPU Accelerated Algorithms for CAD / Point Cloud Digitizer Data Registration

Authors:
Venu Kurella, kurellv@mcmaster.ca, McMaster University
Bob Stone, Bob.Stone@originintl.com, Origin International Inc.
Allan Spence, adspence@mcmaster.ca, McMaster University

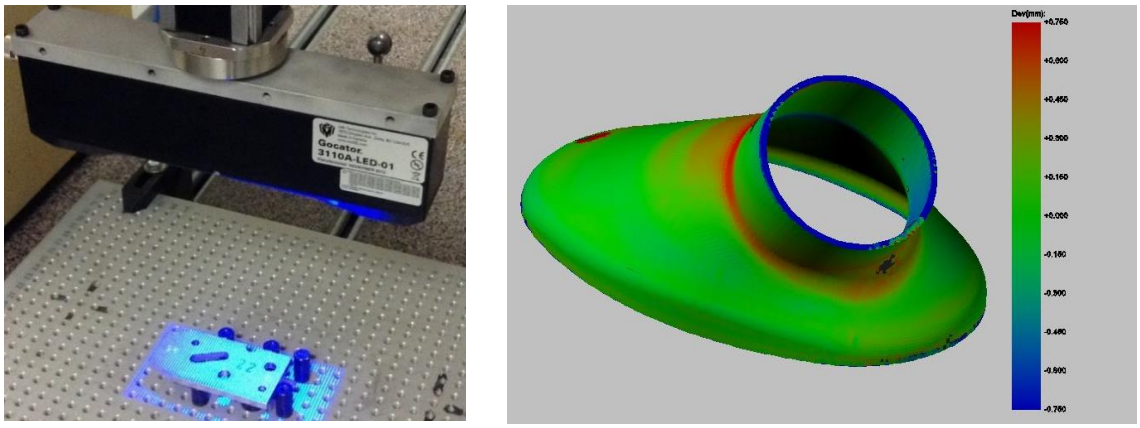
Keywords:
parallel computing, Graphical Processing Unit, GPU, point cloud registration, point-facet matching

DOI: 10.14733/cadconfP.2016.327-331

Introduction:
Automotive sheet metal stamping supplier production rates approach one part every 15 seconds. With increasing demand that comprehensive geometric quality conformance information be communicated to the final assembly plant prior to shipment, the associated digitizing of millions of points requires more computationally efficient analysis algorithms. For example, an industrial blue LED snapshot sensor (Fig. 1(a)) can acquire 1 million points per second. At a 0.1 mm nominal point spacing, for even small part areas, many millions of points need to be registered with the 3D coordinate system of the CAD nominal surfaces. The memory and computing power needed to perform this analysis at part production rates far exceeds the capacity of the conventional personal microcomputers. This paper investigates the alternative of using parallel Graphical Processing Unit (GPU) hardware.
Use of this hardware exploits the massively parallel architecture of a GPU to accelerate data intensive computations. Designed for high graphics intensity CAD and gaming, a GPU has a complex memory and processing architecture. Hence effective programming resource allocation and utilization is much more complex. Therefore, existing serial algorithms, which are not written to run on a GPU, must be extensively rewritten. Compared to visually appealing but approximate gaming applications, dimensional metrology applications require that high accuracy be maintained throughout. For industrial acceptance the research described herein was implemented using an HP Z440 desktop engineering workstation, equipped with Intel Xeon E5 processor and 16 GB RAM. The added NVIDIA Tesla K40 GPU card has 2880 CUDA cores and 12 GB DDR5 RAM. The GPU algorithms were integrated as a Dynamic Link Library (DLL) with the Origin International CheckMate software (www.originintl.com) added to Autodesk Mechanical Desktop, and running under Microsoft Windows 7. Compared to a single Intel CPU thread, a speed-up of 171X on a 400,000+ point cloud registered to a 1.2 million facet CAD model was achieved, reducing wall clock processing time from 6 minutes to 2 seconds.

Main Idea:

GPU Computing in CAD and Dimensional Metrology
Early GPU computing required researchers to mask arithmetic operations as graphical tasks to perform computations on CAD parts [12] or tool paths [3]. The NVIDIA CUDA programming language revolutionized GPU computing [16]. It led to applications like rendering [6], filtering [7] and collision detection [13]. Sheet metal strain measurement was reported by Kinsner et al. [9]. Other computational applications such as distance queries between NURBS surfaces [11] and feature-fitting of geometric primitives [15], showed respectively 300X and 18X speed-up. While 2.5X - 1000X speed-up achievements are reported in literature [14], 20-30X is considered worthwhile. Erdos et al. [4], suggest GPU computing for fast mapping CAD with point cloud data. While Iterative Closest Point (ICP) [1] like registration



(a) (b)

Fig. 1: Point cloud data from digitizers: (a) A blue LED snapshot sensor (www.lmi3d.com) mounted on a McMaster laboratory CMM (b) Matching the CAD facets with the digitizer points as a color map.

methods have already been implemented on GPUs [14], estimating deviation for each and every point/facet in the registered data is more computationally intensive. In this work, we investigated speed-up of a point-facet matching algorithm which estimates deviation for each facet and outputs the results as an informational color map (Fig. 1(b)). Brute force matching models like this have been suggested for accuracy [2]. A smallest sphere distance finding algorithm, that is similar in nature, showed 5X speed-up with naïve GPU implementation [8]. We show that a careful algorithm optimization delivers an impressive 57X speed-up on an inexpensive NVIDIA GTX480 gaming GPU, with further improvement to 171X speed-up using a Tesla K40 and more advanced memory management.

CPU Algorithm

The input for the point-facet matching algorithm is actual scanner data points with normals and the CAD model data as facets with detailed geometric information. Output is the average deviation between a facet and its matching points shown as a color map. In this method, the following computations take place for every model facet (Fig. 2). The registration algorithm is based on the well-known ICP [1] method. Details that follow are specific to the point-facet distance finding subtask that was implemented in parallel on the GPU. Because of the expected high number of digitizer points as compared to the size of the CAD facets, the algorithm begins with transforming the facets into the digitizer part coordinate system. The transformation matrix is initially chosen by manual matching of a few widely separated points chosen from both the digitizer and facet data. This is followed by binary search of digitizer points to find the closest point, j, by distance, to the CAD facet. The final step is a refined search, using matching parameters, in the neighborhood of j, to find all possible matches whose deviation is within a given threshold. The average of the deviations is calculated and reported as color map. Both the binary and neighborhood search algorithms are intricate and time consuming due to loops, branches and many memory and function calls. The worst case complexity of the algorithm is $\boldsymbol{O}(nm)$, where n and m are respectively the number of CAD facets and digitizer points.

Challenges

This method poses many challenges at the algorithm and memory levels: a) it is heavily memory dependent with complex data structures storing about 3 MB of actual point cloud data, which far exceeds the capacity of GPU on-chip registers or shared memory components; b) it exhibits poor instruction-level-parallelism, and this is further affected by intermittent calls to external functions.

Experimental Set-up

Results from a single core of an Intel i7 CPU are compared with an inexpensive NVIDIA 480 core GTX480 gaming card and a server grade Tesla 2880 core K40. Programming was done in Visual Studio 2013 with NVIDIA Nsight 4.1 and CUDA 6.5 on Microsoft Windows 7. A scanned part data with 424307 points was provided as input, and a CAD model of the part was used to generate the facets. The maximum facet

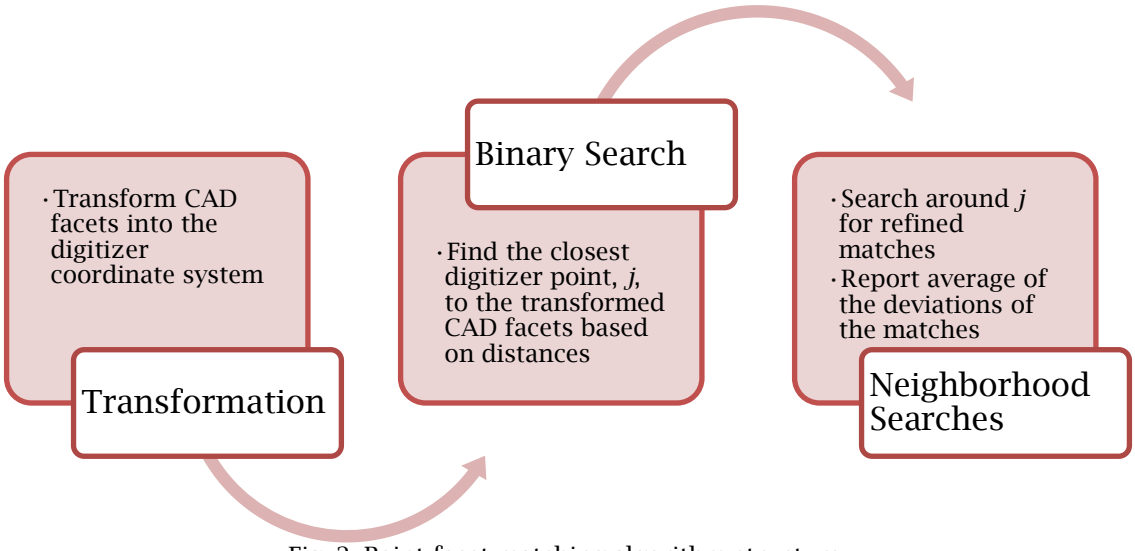


Fig. 2: Point-facet matching algorithm structure.

edge length parameter was varied to get three model data sets. Each of the timing results are averaged over 100 trials with the speed-up (s-u) calculated as below:

$$speed\text{-}up\ (s\text{-}u) = \frac{time\ taken\ by\ the\ CPU}{time\ taken\ by\ the\ GPU} \quad (1)$$

Algorithm and Memory Optimization:

The CPU-optimized algorithm was re-written to make it GPU efficient. Simplified data structures were designed for efficient memory access. Instruction level parallelism (ILP) was achieved using techniques to minimize branching. Experiments on different memory paradigms were conducted to achieve efficient handling of the dense data. Flags are used to preserve the CPU execution order of the algorithm. Details of these enhancements follow.

Instruction Level Parallelism (ILP)

Point-facet matching has two kinds of conditional branches: filtering and critical. Several critical branches were bundled together using Boolean operations. Some conditions perform initial filtering of data before moving to the critical tests. Filters save additional CPU calculations, but impede ILP that is crucial for the GPU performance. Based on serial implementation experience, some of these branches were bundled and others were altogether removed. This increases ILP and, thereby, performance. It ensures more computations between the memory accesses and also bundles memory accesses leading to fewer memory reads.

Initial Results

After improving ILP, experiments on the GTX 480 showed promising results. For the 3 data sets, speed-ups of 9X, 19X and 57X were observed. The same implementation run on the K40 GPU achieved speed-ups of 13X, 33X and 63X. While these are significantly higher, more was expected as the K40 has 6 times as many cores as the GTX 480. Additional study was therefore conducted.

Performance Analysis

The study found that the existing algorithm structure does not use the cores effectively. The achieved speed-up was mainly because the K40 has twice as many registers supporting twice as many threads. The algorithm is made up of global memory accesses and arithmetic operations. A global memory access is more time consuming (200-800 clock cycles) than an arithmetic operation (1-8 clock cycles). The CUDA programming model relies on efficiently scheduling these two components to hide the latency and

achieve performance. The higher number of cores in the K40 can only speed-up computations. As first implemented, before a memory access is completed either the computation is already over or a conditional branch is hit. No speed-up was observed because there is not enough computation between the (many) memory accesses to hide the latency. Hence to improve performance, either the algorithm parallelism has to be increased, or the memory accesses times have to be lowered.

Complete Looping

A simple way to achieve near-perfect parallelism is to eliminate branching and loop over all points, instead of a specific neighborhood (of *j* in Fig. 2). This change

- Eliminates the non-contiguous memory accesses of the binary search
- Removes branching in the neighborhood search

Naïve implementation of the complete looping decreased the performance (10%-70%). This is because looping all points results in more total memory accesses. In attempts to overcome the drawback, use of shared and texture memories was attempted to share information of memory reads between the threads. But they could not help as the cache/memory sizes (8 KB/48 KB) of texture/shared memories are very small compared to actual point data size (~3 MB) that has to be looped. Due to these reasons, the complete looping approach failed to improve speed. So attempts were made to lower memory access times as discussed in the next subsection.

Texture Memory

Overall memory access time can be lowered by re-using information. With binary and neighborhood searches, the search range is different for each facet/thread, so shared memory cannot facilitate efficient re-use of information. Texture memory, an unconventional read-only graphics memory, is located off-chip with up to 8 MB capacity. Although located on global memory, it has an 8 KB on-chip cache making it ideal to store small, but frequently used information. Neighboring facets share at least part of the search ranges thus exploiting texture cache. Its implementation takes additional work on the CPU side as textures support only basic data types. Texture memory produced excellent results with fraction-of-a-second computation times for two of the three cases. The results are discussed in the next section.

Max. facet edge length (mm)	Number of model facets	CPU single core	GTX480 naïve memory		GTX480 with texture		Tesla K40 naïve memory		Tesla K40 with texture	
		Time	Time	s-u	Time	s-u	Time	s-u	Time	s-u
2.000	103,966	36.241	4.129	9	1.948	19	2.827	13	0.964	38
1.000	345,592	111.678	6.004	19	3.022	37	3.347	33	0.883	126
0.500	1,188,408	358.03	6.249	57	7.852	46	5.676	63	2.098	171

Tab. 1: Computation time (seconds) and speed-up (s-u) of Tesla K40 and GTX 480.

Results and Conclusions

The computation times and speed-up are summarized in Tab. 1. While naïve usage of memory, itself, showed an impressive 63X speed-up, texture memory further boosted it to 171X. It can be noticed that the 0.3 M facets case takes time less than the 0.1 M case. It is believed that as the number of model facets gets close to the actual facets number (~0.4M), fewer texture cache misses occur leading to a higher speed. Texture memory implementation on GTX480 also resulted in an improved performance for the 0.1 and 0.3 million cases. However, the 1 million case slows down – probably because the arithmetic computations are now slower than the memory accesses.

In conclusion, this work demonstrates the feasibility and the method of acceleration of point-facet matching of a dense and complex data without sacrificing accuracy. It delivers the product as a practical and industrially applicable library compatible with Microsoft Windows. Higher density means more accurate deviations and the GPU showed improvement in performance with increasing density. Future

work would focus on estimating normal vectors for the point cloud data obtained from the multi-sensor inspection system discussed in previous work [17], and the subsequent estimation of deviations using the algorithm discussed in this paper.

Acknowledgements:

This work was financially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Canadian Network for Research and Innovation in Machining Technology (CANRIMT) and a Discovery Grant. Additional funding support was provided by Origin International Inc. (Markham, ON, Canada) and an NVIDIA Hardware Grant.

References:

[1] Besl, P.J.; McKay, Neil D.: A method for registration of 3-D shapes, IEEE Transactions on Pattern Analysis and Machine Intelligence, 14(2), 1992, 239-256. http://doi.org/10.1109/34.121791

[2] Bosché, F.: Automated recognition of 3D CAD model objects in laser scans and calculation of as-built dimensions for dimensional compliance control in construction, Advanced Engineering Informatics, 24(1), 2010, 107–118. http://doi.org/10.1016/j.aei.2009.08.006

[3] Carter, J. A.; Tucker, T. M.; Kurfess, T. R.: 3-Axis CNC path planning using depth buffer and fragment shader, Computer-Aided Design and Applications, 5, 2008, 612-621. http://dx.doi.org/10.3722/cadaps.2008.612-621

[4] Erdos, G.; Nakano, T., Vancza, J.: Adapting CAD models of complex engineering objects to measured point cloud data, CIRP Annals – Manufacturing Technology, 63, 2014, 157-160.

[5] Georgescu, S.; Chow, P.: GPU accelerated CAE using open solvers and the cloud, ACM SIGARCH Computer Architecture News, 39(4), 2011. http://dx.doi.org/10.1145/2082156.2082161

[6] Gunther, C.; Kanzok, T.; Linsen, L.; Rosenthal, P.: A GPGPU-based pipeline for accelerated rendering of point clouds, Journal of WSCG, 21, 2013, 153-161.

[7] Hu, X.; Li, X.; Zhang, Y: Fast filtering of LiDAR point cloud in urban areas based on scan line segmentation and GPU acceleration, IEEE Geoscience and Remote Sensing Letters, 10(2), 2013, 308-312. http://dx.doi.org/10.1109/LGRS.2012.2205130

[8] Inui, M.; Umezu, N.; Shimane, R.: Shrinking sphere: A parallel algorithm for computing the thickness of 3D objects, Computer-Aided Design and Applications, 4360(December), 2015, 1–9. http://doi.org/10.1080/16864360.2015.108418

[9] Kinsner, M.; Spence, A.; Capson, D.: GPU Accelerated Sheet Forming Grid Measurement, Computer-Aided Design and Applications, 7(5), 2010, 675–684. http://doi.org/10.3722/cadaps.2010.675-684

[10] Kinsner, M.: Close-range machine vision for gridded surface measurement, Ph.D. Thesis, McMaster University, Hamilton, Canada, 2011.

[11] Krishnamurthy, A.; McMains, S.; Haller, K.: GPU-accelerated minimum distance and clearance queries, IEEE Transactions on Visualization and Computer Graphics, 17(6), 2011, 729-742.

[12] Kurfess, T. R.; Tucker, T. M.; Aravalli, K.; Meghashyam, P. M.: GPU for CAD, Computer-Aided Design and Applications, 4(1-6), 2007, 853–862. http://doi.org/10.1080/16864360.2007.10738517

[13] Lee, R. S.; Ren, M. K.: Development of virtual machine tool for simulation and evaluation, Computer-Aided Design and Applications, 8(6), 2011, 849–858. http://doi.org/10.3722/cadaps.2011.849-85

[14] Park, S.-Y.; Choi, S.-I.; Kim, J.; Chae, J. S.: Real-time 3D registration using GPU, Machine Vision and Applications, 22(5), 2011, 837–850. http://dx.doi.org/10.1007/s00138-010-0282-z

[15] Ram, M. P.; M., Kurfess, T. R.; Tucker, T. M.: Least-squares fitting of analytic primitives on a GPU, Journal of Manufacturing Systems, 27(3), 2008, 130-135.

[16] Sanders J.; Kandrot, E: CUDA by example: an introduction to general-purpose GPU programming, Addison-Wesley Professional, 2010.

[17] Xue, K; Kurella, V; Spence, A.: Multi-Sensor Blue LED and Touch Probe Inspection System, Computer-Aided Design and Applications, to appear.